Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Architecting SQL/PGQ support in DuckDB

Author: Tavneet Singh (2668222)

1st supervisor: Prof. Dr. Peter Boncz (CWI, VU Amsterdam)
daily supervisor: Dr. Gábor Szárnyas (CWI)
2nd reader: Dr. Benno Kruit (VU Amsterdam)

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

August 19, 2022

Abstract

Data stored in Relational Database Management Systems (RDBMSs) for domains like social networks and e-commerce is often inter-connected and contains graph-like structures. This provides an opportunity for running graph queries and solving graph problems in RDBMSs which has led to the development of SQL Property Graph Queries (SQL/PGQ), an upcoming SQL extension providing graph query support. However, no major implementation of this extension currently exists. The SQL/PGQ extension is particularly useful for analytics, therefore we use an analytical, vectorised open-source database system, DuckDB for its implementation. Architecting the new extension involves major design decisions on storage, data representation, path-finding algorithms, mapping of graph operators and performance evaluation. Our results show that implementing the extension using an intermediate Compressed Sparse Row (CSR) representation for graph storage is scalable and performant. Furthermore, the Kleene star operator can be mapped to a User Defined Function (UDF) allowing an unobtrusive extension and scalable execution.

Acknowledgements

I would like to thank my advisors Dr. Peter Boncz and Dr. Gábor Szárnyas for their invaluable feedback and discussion sessions. They have been patient and advised me well despite numerous delays. Special thanks to Gabor for always checking up on me, checking the draft proposal multiple times, and guiding me whenever things felt bleak.

I would also like to thank CWI group members - Laurens Kuiper, Dr. Mark Raasveldt and Azim Afroozeh for resolving my numerous doubts about DuckDB, databases and programming in general. They were always willing to help whether it was via a zoom call or texts on telegram.

Finally, I would like to thank my family for their continued support throughout this challenging process and my friend Pulkit for proofreading the thesis.

Contents

List of Figures

List of Tables

1

Introduction

1.1 Motivation

The majority of the data in application domains such as finance, e-commerce, and health services in the past decades has been stored in RDBMSs due to their stability, support for complex analytical queries, transactional guarantees and maturity (1). The underlying data in these fields often contains graph-like connections which emerge along tables encoding many-to-many relationships with foreign keys. Analysing this graph data can provide useful insights for fraud detection and recommendation systems (2). However, support for graph operations in queries such as pattern matching and path-finding in RDBMSs using vanilla SQL (SQL:1999) is cumbersome or absent (3).

This has led to the rising popularity of Graph Database Management Systems (GDBMSs) such as Neo4j (4), TigerGraph (5) and Memgraph (6), and query languages like Cypher (7), PGQL (8), Gremlin (9), etc. All these diverse query languages support two major graph operations - pattern matching and navigational queries (10). However, these systems suffer from two major problems: i) Lack of maturity *i.e.* stability and performance as they are relatively new systems ii) Absence of a standard query language. Furthermore, transporting data from an RDBMS to GDBMS is expensive and time-consuming, and leads to graph queries running on stale data due to a lack of synchronisation between the RDBMS and GDBMS.

The above reasons have led to research efforts to directly add graph query support to RDBMSs (3, 11, 12, 13) and creation of SQL/PGQ (14). SQL/PGQ (14) is an upcoming extension to the SQL ISO standard that allows creation of property graphs (15) (a set of edges, nodes, labels and properties) on underlying RDBMS tables through Data Definition Language (DDL) statements. It further supports graph analysis by adding pattern match-

ing and flexible shortest path queries to SQL using Data Manipulation Language (DML) statements. Implementing the extension gives the opportunity to reuse SQL's powerful analytical functionality, and combine graph queries with relational operators. Running the shortest path queries using the extension is intuitive and likely faster compared to writing complex recursive SQL queries that use a naive iterative algorithm instead of specialised graph algorithms (such as Dijkstra (16)).

RDBMSs are suitable candidates for graph processing as graphs can be represented as tables and the majority of deployed data-intensive systems contain an RDBMS as the primary storage of data sets (2). Furthermore, as graph processing systems are often used for read-heavy analytical workloads (2), analytical database systems (e.g. column stores) are primary candidates to serve as the basis of a graph query engine (17). Still, no efficient implementation of graph query support using SQL/PGQ on a database currently exists.

1.2 Problem Statement

1.2.1 Research Goals

The main goal of the project is to architect a minimal and efficient implementation of SQL/PGQ to DuckDB (18), an analytical embeddable database developed at CWI. We have identified the following technical challenges for this project:

1. Adding a minimal amount of new query operators to the database system to be able to execute SQL/PGQ efficiently.

2. Identifying what new algorithms, data structures and parallel execution mechanisms are needed to make these operators efficient.

3. Determining a way of evaluating the properties of our choice quantitatively (e.g. using established Database Management system (DBMS) benchmarks).

4. Implementing the extension as a pluggable module into the database system.

These challenges are non-trivial to solve because there are many design choices that interact with each other and currently, there are no analytical, highly performant graph systems for a baseline reference.

1.2.2 Research Questions

The technical challenges help us define the research questions to be answered in this thesis.

1. How can SQL/PGQ be mapped to relational query operators including the new path-finding operator?
   (a) What design decisions need to be made to scope down the extension for improving performance and feasibility?
   (b) What approach should be selected to ensure changes can be implemented as a minimal extension?
2. What path-finding algorithm is needed to minimally support SQL/PGQ?
   (a) How can the algorithm be integrated into a vectorised query engine?
   (b) What data structure representation will be required for implementing the algorithm?
3. How can the performance of the system be systematically evaluated?

1.3 Thesis Outline

The rest of the thesis is structured as follows. In Section 2, we provide the background information necessary for understanding the thesis. In Section 3, we elaborate and compare our design choices along with the rationale behind choosing them. In Section 4, we provide the implementation details of our selected approach along with challenges faced implementing it in DuckDB. In Section 5, we present and discuss the results of our benchmarks. In Section 6, we provide the related work as part of our literature study. Finally, in Section 7 we conclude the thesis by revisiting the research questions and answering them, and we give suggestions for future work.

2

Background

In this section, we discuss the background and related work for understanding the thesis. First, we introduce the graph data model (Section 2.1.1) and then elaborate on the property graph model (Section 2.1.2). We discuss the SQL/PGQ (Section 2.2) language that we want to support for our extension. We then discuss the algorithms for the path-finding problem (Section 2.3) and the CSR format for representing graphs (Section 2.4). We then describe the DuckDB DBMS (Section 2.5) which was chosen to be extended with graph query support and for performing the scalability experiments. Finally, we describe the graph benchmarks selected for the scalability experiments (Section 2.6).

2.1 Graph Background

In this section, we introduce terms related to graph processing.

2.1.1 Graph Data Model

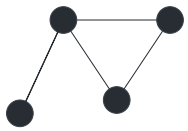


Figure 2.1: A simple graph with 4 nodes and 4 edges.

A simple graph is defined as a tuple G = (V, E) where V is the set of nodes and E is the set of edges. A simple graph is an unweighted, undirected graph containing no graph loops or multiple edges between the same nodes (19). An example of a simple graph is shown in Figure 2.1.

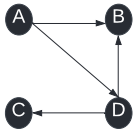


Figure 2.2: A directed graph with 4 nodes and 4 edges.

On the other hand, a directed graph has edges with direction represented by arrows as shown in Figure 2.2. The starting point of the arrow is called the source node (vertex) and the endpoint is called the destination node.

Reachability: In graphs and social networks, a node d is reachable from another node s if there exists a sequence of adjacent nodes starting from s and ending at d. A path in a directed graph is a sequence of distinct edges which joins a sequence of distinct nodes with the edges directed in the same direction. Therefore, in Figure 2.2, node C is reachable from node A through the path (AD, DC) while node A is not reachable from node D.

2.1.2 Property Graph Model

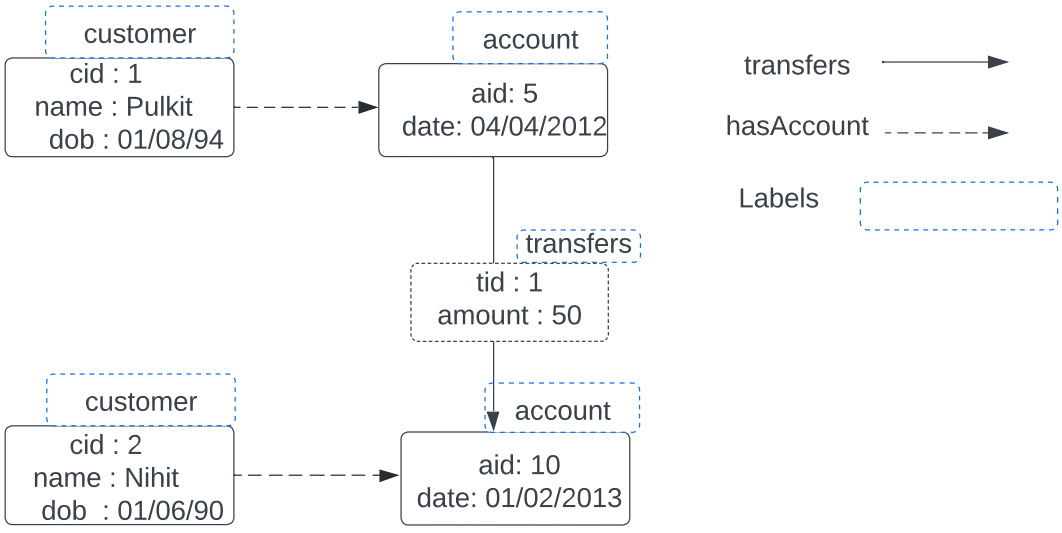


Figure 2.3: Sample Property Graph representing information of bank customers.

The Property graph (PG) model (20) is a popular method to model graph-like data and it is used in graph databases like Neo4j (4), TigerGraph (5), Memgraph (6), and in query languages like PGQL (8) and Cypher (7). The SQL/PGQ extension also uses

the PG model. Property graphs are multigraphs that can have self loops with directed or undirected edges. In the PG model, a graph consists of nodes representing an entity and edges representing relationships between the entities. Both nodes and edges can have multiple properties (key-value pairs) and labels (types).

An example of a property graph for data of bank customers is shown in Figure 2.3, with nodes for customers and accounts, edges for transfers and hasAccount. The labels are shown in blue dashed boxes, node properties in black solid boxes, and edge properties in black dashed boxes. The property graphs can also be represented as tables and the tabular representation is shown in Figure 2.4.

customer table

cid	name	dob
1	Pulkit	01/08/1994
2	Nihit	01/06/1990

account table

aid	cid	date
5	1	04/04/2012
10	2	01/02/2013

transfers table

tid	src_aid	dst_aid	amount
1	5	10	50

Figure 2.4: Tabular representation of Property Graph defined in Figure 2.3.

2.2 SQL/PGQ and GQL

SQL/PGQ is a proposed extension to SQL that supports read-only queries on relational databases and is useful for pattern matching operations, shortest and nested path queries (14). It simplifies the pattern matching operations which would lead to long, complex queries in vanilla SQL(SQL:1999) and it introduces new operators to make the language more expressive (21). The operators introduced are:

- `CREATE PROPERTY GRAPH`: for creating property graph views on existing tables.
- `MATCH`: for pattern matching.
- Kleene star: for reachability computation.
- `MATCH SHORTEST`: for shortest path computation.
- `MATCH CHEAPEST` (optionally): language opportunity (22) for finding cheapest paths if the graph has weighted edges.

Graph Query Language (GQL) (23) is an upcoming new composable graph-specific query language that draws from G-CORE (Section 6.2.4), Cypher (Section 6.2.1), PGQL (Section 6.2.3) and supports CRUD operations and graph data maintenance (*i.e.* updates on graphs) (23). GQL and SQL/PGQ have the same pattern matching syntax. A major difference besides GQL supporting graph updates is that SQL/PGQ will project tables from a graph, while GQL would be used to project graphs from graphs, and treat graphs as first-class objects inspired by G-CORE (14).

An example of the creation of property graphs is shown in Listing 2.1. Furthermore, an example of pattern matching without Kleene star (simple pattern matching) in Listing 2.2, and reachability syntax (with a Kleene star operator) in Listing 2.3. Under reachability (Kleene star), there can be 0 to n-1 intermediate nodes (given n nodes in the graph). Shortest path syntax is similar to the reachability query with the addition of `MATCH ANY SHORTEST` clause. The synatx takes inspiration from Cypher (Section 6.2.1) and PGQL (Section 6.2.3).

```
1 //Property graph is created which is a view on top of existing relational
     tables Person, Technology and Likes.
2 CREATE PROPERTY GRAPH bank_pg
3 VERTEX TABLES (customer, account)
4 EDGE TABLES (transfers SOURCE account DESTINATION account,
5       hasAccount SOURCE customer DESTINATION account)
```

Listing 2.1: Creation of property graphs in SQL/PGQ.

```
1 SELECT a2.id
2 FROM GRAPH_TABLE( bank_pg,
3 MATCH (a1 IS account)-[t IS transfers]->(a2 IS account)
4 WHERE a1.aid = 5
5 COLUMNS (a2.id, t.amount)
6 ) gt
```

Listing 2.2: Sample Pattern matching in SQL/PGQ.

```
1 SELECT a2.id
2 FROM GRAPH_TABLE( bank_pg,
3 MATCH (a1 IS account)-[t IS transfers*]->(a2 IS account)
4 ) gt
```

Listing 2.3: Kleene star operator in SQL/PGQ.

2.3 Reachability and Path Finding Algorithms

Breadth-First Search (BFS) is a common traversal algorithm for deciding reachability and running centrality-based computations (24). During BFS, vertices at a certain level/hop from the source vertex are traversed before moving on to the next level. Plain BFS suffers from random access of nodes, inefficient storage of data, and redundant computation of the visited nodes (25). Furthermore, the Kleene star operator for reachability introduced in SQL/PGQ would amount to a path problem with multiple source-destination pairs which can be sped up by running multiple, concurrent BFSs. MSBFS (25) developed by Then *et al.* is a sequential, multi-source version of BFS that shares traversal information across the concurrent BFSs, uses bitsets to improve memory usage, and reduces random access. It uses an ordered adjacency list representation of the graph to sequentially traverse the graph. It uses three k-wide bitsets to store the state of the vertices with k being set to the cache line width (*e.g.* 512 bits in case of AVX-512). The three bitsets used are:

- *seen*: stores nodes seen so far by each BFS.
- *visit*: stores information per BFS about nodes to be visited in the current iteration of the algorithm.
- *visit_next*: stores nodes to be visited in the next iteration.

The MSBFS pseudocode with comments explaining its execution is presented in Listing 2.3. For further reading on the algorithm, we refer the reader to the original paper (25).

```
1  //G: Graph, B: set of BFSs, S: set of source vertex for each BFS
2  Input: G, B, S
3
4  //initalise only the starting nodes for each BFS as seen and visited for
       that BFS.
5  // seen and visit arrays size = number of BFS,
6  // while size of each element = number of nodes
7  // seen[v] : bit at position i indicates v was seen by bfs_i
8  // visit[v]: determines if v must be visited in the current iteration
9  // visit_next[v]: a set bit i indicating v must be vistied in the following
       iteration for bfs_i
10 // seen[0] = 10 means vertex 0 has been seen by bfs_0; 01 means it has been
       seen by bfs_1
11  for each b_i ∈ B
12   seen[s_i] ← 1 << b_i
13   visit[s_i] ← 1 << b_i
14
15 reset visitNext
```

```
16 while visit ¬ ∅
17
18   //BFS traversal split into two loops for better memory access
19   //all vertices explored sequentially to calculate which BFS explores the
         neighbors of vertices in visit array.
20   for i = 1, . . . , N
21     if visit[v_i] = B_∅, skip
22
23     for each n ∈ neighbors[v_i ]
24       visitNext[n] ← visitNext[n] | visit[v_i]
25
26   for i = 1, . . . , N
27     if visitNext[v_i] = B_∅ , skip
28     // computation performed only once for every newly discovered vertex
29     visitNext[v_i] ← visitNext[v_i] & ~seen[v_i]
30     seen[v_i] ← seen[v_i] | visitNext[v_i]
31     if visitNext[v_i] = B_∅
32       do BFS computation on v_i
33
34   visit ← visitNext
35   reset visitNext
```

2.4 Compressed Sparse Row (CSR) Data Structure

CSR is a compact representation of graphs that reduces random access while accessing nodes and edges as compared to an edge list representation of graphs. CSR consists of two arrays (columns) - the first array for vertices and the second for edges. The second array (of size |E|) is the second column of the edge list graph representation and the first column of size |V| + 1 contains indices for each vertex to index into the second column. The space required is also proportional to the number of vertices and edges of the graph. It is fast for finding neighbours of a particular node but adding or removing nodes or edges requires a change to the entire data structure (26). For our case, since SQL/PGQ in the first iteration does not support any direct updates on the graphs, we can rebuild a (read-only) CSR graph for every `MATCH` query.

A CSR for a sample graph is presented in Figure 2.5. For node i, the adjacent vertices are from $csr\_e[csr\_v[i]]$ to $csr\_e[csr\_v[i+1]-1]$.
For node 0: neighbours are from $csr\_e[csr\_v[0]]$ to $csr\_e[csr\_v[1]-1]$ *i.e.* from $csr\_e[0]$ to $cse\_e[2]$ which are 1, 1, 2 (first 3 elements in csr_e).

The graph in Figure 2.5 has dense node identifiers (0...|V|-1) for faster memory access; the node identifiers can be sparse but in that scenario, we can use another array to relabel

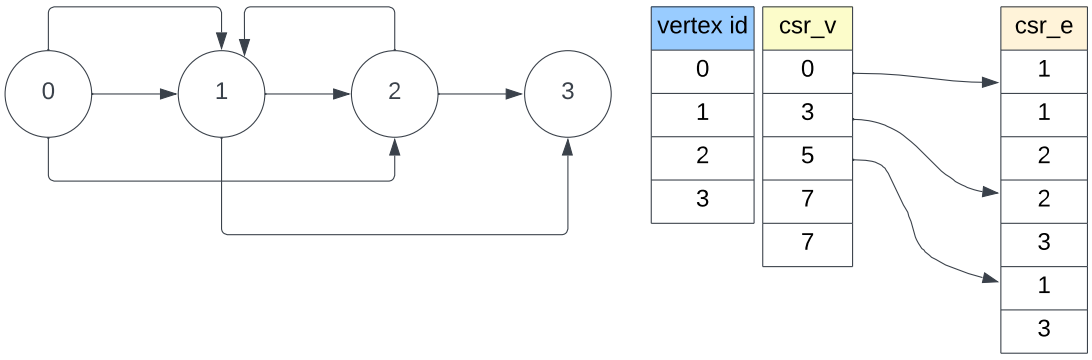


Figure 2.5: Graph and the corresponding CSR representation. Since node 3 is a sink node, it does not index into the csr_e array.

the nodes to dense ids.

2.5 DuckDB

DuckDB (18) is an open-source, embeddable database primarily built for OLAP queries. It supports vectorised execution and is ACID compliant using Multi-Version Concurrency Control (MVCC) (18) supporting hybrid OLTP/OLAP queries. The basic unit of data in DuckDB is a Vector, which is a subset of logical columns (default value is 1024 tuples), thus a table is a list of vectors (27). Vectors in DuckDB support compression, DuckDB supports relational operators as vector operations, and a bit vector is used for faster filter operations on vectors. DuckDB also has inter-pipeline parallelism and supports defining scalar UDFs which are parallel by default. This is advantageous as for architecting a minimal extension, we can leverage User Defined Functions (UDFs) and use the fact that DuckDB already parallelises their execution for scalable results. DuckDB has the following components:

- Parser (❶): parses the SQL statements and converts them to C objects (parse trees). It is heavily influenced by the PostgreSQL parser and uses Flex[1] and Bison[2] to generate the parsed grammar.
- Transformer (❷): transforms the C objects into C++ objects defined by DuckDB.
- Catalog (❸): stores the metadata information (tables, schemas, functions) of the database.

[1]https://github.com/westes/flex
[2]https://www.gnu.org/software/bison/

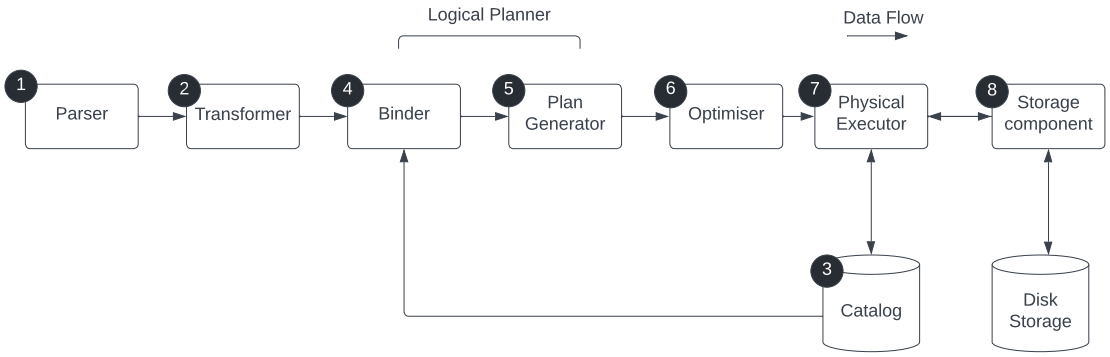


Figure 2.6: DuckDB execution pipeline.

- Binder (❹): resolves the parsed column and table names to the metadata information stored on the Catalog and verifies that the query is semantically correct.
- Plan Generator (❺): transforms the parse tree into logical query operators like scan and projection.
- Optimizer (❻): responsible for the generation of a new logical plan by query rewriting, predicate pushdown and join order optimisations for faster execution.
- Physical Executor (❼): transforms the logical plan into a physical plan. Then push-based execution (28) is called for the physical operators and metadata is stored into the Catalog.
- Storage manager (❽): manages access to the physical data stored in memory and on disk. The executor calls the storage component for accessing base table data. It also contains components for checkpointing, write-ahead logging which are used in error recovery in case of a crash.

DuckDB tables contain rowids, a pseudo column[1] that contains row identifiers based on physical storage. These are dense identifiers starting from 0 for the first row, 1 for the second row and $n-1$ for the last row if the number of rows is n and none of the rows have been deleted. Deleting a row introduces gaps in the rowids that are reclaimed later (*e.g.* during checkpointing), however, rowids are stable within a transaction (29).

[1] https://docs.oracle.com/cd/B19306_01/server.102/b14200/pseudocolumns.htm

SF	Nodes	Edges
0.1	1,700	18,135
0.3	3,900	57,095
1	11,000	226,293
3	27,000	698,627
10	73,000	2,380,850
30	184,000	7,273,036
100	499,000	23,841,591
300	1,254,000	70,144,162
1000	3,600,000	233,283,101

Table 2.1: Person data sizes along with the edges representing their friends in the LDBC benchmark.

2.6 Graph Benchmarks

We describe the graph benchmarks that are used in the thesis.

2.6.1 SNB Business Intelligence Workload

The Linked Data Benchmark Council (LDBC) Social Network Benchmark (30) Business Intelligence is an industry benchmark for measuring the performance of DBMSs for analytical graph data management. The benchmark uses a realistic social network dataset that contains information about people, their friends, comments, their city and country. The business intelligence workload has join-and aggregation-heavy queries touching a large portion of the graph. Since SQL/PGQ is also intended for read-only analytical queries it is convenient to use a benchmark that does not deal with frequent updates and is analytics heavy.

2.6.2 Labelled Subgraph Query Benchmark (LSQB)

The Labelled Subgraph Query Benchmark (LSQB) is a subgraph query matching micro benchmark to test the query optimiser and join performance of DBMSs. Its reference implementation contains different databases like Neo4j (4), DuckDB (Section 2.5), Memgraph (6), and Umbra (31), and allows us to run query benchmarks. Since it supports DuckDB and uses the same dataset as LDBC SNB (Section 2.6.1), it is useful for running microbenchmarks with the translated Kleene star query candidates.

3

Design Considerations

In this section, we provide details and rationale for scoping down the specification (Section 3.1). We then give an overview of the design choices that were considered for supporting SQL/PGQ in DuckDB (Section 3.2), focussing on the Kleene star translation and the reasoning behind choosing our approach. Finally, we present an overview of the DuckDB components affected by our approach (Section 3.2.3).

3.1 Simplification of SQL/PGQ Specification

As of 2022, the SQL/PGQ specification has not been finalised, and it contains some parser rules that make the implementation cumbersome such as supporting `PROPERTIES` clause, optional `KEY` clause, and abbreviated patterns which are elaborated below. We deemed that these rules are not required for having a minimal implementation. We also propose some minor changes for an easier implementation and expected performance improvements. The major proposed changes are:

- **Remove `PROPERTIES` keyword:** In the SQL/PGQ specification, properties can be explicitly defined to limit the attributes returned per edge or node. However, we implicitly assume all properties to be enumerated for a simpler design.
- **Explicit mandatory definition of `KEY` clause:** In the SQL/PGQ specification, the `KEY` clause is optional. It has to be inferred from the primary/foreign keys of the underlying table if the `KEY` is not used during creation of property graph. However, we have made the `KEY` field a mandatory argument that is defined explicitly for Vertex and Edge tables. This leads to a lower implementation overhead due to avoiding key inference.

- **Removal of `DROP` and `ALTER`:** Since property graphs are materialised views, support `DROP` and `ALTER` will lead to synchronisation issues with the underlying data making the implementation more complicated. To reduce the design overhead, property graphs will be newly materialised for each `MATCH` query, leading us to always use current data.

- **Filtering of tables only supported at the top level for pattern matching**: In the SQL/PGQ `MATCH` query, nodes and edges can have filtering clauses *e.g.* `MATCH (a:account{aid = 10})-[t:transfers]->(a2:account)`. However, we only support filtering using a `WHERE` clause for pattern matching as shown in Listing 2.2. Since this is the first version of the implementation, and there was a way to provide global filtering we deemed this to not be a core feature.

- **Unique labels per property graph**: We enforce a uniqueness for every label used per property graph across all vertex and edge tables. Without this constraint, we will have to do extra computation to figure out the underlying table being referenced by the label in the `MATCH` clause. For example, in Figure 2.3 if we allow customer and transfer tables to have the same label (customer), we would have to infer which label is being used in a `MATCH` query (in Listing 2.2)with account. We do this to simplify our implementation.

- **Removal of abbreviated edge patterns:**: The SQL/PGQ `MATCH` queries allow abbreviated edge patterns like `MATCH (a1:account)->(a2:account)` where the edge pattern `transfers` has to be inferred from the property graph definition. To reduce the implementation overhead for key inference, we mandate mentioning edge patterns in `MATCH` and Kleene star queries.

3.2 Supporting SQL/PGQ in DuckDB

In this section, we elaborate on the changes required for supporting SQL/PGQ in DuckDB. We want SQL/PGQ to be an extension in DuckDB (Research Question 1b) which is why we prefer lightweight changes over more obtrusive ones.

3.2.1 Parser Rules

New parsing rules have to be added in DuckDB's parser to incorporate the statements for creation of property graphs (Listing 2.1), pattern matching (Listing 2.2) and Kleene

star (Listing 2.3). Currently, DuckDB does not offer mechanisms to extend its parser nor are there any plans to do so due to the difficulty of an extensible grammar. Therefore, a SQL/PGQ implementation must change the grammar rules in the parser.

3.2.2 Adding Support for the Operators

```
1  -- SQL/PGQ pattern match query
2  SELECT a1id, a2id
3  FROM GRAPH_TABLE (aml,
4  MATCH (a1 IS account)-[t1 IS transfers]->(a2 IS account)-[t2 IS transfers
       ]->(a1)
5  COLUMNS (a1.aid AS a1id, a2.aid AS a2id)) gt
6
7  -- SQL query
8  SELECT a1id, a2id
9  FROM
10   (SELECT a1.aid as a1id, a2.aid as a2id
11   FROM account a1
12   JOIN transfers t1 ON a1.aid = t1.src_aid
13   JOIN account a2 ON a2.aid = t1.dst_aid
14   JOIN transfers t2 ON a2.aid = t2.src_aid
15   AND t2.dst_aid = a1.aid
16   ) gt
```

Listing 3.1: Pattern matching query in SQL/PGQ translated into SQL.

The `CREATE PROPERTY GRAPH` statement (Listing 2.1) is similar to creating a `CREATE VIEW` statement in DuckDB and can follow a similar execution pipeline without introducing a new physical operator. The pattern matching syntax without Kleene star (simple pattern matching) is syntax sugar for equijoins (as shown in Listing 3.1) which can be leveraged for mapping it to the DuckDB execution pipeline. However, adding support for the Kleene star operator requires specialised operators to yield efficient execution. It can be mapped to `RECURSIVE CTE` but that would likely be inefficient as compared to using a specialised reachability graph algorithm. Furthermore, we will have to create and maintain a graph representation like CSR to run graph algorithms efficiently. There are two major approaches that can be used for its integration that are described below:

3.2.2.1 Approach 1: Adding a new physical operator

To integrate the Kleene star, we can add a new logical and physical operator in DuckDB that follows its own execution pipeline. Kleene star would be treated like an operator like `SCAN` and `JOIN`. This gives us the advantage of having full control over the execution of

the operator, and the lifecycle of the CSR representation, and we can optimise the code specifically for our use case. However, it requires a major change in the execution pipeline of DuckDB as the introduction of a new operator would require changes to the Optimiser, Logical and Physical Execution part of the DuckDB pipeline (Figure 2.6). We would have to implement parallelisation for the new operator. For parallel MSBFS, Kaufmann *et al.* (32) describe a within-search parallelism approach that is at odds with the DuckDB execution model where every search is executed by a single core, and parallelism is achieved by giving different chunks of data to different cores.

Furthermore, as the DuckDB codebase is constantly changing with new features (*e.g.* DuckDB execution changed from pull based to push-based execution during the duration of this thesis), such heavy changes would create maintainability issues. Adding such a major change also goes against our design goals of architecting a minimal extension. As DuckDB is an open source system, it can be forked to create a new system where the changes of the parser are implemented and any changes to the execution pipeline can be made without a dependency on the current design goals of the DuckDB developers. However, this approach will require us to maintain this separate repository and incur a huge technical debt. Lastly, this will require us to popularise a new database or our fork which is an uphill task and we lose out on the existing popularity of DuckDB.

3.2.2.2 Approach 2: Using DuckDB's User Defined Functions

Another approach is to leverage UDFs in DuckDB for the creation of CSR and reachability, and then transpile/remap the shortest path query to a `SELECT` query. This mapping changes the query in the Binder stage, and can then leverage the existing execution pipeline without affecting the rest of the pipeline (Figure 2.6). This approach also is parallelisable by default as UDFs are parallel out of the box. A major advantage of this approach is maintainability and ensuring minimal code changes.

Before running the reachability algorithm, a graph representation of the source and edge tables has to be created for efficiently running graph algorithms. The CSR is kept in memory using two large arrays in the DuckDB's client context and is created for every new Kleene star query. This keeps the data in sync because if we do not rebuild CSR for every query, we would end up with stale snapshots in case data from the original tables has been modified. However, we have to ensure that the CSR creation process is fast and has minimal overhead; otherwise this approach might be too slow.

The major disadvantage of this approach is that we are using scalar UDFs for allocating memory using malloc (which goes against the design goals of UDFs), while majority of

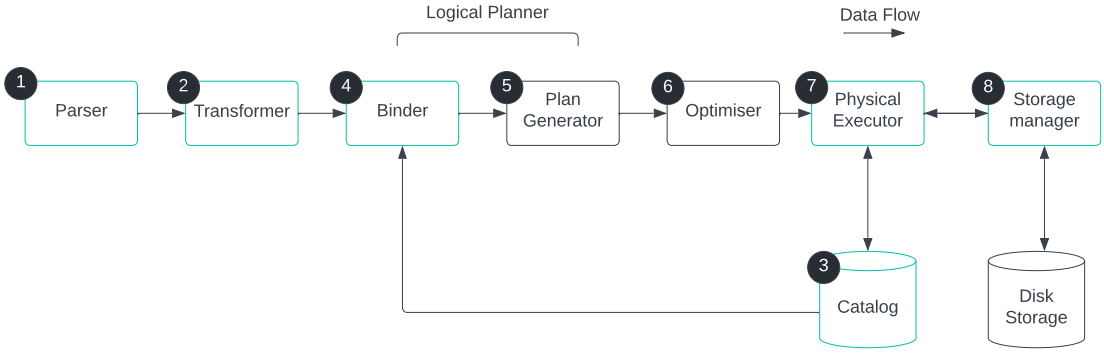


Figure 3.1: DuckDB components affected by the SQL/PGQ support.

the UDFs are used for operations like text search, aggregation and arithmetic operators. Another thing to note is that the arrays used for CSR consume a large amount of memory which is not expected from DuckDB. Therefore, we should keep memory guards and throw an exception if memory cannot be allocated to avoid bringing down the entire application with an "Out of Memory" error. Furthermore, while DuckDB's UDFs have out-of-the-box parallelism, this is only advantageous if we find paths for multiple source-destination pairs.

However, due to the minimal changes and ease of convenience/pluggability in DuckDB, we decided to go with the UDF approach.

3.2.3 DuckDB Components Affected by the Code Change

In Figure 3.1, we highlight the DuckDB components requiring a change to support SQL/PGQ operators in DuckDB with the selected UDF approach.

- The parser ❶ is affected as we parse the SQL/PGQ queries for `CREATE PROPERTY GRAPH` and `MATCH` queries as explained in Section 3.2.1.
- The transformer ❷ is utilised to convert the trees to custom C++ objects used by the rest of the pipeline. This requires changes to handle the tree nodes for the introduced grammar rules. Hence the parser changes necessitate a change in the transformer.
- Binder ❹ is utilised in a `CREATE PROPERTY GRAPH` statement to resolve the table names. For `MATCH` and the Kleene star queries, it is used to transpile the query to a `SELECT` query. This is explained in more detail in Section 4.
- Simple `MATCH` and the Kleene star queries leverage the existing `SELECT` query execution, hence they avoid any other changes in the rest of the pipeline while `CREATE`

`PROPERTY GRAPH` requires changes to Catalog ❸ to store the Property Graph metadata, Physical Executor ❼ to execute the query and some changes to Storage Manager ❽ for logging and checkpointing.

4

Implementation

This section presents important implementation details for adding SQL/PGQ support in DuckDB. We first explain the implementation of the `MATCH` clause without the Kleene star operator in Section 4.1. We then explain the creation of CSR in DuckDB using UDFs in Section 4.2. Finally, we explain the reachability UDF used for the Kleene star operator in Section 4.3.

4.1 Implementing `MATCH` Clause Without the Kleene Star Operator

For implementation of the `MATCH` clause without the Kleene star operator, we had described its translation to equijoins in Listing 3.1. Furthermore, the equijoins query is equivalent to a SQL cross product query where the `JOIN` conditions have been added to the `WHERE` clause (Listing 4.1). We chose the option of using the cross product query for simple `MATCH` query translation as it was simpler to implement in the Binder (Figure 3.1 ❹).

The pseudocode along with the comments for the translation of SQL/PGQ `MATCH` query without the Kleene star operator is shown in Listing 4.2. We iterate over the `MATCH` clause list (Listing 3.1 line 12) to: i) create the expressions for the `WHERE` clause ii) populate the *alias_table_map* used to create the cross product table reference for the `FROM` clause. Based on the edge direction, we create the source and the destination expressions as shown in Lines 25-28 of Listing 4.2. The columns for the equality condition (used in source and destination expressions) are selected using the key of the vertex and edge tables (defined at the creation of property graphs as shown in Listing 2.1). We have shown the snippet for the LEFT edge direction in Listing 4.2 (Lines 23-31). For the RIGHT edge direction, the source and destination expressions will be switched. For ANY direction (bi-direction),

we create a conjunction of disjunction of both LEFT and RIGHT edge direction expressions. For example, if left_src_expr, left_dst_expr are the source-destination expressions for LEFT direction, and right_src_expr, right_dst_expr are source-destination expressions for the RIGHT direction, then ANY direction expression will be: (left_src_expr AND left_dst_expr) OR (right_src_expr AND right_dst_expr). We use the populated *alias_table_map* to create the FROM clause and move the disjunction of source-destination expressions to the WHERE clause of the SELECT statement (Lines 35, 36). We finally wrap the SELECT statement under a SubQuery (Lines 37,38). Using this approach, our statement then follows the execution pipeline of a SELECT statement in DuckDB (Figure 3.1 ❹-❽).

```
1  SELECT a1id, a2id
2  FROM
3    (SELECT a1.aid as a1id, a2.aid as a2id
4    FROM account a1, transfers t1, account a2, transfers t2
5    WHERE a1.aid = t1.src_aid
6      AND a2.aid = t1.dst_aid
7      AND a2.aid = t2.src_aid
8      AND t2.dst_aid = a1.aid
9    ) gt
```

Listing 4.1: Pattern matching using cross products.

```
1  Input: match_list, where_expr, select_list
2
3  //conditions to store the conditions for WHERE clause, alias map to store
       the table and alias names used for the FROM clause.
4  conditions, alias_table_map
5
6  //match_list cannot be empty.
7  prev_vertex = match_list[0]
8  // property graph entry in the catalog during the CREATE PROPERTY GRAPH.
9  prev_vertex_entry = GetEntryFromLabel(prev_vertex->label)
10 alias_table_map[prev_vertex->alias] = prev_vertex_entry->name
11
12 for (i = 1; i < match_list.size(); i+=2) {
13   //edge pattern will always be between 2 vertex patterns.
14   edge = match_list[i]
15   vertex = match_list[i+1]
16
17   edge_entry = GetEntryFromLabel(vertex->label)
18   vertex_entry = GetEntryFromLabel(edge->label)
19   //alias to be used in the FROM query.
20   alias_table_map[vertex->alias] = vertex_entry->name
21   alias_table_map[edge->alias] = edge_entry->name
22   switch(vertex->direction) {
```

```
23      case LEFT:
24        //keys defined in CREATE PROPERTY GRAPH used for equality columns.
25        src_expr = CreateExpression(vertex_entry->key, edge_entry->source_key
              , vertex->alias,
26        edge->alias)
27        dst_expr = CreateExpression(prev_vertex_entry->key, edge_entry->
              destination_key,
28            prev_vertex->alias, edge->alias)
29        conditions.append(src_expr)
30        conditions.append(dst_expr)
31        prev_vertex_entry = vertex_entry
32    }
33  }
34  //create the cross product reference using the alias_map
35  select->from_list = CrossProductRef(alias_map)
36  select->where_expr = conditions.append(where_expr)
37  subquery->node = select
38  Bind(subquery)
```

Listing 4.2: Code snippet showing `MATCH` query translation to a `SELECT` statement in DuckDB Binder.

4.2 Creating CSR

For supporting the shortest path query in SQL, we create a Compressed Sparse Row (CSR) representation of the graph using two large arrays, namely csr_v and csr_e, for storing nodes and edges. The CSR is compact and limits the number of random accesses required as the identifiers of a given node's neighbour nodes are stored together allowing fast scan for large outdegree nodes in order as shown in Section 2.4. We use two UDFs to create the CSR and two helper functions to initialise these functions. The functions have an extra parameter called *index_id* which refers to the id of the Kleene star instance if there are multiple instances of the operator in a single `MATCH` query. However, without loss of generality for explaining the functions and the queries, we assume that we only have a single Kleene star referred to by the index 0 in our translated SQL queries. We describe the functions and their parameters below.

- *create_csr_v(size, dense_id, cnt)*: It is used to create the first (vertex) CSR array and returns the total edge count.
  - size: the total size to be allocated for csr_v.
  - dense_id: the new identifier that vertex will be mapped to, in the range of $0, \ldots, |V| - 1$.

  - cnt: the number of outgoing edges (outgoing degree) from the vertex.

- *csr_init_vertex(size)*: It is used to initialise and allocate space for the csr_v array. This function uses a lock for the csr_v allocation by the first thread (as DuckDB is multi-threaded). We allocate two extra 0s in the array for creation of running sum and which will be explained in the Section 4.2.1. Afterwards, a flag is set to mark the array as initialised and any other thread will not call init to avoid reinitialisation and locks.

- *create_csr_e(v_size, e_size, src_rowid, dst_rowid)*: It is used to create the second (edge) array of the CSR. It returns a place-holder constant value (1).

  - v_size: total number of vertices in the graph.
  - e_size: total number of outgoing edges in the graph.
  - src_rowid: `rowid` of the row representing the source node of the edge in the vertex table.
  - dst_rowid: `rowid` of the row representing the destination node of the edge in the vertex table..

- *csr_init_edge(v_size, e_size)*: Allocates space for csr_e array and calls the running sum for the csr_v array. The flag and locking mechanism are similar to the csr_init_vertex function.

  - v_size: total number of vertices in the graph.
  - e_size: total number of outgoing edges in the graph.

```
SELECT sum(CREATE_CSR_VERTEX(0, (SELECT max(a.rowid) + 1 FROM account a),
sub.dense_id , sub.cnt )) AS numEdges
FROM (
  SELECT a.rowid as dense_id, count(t.src_aid) as cnt
  FROM Account a
  LEFT JOIN  Transfers t ON t.src_aid = a.aid
  GROUP BY c.rowid
)
```

Listing 4.3: SQL to create CSR Vertex array

```
SELECT CREATE_CSR_EDGE(0, (SELECT max(a.rowid) + 1 FROM Account a),
<<invoke sum(create_csr_vertex)>> ,
src.rowid, dst.rowid )
FROM
```

```
5   Transfers t
6   JOIN Account src ON t.src_aid = src.aid
7   JOIN Account dst ON t.dst_aid = dst.aid
```

Listing 4.4: SQL to create CSR Edge

The two CSR creation functions have atomic increment operations and the `csr_v` array uses atomic integers to avoid locking besides the first init functions. This is because we want CSR creation to be a fast operation and have minimal overhead to ensure that CSR (re)creation is not a major bottleneck. The SQL query used for the creation of `csr_v` array is shown in Listing 4.3 and for creation of `csr_e` array is shown in Listing 4.4. *create_csr_vertex* is a parameter to the *create_csr_edge* function which, by introducing a dependency ensures `csr_v` array is created first. In Listing 4.4, we only add the placeholder for invocation of *create_csr_vertex* function without the parameters to avoid code repetition, for a compilable query one can copy the Listing 4.3 to replace the placeholder.

SF	Rank (s)	Dense rank (s)	`rowid` (s)
0.1	0.0068	0.0079	0.0036
0.3	0.0203	0.0145	0.0110
1	0.0336	0.0292	0.0254
3	0.0669	0.0767	0.0471

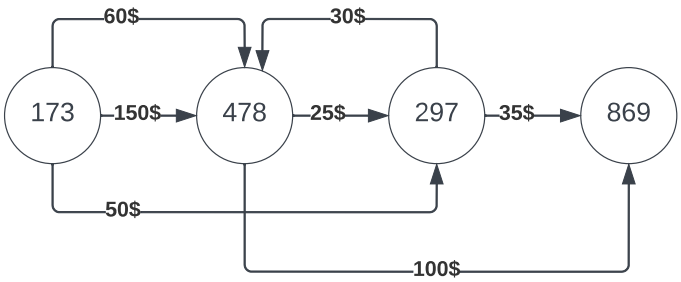
Table 4.1: Comparison of running time in seconds for generation of dense identifiers using window functions (rank, dense_rank) and `rowid`.

We leverage DuckDB rowids to get the dense ids for our CSR. The other option was to use window functions (rank, dense_rank) [1] over the vertex table and apply sorting. To compare the performance of these two approaches, we ran a LSQB microbenchmark (Section 2.6.2) on DuckDB. The approach of using rowids was selected as it is faster as shown in Table 4.1. Furthermore, rowids are a common feature across RDBMSs, therefore this approach can be used in other RDBMSs too.

```
1 def create_csr_edge(src, dst):
2   pos = ++csr_v[src + 1];
3   csr_e[pos - 1] = dst;
```

Listing 4.5: Code snippet for *create_csr_edge* for creating the `csr_v` and `csr_e` arrays using atomic increment.

[1] `https://duckdb.org/docs/sql/window_functions`



vertex	rowid
173	0
478	1
297	2
869	3

Figure 4.1: Sample Graph containing the transfer information from different account ids. Sparse vertices represent the account ids and edges represent the transfer amounts. Vertices are relabelled using dense rowids of DuckDB.

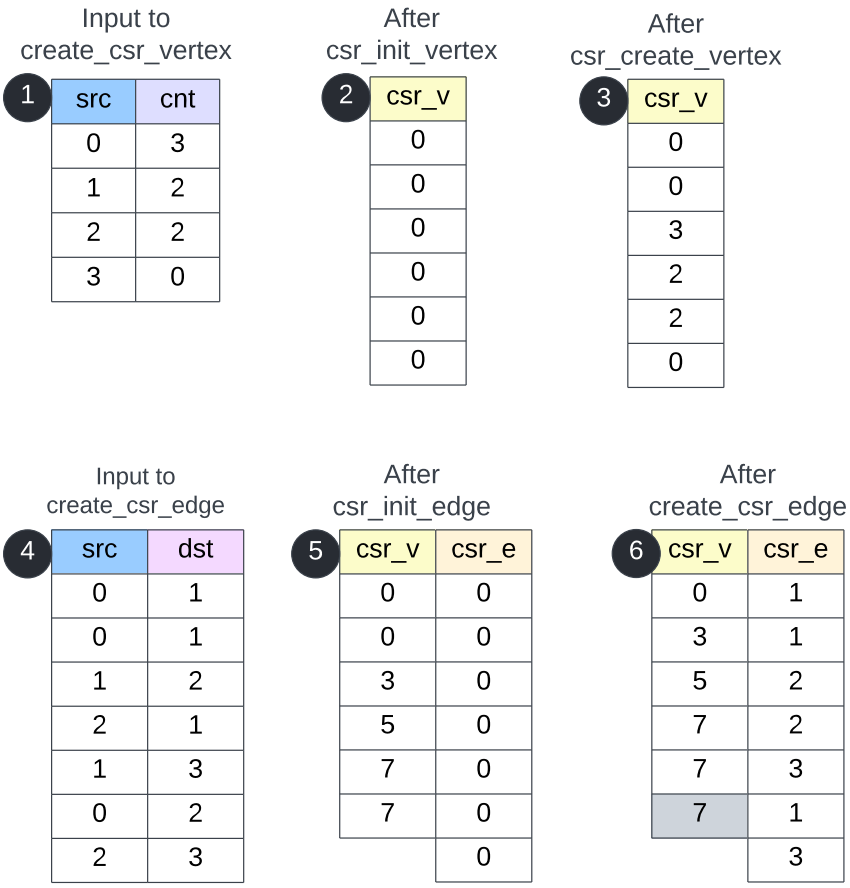


Figure 4.2: CSR creation pipeline for the graph in Figure 4.1. The grey box represents an extra element not used in the final CSR.

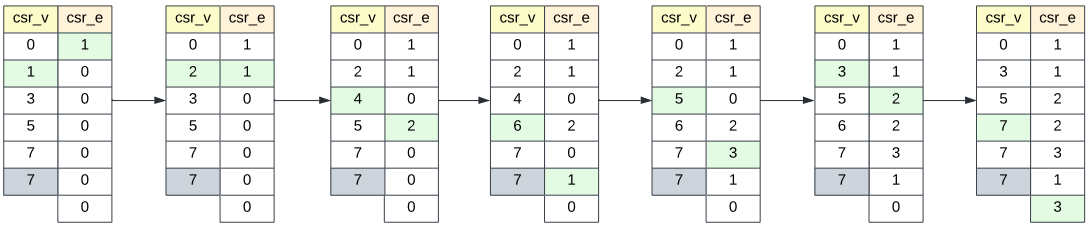


Figure 4.3: Intermediate states of arrays during create_csr_edge. The green boxes show the change for each source-destination pair. The grey box represents an extra element which is not used during the process.

4.2.1 CSR Creation Pipeline

As we leverage the rowids of DuckDB, the original vertex ids (primarily the primary key of the vertex table, e.g. an arbitrary int64 value for each row) are relabelled to dense ids as shown in Figure 4.1. The steps of CSR execution are:

- As shown in Figure 4.2, *create_csr_v* gets the relabelled source vertex and outgoing edge counts as input ❶.
- `csr_v` is allocated the size: |V| + 2, using malloc in *csr_init_vertex* ❷.
- In *create_csr_vertex*, the `csr_v` array is created with the two dummy 0s (for padding), followed by the outgoing edge counts of each vertex ❸.
- The relabelled edge table (source and destination vertices are replaced with their rowids) is given as input to *create_csr_edge*. Note that source and destination pairs are not ordered by any column(s) ❹.
- During *csr_init_edge*, space is allocated for `csr_e` array equivalent to total edge counts in the graph and it creates a running sum on `csr_v` ❺.
- Finally, in *create_csr_edge* for every source-destination pair, the source index in `csr_v` is used to determine the index of destination in `csr_e` array and it is then pre-incremented (atomic increment) by 1 as shown in Listing 4.5. In effect, this moves the running sum up an index in the `csr_v` array and assigns the corresponding neighbours in the `csr_e` array ❻. The CSR for the graph in Figure 4.1 is shown in Figure 4.2 ❻.

The code snippet for the *create_ csr_ edge* is shown in Listing 4.5 and the states of arrays during execution are shown in Figure 4.3 with the changing values highlighted at each step. Creating the CSR in such a way allows multiple threads to concurrently access the CSR arrays even for out of order source-destination pairs.

4.3 Reachability UDF

For mapping Kleene star into SQL operators, we need a reachability UDF that takes as input a source-destination pair of vertices and returns $True$ if the destination is reachable from source and $False$ otherwise. We implement reachability using the Then *et al.* MSBFS algorithm (25) (Section 2.3) as it is batched and suitable for many source - many destination pairs. Furthermore, since the outcome of pairs does not depend on an order they can be executed by multiple threads and performance should also improve with DuckDB's vectorised execution (1024 tuples being processed at the same time). However, before running the reachability algorithm, we first need to create the CSR so that we have a graph representation in memory. We utilise the execution order in Common Table Expressions (CTEs) for this where *create_ csr_ vertex* and *create_ csr_ edge* functions are called in the CTE while reachability is put in the `WHERE` clause of the outer `SELECT` query. Another approach to ensure ordered execution was to use `LATERAL JOINS`, however they are not supported in DuckDB.

```
1  SELECT gt.a1id, gt.a2id
2  FROM GRAPH_TABLE (aml, MATCH (a1 IS account)-[t1 IS TRANSFERS*]->(a2 IS
       account) COLUMNS (a1.aid AS a1id, a2.aid AS a2id)) gt;
3
4  //Translation of query with Kleene star
5  WITH cte AS (
6  SELECT min(CREATE_CSR_EDGE(0, (SELECT max(a.rowid) + 1 FROM account a),
7  CAST (SELECT sum(CREATE_CSR_VERTEX(0, (SELECT max(a.rowid) + 1 FROM account
       a),
8  sub.dense_id , sub.cnt )) as numEdges
9  FROM (
10   SELECT a.rowid as dense_id, count(t.src_aid) as cnt
11   FROM Account a
12   LEFT JOIN  Transfers t ON t.src_aid = a.aid
13   GROUP BY c.rowid
14 ) sub) AS BIGINT),
15 src.rowid, dst.rowid )) as csr, (SELECT max(a.rowid) + 1 FROM account a) as
       vcount
16 FROM
17   Transfers t
```

```
18    JOIN Account src ON t.src_aid = src.aid
19    JOIN Account dst ON t.dst_aid = dst.aid
20  )
21  SELECT src.aid AS a1id, dst.aid AS a2id
22  FROM Account src, Account dst
23  WHERE
24  ( REACHABILITY(0, false, (SELECT cte.vcount FROM cte1), src.rowid, dst.
        rowid) = (SELECT cte.csr FROM cte));
25  //cte.csr = 1
```

Listing 4.6: Translation of match query with a Kleene star operator

The translation of a Kleene star query to a `SELECT` query with the three UDFs is shown in Listing 4.6. The CTE is called in an inner query to force DuckDB logical planner to use a hash join (33) and avoid using a blockwise nested loop join (33) as it leads to scalability issues. The *create_csr_edge* function returns a constant (1) which is used as the equality condition.

The entire translation of the query is done in the Binder (Figure 3.1 ❹) of DuckDB as it is the first place in the execution pipeline with access to the Catalog. This is done to ensure that labels used in the query actually refer to an underlying table. The label existence is done using a map stored in the Catalog (between labels and property graphs) defined during the `CREATE PROPERTY GRAPH` statement. Furthermore, the vertex and edge labels can have empty identifiers, *e.g.* `(IS account)-[IS transfers]->(a2 IS account)` in which a temporary identifier is assigned to transfers and the first account label, as it will be used to alias to that table in the translated SQL query. If we would only use the table names then in the translated `JOIN` query, we will not be able to differentiate if we are referring to the first account table (source) or the second (destination).

4.3.1 Sub-Optimisation for MSBFS

Due to DuckDB's vectorised execution model, 1024 source-destination pairs are processed at the same time. There is a possibility that some sources are being repeated in 1024 pairs, so it provides an opportunity for a suboptimisation to reserve 1 MSBFS lane for every unique source as MSBFS computes the reachability of all nodes from the source nodes. The occurrences of the repeated sources are saved in a map to avoid extra computation (re-running the same BFS), however we require extra memory to store this mapping. Furthermore, the MSBFS algorithm according to the experiments in the original paper (25) works best with bitsets as wide as the CPU cache line width (which is generally 128-512 bits in most modern computers). If there are multiple cases of repeated source-destination

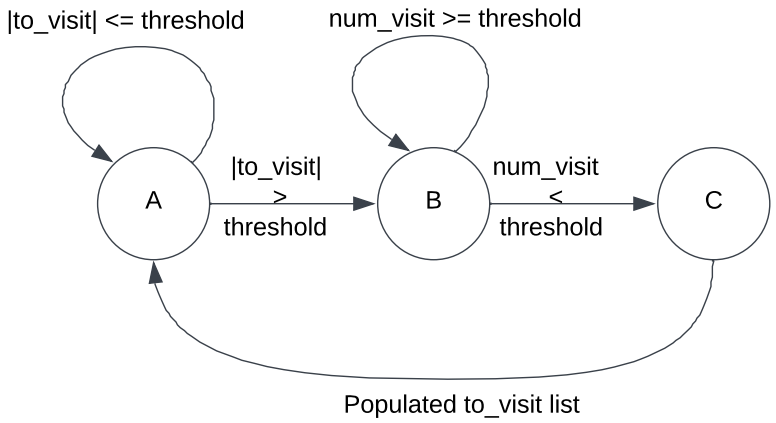


Figure 4.4: States for the dynamic MSBFS.

pairs, we have the opportunity of saving the entire seen bitset even for thousands of source-destination pairs on the cache line for faster execution. We try to vary the lane sizes and see the effect of this suboptimisation through scalability experiments we run in Section 5.4.2 and Section 5.4.1.

4.3.2 Dynamic MSBFS

A dynamic version of the MSBFS can be tried with multiple states where we use and populate a list of vertices (*to_visit*) to be explored in the next iteration MSBFS in some states. Instead of running the `for` loop for all vertices in the graph (line 21 in Listing 2.3), we use the *to_visit* array. In the initialisation, *to_visit* only contains the source vertices, and to transition among states we use a threshold size $|V|/i$ (i is a variable used for our experiments with values 2, 3 and 4): The states are shown in Figure 4.4 and we describe them in detail below:

- State A: This is the starting state with a populated *to_visit* list that we use instead of iterating all nodes in the graph. In the second for loop (line 27 in Listing 2.3), we populate the *to_visit* list. At the end of the iteration, we check the size of *to_visit* list and if it is less than the threshold, then we remain in state A, otherwise we transition to state B.

- State B: We revert to the original algorithm and iterate all vertices in the graph. In the second `for` loop, we do not populate a *to_visit* list; instead we keep a size variable (*num_visit*) to keep track of the number of elements to be visited. If the

size (*num_visit*) is greater than threshold then we remain in state B, otherwise we transition to state C.

- State C: This is an intermediate state. We know the number of elements to be visited is lower than the threshold, however we do not have the *to_visit* list. Therefore, we iterate over all the vertices but we populate the *to_visit* list which will be used in the next iteration. We transition back to state A from this state.

The idea of the dynamic approach is to reduce the number of iterations in the graph. We keep a threshold based on the $|V|/i$ as we want to see if keeping a list of 25% or 50% of the elements might amount to more work than iterating the elements in sequential order. The experimental results of this dynamic MSBFS are shown in Section 5.4.4.

4.3.3 Lack of Support for Filtering using UDFs

Filtering is not supported for reachability queries (using the Kleene star operator) as it would require changes to the CSR for filtering out vertices and edges, and would increase code complexity. As this was a minimal extension, we deemed pushing this support to a subsequent version.

5

Evaluation

To assess the scalability of the proposed approach, we conducted a series of experiments. In this section, we discuss the performance experiments and their results. We first explain the experimental setup (Section 5.1) and the query selection. We then present and discuss the results of different scalability experiments (Section 5.3 and Section 5.4).

5.1 Experimental Setup

A machine in the internal CWI SciLens (34) cluster was used to run the experiments. Furthermore, a sufficiently large RAM allowed us to keep the CSR in memory for larger scale factors and efficiently run MSBFS, including its multi-threaded variant which was not possible on a commodity machine. The configuration of the machine is described in Table 5.1.

Operating System	Linux
OS Distribution	Fedora 28
Kernel Version	4.18.7
CPU	Intel Xeon
RAM	128GB
Threads	16
Disk Size	1.5TB
Disk Type	SSD

Table 5.1: Configuration of the jewels05 machine in CWI's SciLens cluster.

5.2 Query Selection

From the SNB Interactive workloads (Section 2.6.1), we choose query 13 for our scalability experiments. The query aims to find the length of the shortest path between two Person nodes along Knows edges over any number of intermediate hops as it uses the Kleene star operator as shown in Figure 5.1. The query has a large search space and captures a typical use of the Kleene star. In this query, Person is the vertex table and PersonKnowsPerson is the edge table.

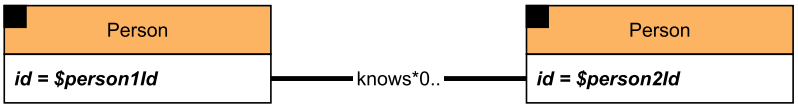


Figure 5.1: SNB Interactive Query 13 for a complex read.

5.3 Create CSR Scalability

CSR creation is run for the entire graphs up to Scale Factor 300. *create_csr_edge* takes the creation of *create_csr_vertex* as its parameter, therefore its execution time will always be greater than *create_csr_vertex*. For the smaller scale factors, the data size is too small where the overhead of using multiple threads actually hampers execution time for the smaller scale factors as seen in Figure 5.2. However, for larger scale factors CSR creation scales well with an increase in threads as seen in Figure 5.2, and is a relatively fast process (*e.g.* csr_edge creation for SF 100 takes 1.72 seconds with 8 threads). This is extremely important as the CSR is regenerated for every new `MATCH` query.

5.4 Reachability

Reachability is implemented as a DuckDB scalar UDF. In the experiment, the number of rows to be checked for reachability is fixed at 10000 as the cross product of 100 unique sources and 100 unique destinations from the Person table selected using md5 hashing (to select random vertices but with a reproducible order). Since the aim was primarily to observe scaling behaviour and reachability is a significantly complex and slower function than CSR creation, a smaller data size was enough to observe the scaling behaviour (in juxtaposition with the create_csr functions that required a larger data size to have a high enough execution time). The CSR has to be created before we run the reachability algorithm, so we add the dependency in the CTE as shown in Section 4.3. The fullquery

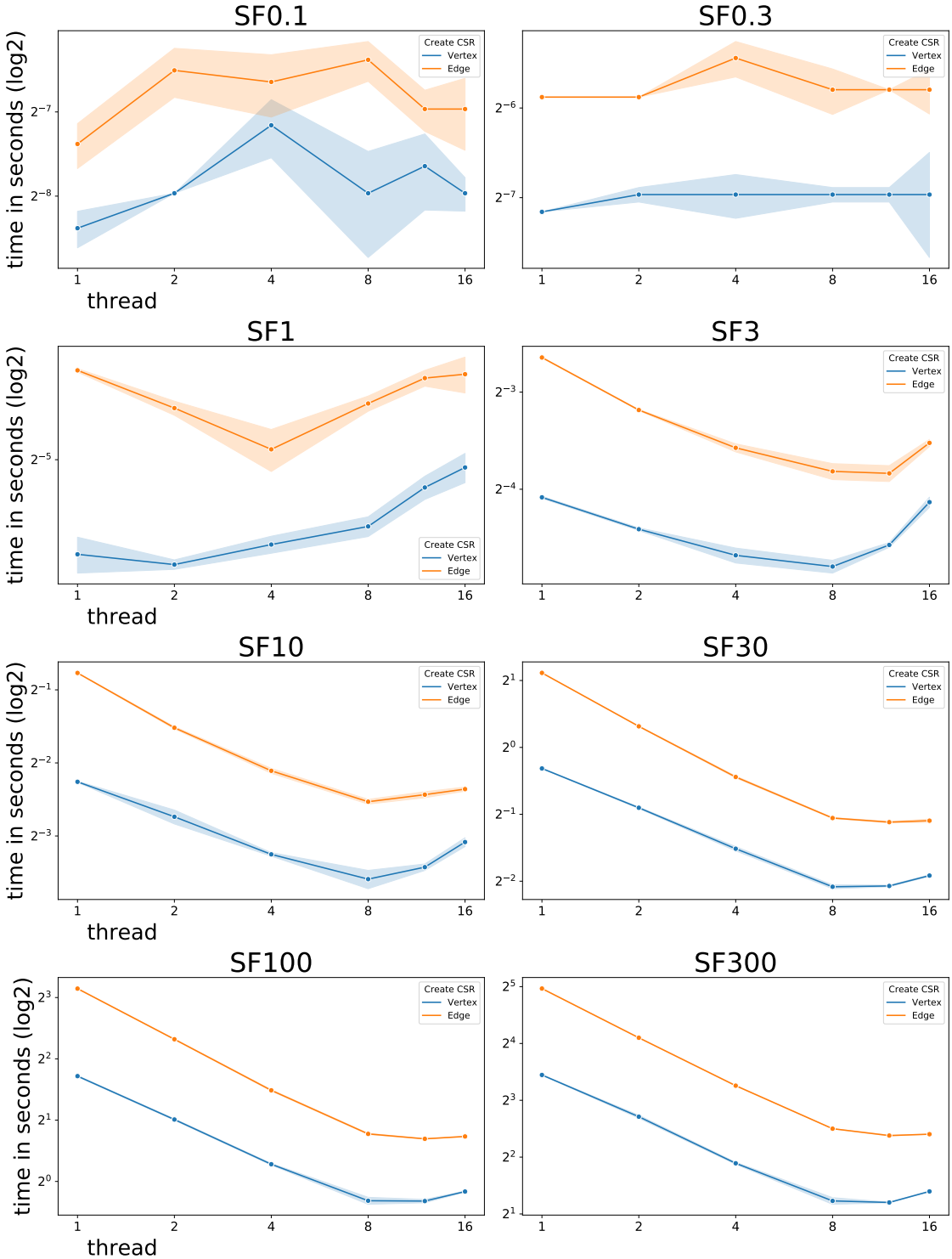


Figure 5.2: Running times of create CSR vertex and edge.

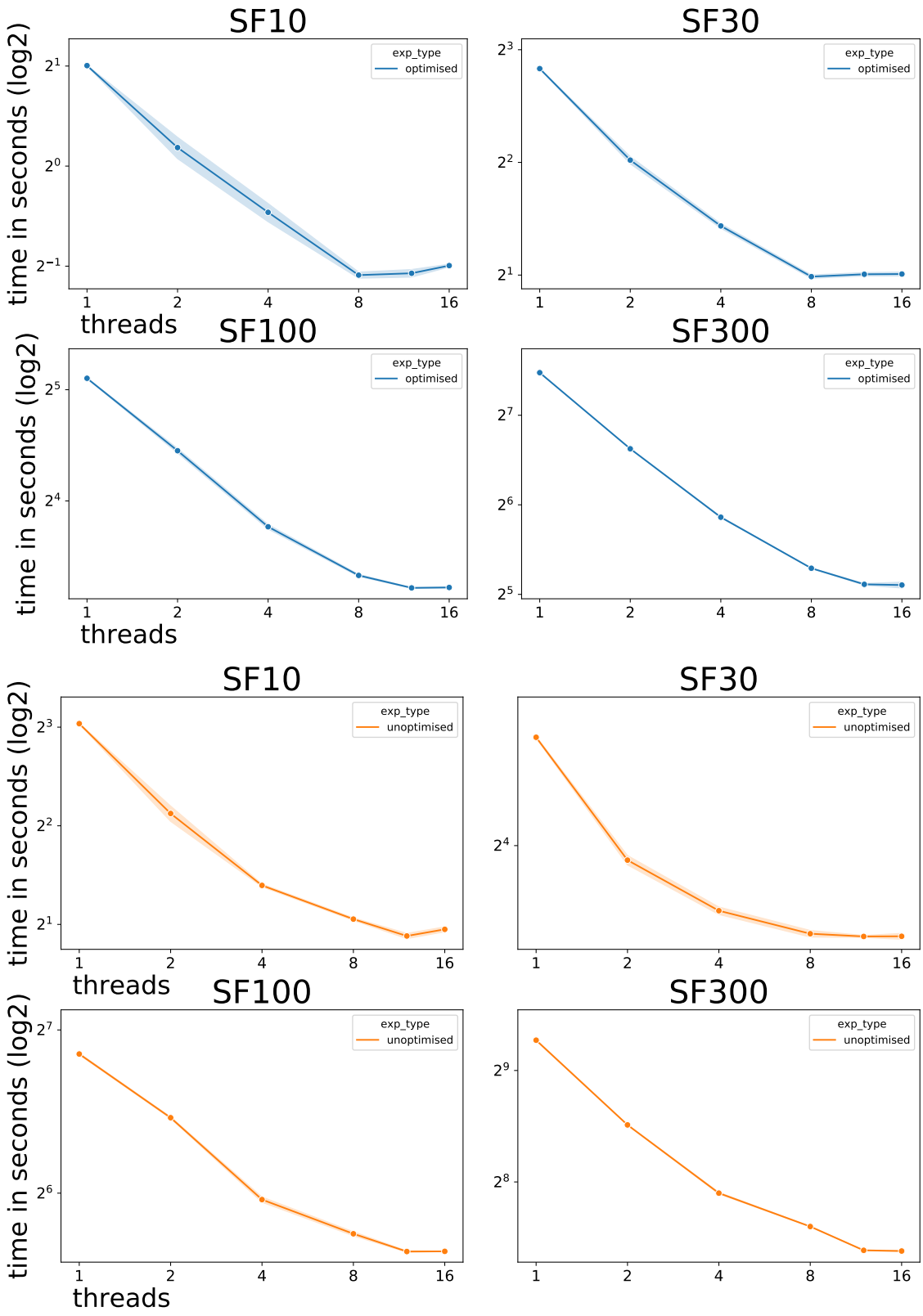


Figure 5.3: Running times of full query with the reachability function run for 10k source, destination pairs. The first four figures refer show the MSBFS with the suboptimisation (blue lineplots) and the last 4 without the sub-optimisation (orange lineplots).

is similar to Kleene star translated query as shown in Listing 4.6 with edge tables replaced with PersonKnowsPerson and vertex tables with Person.

5.4.1 Sub-Optimisation

A sub-optimisation in the implementation was introduced where a single MSBFS lane is utilised for a unique source vertex. The source vertices used are stored in a hashmap and at the end of the algorithm, the mapping is used to retrieve the position in the Vector and return whether the source-destination pairs are reachable. For example, running reachability with the sub-optimisation on 2 source-destination pairs (1, 4), (1,5) will lead to only 1 lane being utilised, instead of 2 lanes without the sub-optimisation. The results of running MSBFS with and without this optimisation are shown in Figure 5.3. We can observe the query runtimes without the optimisation are 2-3X slower than the running times with the optimisation. The experiments were conducted with the cross product of 100 unique source and 100 unique destination pairs. Therefore, on average, a source was repeated 10 times for a DuckDB vector of size 1024.

5.4.2 Lane Size

To figure out if increasing the lane size had an impact on execution times, we ran fullquery benchmark with various lane sizes (32, 64, 128, 256 and 512). With the smaller scale factors, it became clear that bigger lane sizes were faster as shown in Figure 5.4, therefore for larger scale factors (SF $\geq$ 10), benchmarks were only run for the three large lane sizes - 128, 256 and 512. In these experiments, 100k rows were checked for reachability with the cross product of 1000 unique source and 100 destination vertices selected using md5 hashing. The source unique vertices were increased by a factor of 10 because with the suboptimisation, 128 lane bits were enough to handle 100 unique vertices at every iteration of the UDF; hence 128 lane sizes ended up performing better than the bigger lane sizes in all cases.

As shown in Figure 5.4, lane sizes 128 and 256 perform better than lane size 512 in some cases for SF 10 (threads 8, 12) and 30 (threads 1, 2). However, at SF 100 and 300, lane size 512 outperforms the smaller lanes (128, 256) for all cases. We think this is because for scale factors 10 and 30 using the huge lane size is pushing data out of cache lines which worsens our performance. However, the computation advantage that we get with the bigger lane size of running more BFSs concurrently negates this loss of performance.

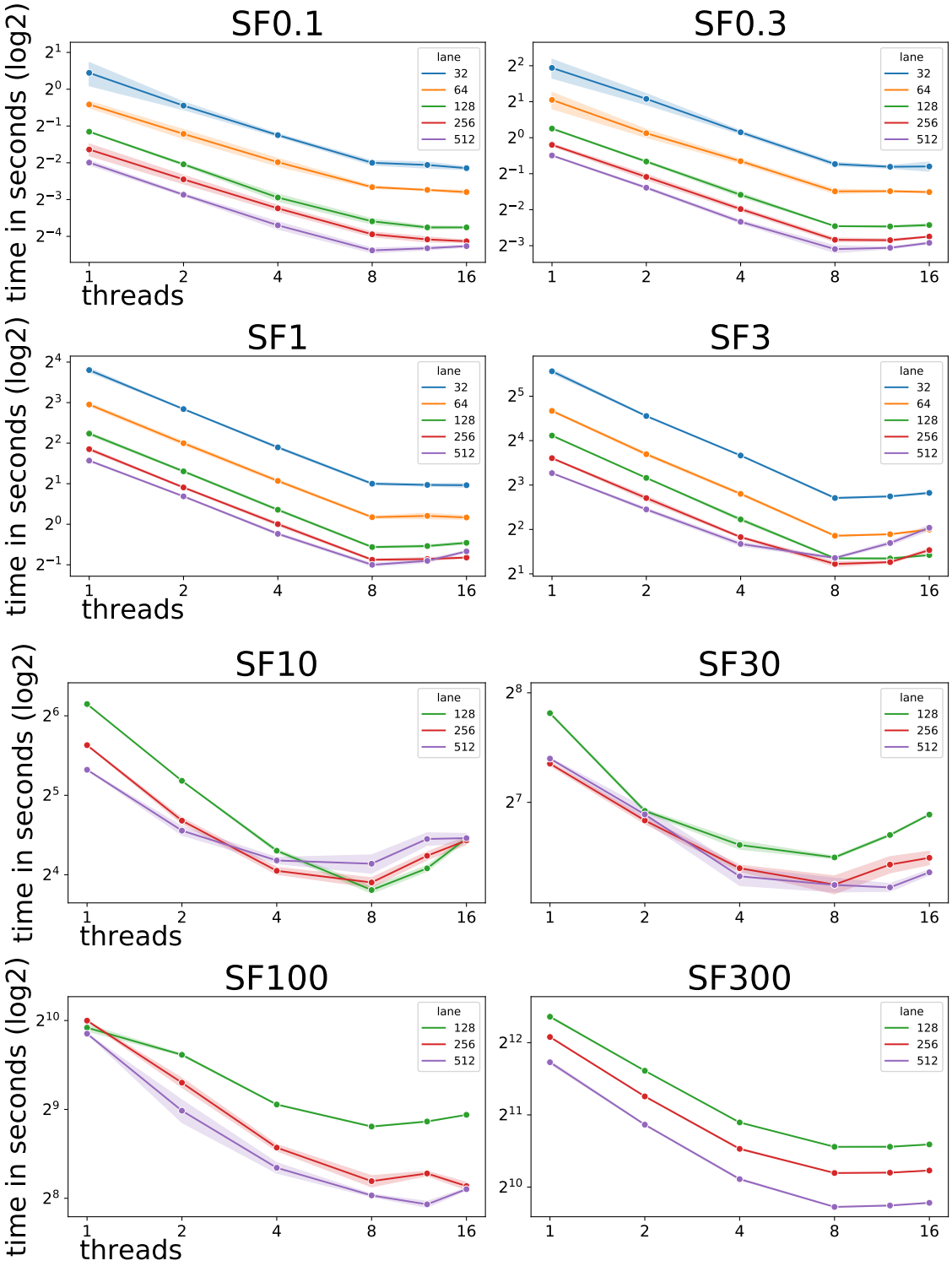


Figure 5.4: Running times of fullquery with different lane sizes.

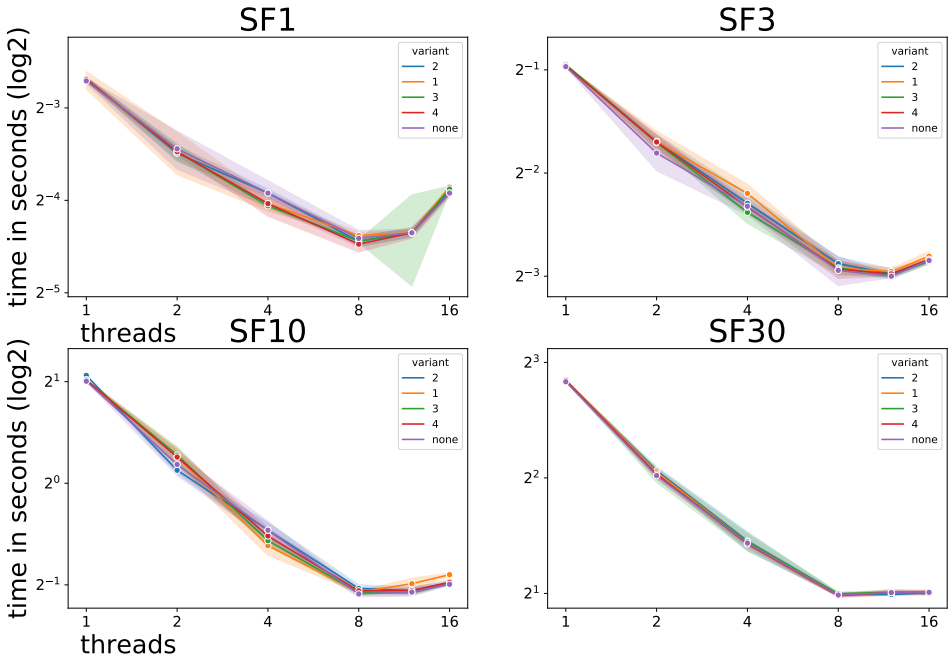


Figure 5.5: Running times in seconds for dynamic MSBFS. None variant refers to the original MSBFS.

Therefore, for future work we could explore dynamically changing the lane size depending on the underlying data statistics/size.

5.4.3 AVX512

Observing the trend of a bigger lane size improving performance (in Section 5.4.2), we tried compiling the code in Clang[1] (version 14.0.0) on an AVX-512 [2] enabled machine, and using AVX512 hints to SIMD-ise the code. However, the auto compiler failed to improve results with the compiler indicating that the loops were not vectorisable. Therefore in future work, adapting the code by explicitly writing SIMD statements could be looked into for improving performance.

5.4.4 Dynamic MSBFS

The dynamic MSBFS defined in Section 4.3.2 gave no notable performance improvement as compared to the original algorithm as shown in Figure 5.5. In the figure, the variant

[1] https://clang.llvm.org/
[2] https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html

refers to the i where the threshold is defined as $|V|/i$, and none variant refers to the original algorithm. Overall the sequential access of nodes in the original algorithm compensates for iterating more elements in the graph and performs as good or better than the dynamic version.

6

Related Work

In this section, we describe the related work and existing research in the field of graph queries in RDBMSs, GDBMSs and graph query languages. We first describe the surveys related to graph queries and graph systems (Section 6.1). We then introduce some popular graph query languages along their query syntax (Section 6.2) to show how they have evolved. We then discuss the approaches taken by different RDBMSs in supporting graph queries (Section 6.3) and compare them with our approach. Finally, we end with the literature on reachability and path-finding algorithms (Section 6.4).

6.1 Surveys

6.1.1 Foundations of Modern Query Languages for Graph Databases

Angles *et al.* (10) published a survey categorising the basic features and evaluation semantics of graph query languages, focusing on SPARQL (35), Cypher (Section 6.2.1) and Gremlin (Section 6.2.2). They identify pattern matching and navigational expressions as the major use case of query languages. The authors sub-categorise the evaluation seman-

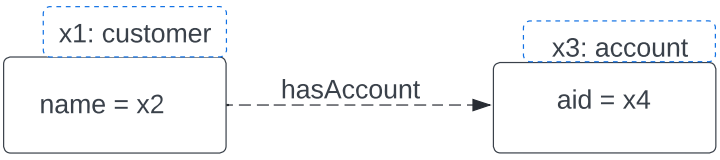


Figure 6.1: Basic graph pattern for Figure 2.3.

tics for pattern matching and navgiational expressions, and elaborate on their usage in the 3 languages. The authors categorise graph patterns into:

- *Basic graph patterns (bgps)* follow the same structure as the type of graph database they are intended to query but instead of only allowing constants, basic graph patterns also permit variables. An example of a bgp for the property graph of Figure 2.3 is shown in Figure 6.1.
- *Complex graph patterns* are basic graph patterns extended with operations like projection, union, difference, optional (aka left-outer-join), and filter.

A match of a graph pattern is finding a mapping between the graph pattern variables to the constants/exact values in the original graph while ensuring the structure is preserved (edges, vertices, labels in a graph pattern map to edges, vertices and labels in the underlying graph). The matching can be done under various semantics:

- *Homomorphism based semantics*: This is the unconstrained semantics where multiple variables can map to the same term in the graph (*e.g.* multiple nodes in the graph pattern can map to the same node in the underlying graph). It is used in SPARQL and Gremlin.
- *Isomorphism based semantics*: This semantic has more constraints on the structure of the mappings. It is further sub-divided into:
  - *No-repeated-anything semantics*: The direct injective mapping between graph pattern and the underlying graph.
  - *No-repeated-edges semantics*: Only edge variables need to be injective; labels, nodes and properties need not be. It is currently used in Cypher.
  - *No-repeated-nodes semantics*: Only node variables need to be injective.

The authors discuss the complexity of evaluating the match semantics for a bgps with projection. The evaluation problem (bgps with projection) is NP-complete for the homomorphism-based semantics and the three versions of the isomorphism-based semantics. However, in the concept of data-complexity (36), where the basic graph pattern is fixed, and the complexity is defined on the basis of the underlying graph size, the problem becomes solvable in polynomial time. For cgps, the authors further point out the lack of complexity analysis in Cypher and Gremlin.

Navigational expressions are used to navigate the topology of the underlying graph and can recursively match paths of arbitrary length. The authors focus on path expressions as they are the most common form of navigation queries. Due to the paths potentially having cycles, evaluation semantics for path queries are categorised into:

- *shortest path semantics*: Paths with the minimal length that satisfy the constraint are considered under this semantics.
- *arbitrary path semantics*: All paths are considered under this semantics, which could be infinitely many, which is why the existence of a path is a more practical use-case of this semantics.
- *no-repeated-node semantics*: All paths are considered where the node appears exactly once in the path (simple paths).
- *no-repeated-edge semantics*: All paths are considered where an edge appears exactly once in the path. Cypher uses this semantics.

Finding nodes connected by arbitrary paths or finding a shortest path satisfying a regular path query can be done in polynomial time, whereas for no-repeated-node/edge semantics, the problem becomes intractable. The authors conclude that providing the categorisation and examples should help in understanding query languages and can be used as a guide for any proposed query language.

6.1.2 The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey

Sahu *et al.* (2) conducted a survey on the major user challenges with graph processing systems in industry and academia through an online questionnaire and reviewing mailing lists/bug reports (2). The participants provided an overview of entities being stored, the size of the graphs, graph processing software, used and the major computations done on graph data. They found scalability and visualisation being the two major challenges for existing software. Furthermore, they found even small to medium size organisations using large graphs ($>$ 1 billion edges) pointing to the pervasiveness of large graph data.

6.2 Graph Query Languages

Due to the existence of multiple graph management systems and the space being fragmented, multiple graph query languages exist which we describe below. The languages

can be imperative or declarative, however they have two common features - support for i) pattern matching and ii) navigational queries (10). There has been an effort by the LDBC and the industry to standardise the languages which first led to the joint development of the G-CORE research language (37) and eventually resulted in the formation of SQL/PGQ and GQL by ISO, described in detail in Background (Section 2.2).

6.2.1 Cypher

Cypher (7) is a declarative property graph query language initially developed by Neo4j (4) team and was open sourced as the OpenCypher project[1]. It is used by Neo4j, SAP Hana (17), Memgraph (6) and was one of the first languages to introduce ASCII art-syntax for graph pattern matching. Pattern matching is the central feature of the language; an example `MATCH` clause is shown in Listing 6.1. A Cypher query takes a property graph as input and outputs a table. Cypher structures the queries linearly, which is why in Listing 6.1 the projection (`RETURN` statement) is at the end of the query. Furthermore, Cypher supports graph updates, filtering, aggregation, and chaining of queries (7).

```
1 // filter accounts with aid=5 and return the ids of the accounts to which
      it transfers money.
2 MATCH (a1:account{aid:5})-[t:transfers]->(a2:account)
3 RETURN a2.id
```

Listing 6.1: Sample pattern matching in Cypher.

6.2.2 Gremlin

Gremlin (9) is a graph query language, part of the Apache TinkerPop graph framework with a focus on navigational (path) queries and a more functional syntax. Gremlin supports both imperative and declarative syntax. Gremlin is used in Amazon Neptune (38), JanusGraph (39) and Azure Cosmos DB (40). Furthermore, Gremlin traversals can be evaluated as OLAP and OLTP graph system traversals using the Gremlin Traversal Machine (41), a distributed, graph-based virtual machine. Gremlin queries can be embedded inside a host programming language leading to language variants like Gremlin-Java, Gremlin-Groovy, Gremlin-Python, etc.

A sample of a `MATCH` query in Gremlin is shown in Listing 6.2. Gremlin further supports recursive, branching, path and mutating (changing the graph) traversals (41).

[1] https://www.opencypher.org

```
1 //g is the graph, V() returns the set of vertices in the graph,
2 // .as is used to define a temporary variable 'a1',
3 // __ applies to the command one level up.
4 g.V().
5   match(
6     __.as('a1').hasLabel('account').has('aid', 5).out('transfers').hasLabel
          ('account').as('a2')
7   ).select('a2').id
```

Listing 6.2: Sample pattern matching in Gremlin.

6.2.3 PGQL

PGQL (8) is a property graph query language designed by Oracle with SQL-like syntax, and used in Oracle PGX (42). Due to the syntax similarities with SQL and multiple SQL features like filtering, aggregation, ordering, and nested queries, it is intuitive to use for existing SQL users. PGQL introduced the concept of using the *graph* as an intrinsic type, allowing queries to construct and return graphs (by the `SELECT GRAPH` statement). PGQL, like Cypher is based on the paradigm of graph pattern-matching. Pattern-matching queries in PGQL involves three clauses - `SELECT`, `FROM` and `WHERE` followed by optional clauses for limiting, grouping and ordering results. A `MATCH` clause example in PGQL is shown in Listing 6.3 where *amlGraph* contains the account activity data of bank customers (Figure 2.3).

```
1 SELECT a2.id
2 FROM amlGraph
3 WHERE
4 (a1 WITH aid = 5)-[:transfers]->(a2)
```

Listing 6.3: Sample pattern matching in PGQL.

6.2.4 G-CORE

LDBC[1] initiated the design of a new graph query language starting with G-CORE (37), a composable and tractable graph query language with Cypher-like syntax (7). G-CORE has been designed to support major features from PGQL, Cypher and Gremlin. It introduces the concept of treating *paths* as first-class citizens (37) which allows paths to have labels and properties. Furthermore, direct operations on paths like filter, match are also supported. G-CORE uses the Path Property Graph Model, an extension of the Property Graph Model with paths. G-CORE is a design language and was not implemented in any industrial

[1] https://ldbcouncil.org/

system. G-CORE queries have a complexity analysis to show they are tractable in data complexity (36), a minimum requirement for a practical query language. Design decisions have been taken by the authors like using shortest path semantics instead of all paths to avoid creating an NP-complete problem. A pattern matching example in G-CORE is shown in Listing 6.4. Furthermore, G-CORE supports graph aggregation, multi-graph queries, weighted shortest paths and reachability queries. For further queries, we refer the reader to the original paper (37).

```
1 //GCORE always returns a value through the CONSTRUCT clause.
2 CONSTRUCT (a2.id)
3 MATCH (a1:account)-[:transfers]->(a2:account)
4 WHERE a1.aid = 5
```

Listing 6.4: Sample pattern matching in G-CORE.

6.2.5 GSQL

GSQL is a query language used in Tigergraph (5) with extended support for aggregation-based graph analytics (43). Pattern matching in GSQL is similar to PGQL syntax and an example is shown in Listing 6.5. Deutsch *et al.* (43) introduce the concept of data containers called accumulators to store aggregated internal values. Accumulators are of two types:

- Global accumulators (defined using @@): They have a single instance per query for calculating global aggregates.
- Vertex accumulators (defined using @): Each vertex stores its own accumulator instance for calculating vertex-centric aggregates. An example of a vertex accumulator is shown in Listing 6.6 where money transferred per account is stored in *totalTransfer*.

Accumulators allow composition, single-pass multi-aggregation (by different grouping criteria) and parallelisation opportunities. Furthermore, GSQL supports loops enabling the specification of graph algorithms like Pagerank without using UDFs. The authors achieve 3X speedup for aggregation queries using accumulators over using SQL-style aggregation (43).

```
1 SELECT a1.aid
2 FROM amlGraph AS account:a1 -(transfers>:t)- account:a2
3 WHERE a1.aid = 5;
```

Listing 6.5: Sample pattern matching in GSQL.

```
1 SumAccum<int> @totalTransfer
2 SELECT a1
3 FROM account:a1 -(transfers>:t)- account:a2
4 ACCUM a1.@totalTransfer += t.amount;
```

Listing 6.6: Sample vertex accumulator in GSQL.

6.3 Relational Systems with Graph Query Support

6.3.1 Graph Pattern Matching - Do We Have to Reinvent the Wheel

Gubichev *et al.* (11) use Lehigh University Benchmarks (LUBM) on three different data models - RDF, property graph and relational systems to compare graph pattern matching performance. Multiple systems are used for different models like Neo4j, Sparksee[1] for Property Graph, Virtuoso[2] for relational systems. The queries are rewritten to take into account rules of the ontology and data is enriched with inference from the ontology rules as LUBM has primarily been used for RDF data. Under analysis, the authors mention disadvantages of using API over a declarative language:

- Intermediate results are immediately returned reducing performance.
- Application developer has the additional work of query optimisation and is labour-intensive.

The authors also point out that native graph stores have suboptimal performance for pattern matching operations with queries timing out for large datasets on Neo4j and Sparksee (11). The authors conjecture that relational systems with graph query languages and graph operations are best suited for pattern matching operations.

6.3.2 Extending SQL for Shortest Paths

It is cumbersome to perform reachability and cheapest path queries on relational databases using plain SQL (SQL:1999). The three ways currently used are: i) Recursion ii) Programming State Model (PSM) iii) Explicit chain of joins. However, these methods make the query writing error-prone and verbose, break the declarative paradigm of SQL and can have poor performance due to not using specialised graph algorithms (44). Leo *et al.* (44) propose a new way - an extension to SQL introducing a new predicate for reachability and

[1] https://dbdb.io/db/sparksee
[2] https://virtuoso.openlinksw.com/

a new function to calculate the cost of the cheapest path. This is implemented in MonetDB by adding two operators to the compiler – i) GraphSelect - model a graph based on the edges and vertices parameters and then verify whether each tuple is connected and ii) GraphJoin - cross product followed by a GraphSelect. Their prototype shows that dynamic graph construction dominates the query run time for a single source-destination pair but batching multiple source-destination pairs amortises the time for construction reducing the execution time to become almost linear with respect to number of edges.

6.3.3 Grail

Due to the popularity of graph analytics, specialised GDBMS have become popular. However, Yildirim *et al.* (45) argue that specialised systems for graph processing are not required, and already ubiquitous RDBMSs can be used for supporting graph queries with an intermediate layer for query translation. They present Grail, an intermediate layer that translates and optimises graph-like queries into T-SQL to be used directly on the Microsoft SQL Server RDBMS. Grail stores the graphs as vertex and edge tables in the RDBMS and transforms the algorithms and intermediate values using relational algebra. In the comparison for with Giraph [1], GraphLab (46), the transformed T-SQL queries perform best for large data sizes and the specialised systems time out, while for smaller data sets, the specialised systems are faster.

6.3.4 IBM Db2 Graph

Tian *et al.* (47) add Gremlin support to IBM Db2 database where they optimise synergy with existing analytic queries (SQL and ML queries) over optimising for performance. The authors are able to extract graphs using a configuration file that defines mapping between vertex and edge tables; automatic extraction on existing tables is also possible by using of primary and foreign key constraints (47).

6.3.5 GRFusion

Hassan *et al.* (48) add graph query support in VoltDB, an in-memory relational system focussing on transactional (OLTP) workloads. They materialise graph views using an in-memory data structure (based on adjacency list) that has bi-directional linkage to the relational data. The advantage of having a separate topology avoids joins during graph

[1] http://giraph.apache.org/.

traversals. GRFusion supports the graph and the relational model in the same query execution pipeline which allows the predicate pushdown optimisation for traversals.

6.3.6 SQLGraph – When ClickHouse Marries Graph Processing

Zheng (49) presents his work of forking out ClickHouse and extending it with a graph processing engine to create SQLGraph. The presenter adds graph query support to an existing RDBMS as they recognise the challenges of graph processing in plain RDBMSs:

- Slow traversal of graphs through joins.
- Difficulty in extraction of graphs.
- Cumbersome to support iterative graph algorithms using recursive SQL queries.

SQLGraph treats graphs as first-class citizens, uses the Clickhouse API for graph visualisation, and supports graph algorithms like pagerank, community detection, random walks, etc.. Furthermore, additional UDFs can be added to support different algorithms in Python or C++. For running graph queries, users have to define vertex and edge tables which are converted to a CSR and Compressed Sparse Column (CSC) representation in the background for faster access. SQLGraph runs on a single machine and main memory, as main memory is able to support large graphs (with billions of edges) and it is costly to have independent partitions of a graph. The system performs better than Neo4j (4), TigerGraph (5), JanusGraph (39) when compared for pagerank and multi-hop path queries.

6.3.7 GraphGen

Xirogiannopoulos *et al.* (12) present the GraphGen framework which allows the extraction of graph views from RDBMS and supports analytics through GraphGenDL (based on Prolog) or Java programs. The system has a relational backend, and a relational + graph frontend where the extracted graphs are independent entities supporting complex graph algorithms using the Java API (50). The authors enumerate the challenges of using such an approach – i) difficulty in mapping graph edges/nodes with relational databases, especially in the case of self joins ii) keeping track of underlying database updates and iii) lack of ability to build a new schema or rearrange indices for optimisation. The authors' experiments expose new challenges for query optimisation:

- Query rewriting in optimisers

- Challenges in dealing with data blowup due to large self joins (finding a concise representation to load the view in memory)
- Building a cost model to decide where to execute the queries (either in memory for complex queries or pushing the computation into the database for simple or infrequent queries)

They conclude that CSR is a suitable representation to load graphs in memory.

6.3.8 GRainDB: A Relational-core Graph-Relational DBMS

Jin *et al.* (51) extend DuckDB with graph modelling, visualisation, and querying capabilities to create a new DBMS called GRainDB. The authors comment on the complementary nature of GDBMSs and RDBMSs and users benefitting from their combined capabilities like RDBMS being able to model n-ary relations while GDBMS providing the ability to store semi-structured data. The authors recognise many-to-many join being common in graph workloads, for which they extend DuckDB's physical storage and query processor by adding physical RIDs (`rowids`) as columns to tables with primary and foreign key constrains. These dense integer `rowids` based columns are utilised for joins instead of column values based joins in vanilla RDBMSs. Futhermore, the authors add visualisation support for nodes, edges of a graph and pre-create an index based on `rowids` when users model a graph. Their experiments a performance speedup in many-to-many joins using these pre-defined indexes and utilising `rowids` for joins.

6.3.9 Relation to Existing Papers

The major difference between our work and the elaborated systems is that none of them make design decisions related to SQL/PGQ implementation. However, some of their results motivate our design choices like:

- Xirogiannopoulos *et al.* (12) using CSR for in-memory graph representation.
- Leo *et al.* using batched mamy source-destination pairs to amortise the running time of graph construction.
- GRainDB (51) using `rowids` in DuckDB for improving join performance.

In terms of design, Clickhouse implementation is the closest to our implementation as it also materialises graphs in-memory, on a single machine and utilises CSR representation for graphs. However, its implementation is not open source and the design details are

only mentioned in a presentation. The authors also do not mention any algorithm specific optimisations, however they do provide a more diverse range of algorithms and use cases than our current implementation.

GRainDB (51) also uses DuckDB as a backend but they fork from the original DuckDB and create a new system. In both systems, the idea of using `rowids` is important as they use it for creating their indices and joins and we use these dense ids for CSR representation of graphs. GRainDB supports a custom query language while we support SQL/PGQ. Furthermore, they go with the design decision of making changes to the executor and working outside of DuckDB for improving performance which comes with the additional challenge of popularising the system for other users. In contrast, our non-intrusive approach of using UDFs and SQL queries, makes it possible to keep it in the original DuckDB as an extension.

6.4 Reachability Algorithms

6.4.1 Floyd-Warshall

Floyd Warshall is an all-pairs shortest path algorithm (52) (which is a special case of the multi-source multi-destination shortest path problems that are sometimes formulated in SQL/PGQ). The algorithm uses dynamic programming. For finding a shortest path between nodes i and j, the algorithm uses an intermediate node k, where the shortest path could either pass through k or not pass through it. This leads to the recursive strategy described in Listing 6.7. The space requirement for the algorithm is $\theta(V^2)$ and run time complexity is $O(V^3)$, which makes it unsuitable for using it with SQL/PGQ (as graphs generally tend to have large number of vertices). There are parallel versions of the algorithm to reduce the space and run-time complexity by partitioning the adjacency matrix to different processors and using individual processors to compute a part of the result matrix (53). However, the methodology would require low level constructs (like Message Parsing Interface (54)) for broadcasting information and synchronisation using a barrier which would be a major implementation overhead in DuckDB.

```
1 if(k == 0):
2   shortestPath(i, j, 0) = wt(i, j)
3 else:
4   shortestPath(i, j, k) = min( shortestPath(i, j , k-1), shortestPath(i, k,
        k-1) + shortestPath(k, j, k-1 ))
```

Listing 6.7: Recursive strategy for Floyd-Warshall algorithm.

6.4.2 Multiple-Source Shortest Paths in Planar Graphs

Klein (55) introduces an efficient multiple source-destination pair shortest path algorithm for planar graphs. The basic idea of the algorithm is to find the shortest path tree rooted at a node, and then iteratively modify the tree to find shortest path tree rooted at its neighbours. Some edges are removed, and some are added for the tree rooted at the neighbour but we can re-use edges and reduce the amount of computation as compared to creating the tree from scratch. The optimisation works on the principle that edges that are not in the tree form a tree in the shortest path tree of the planar dual graph. This optimisation only works for planar graphs which social network graphs are not.

Cabello *et al.* (56) extend the Klein algorithm to general graphs by modifying the shortest path tree data structure to a kinetic data structure (57). However, the algorithm performance depends on the genus (58) of the graph. For random graphs like social networks, with a high genus the algorithm does not scale well.

6.4.3 Delta-stepping: a Parallelizable Shortest Path Algorithm

Meyers *et al.* (59) introduce Delta (Δ) stepping, a single source shortest path algorithm. In the algorithm, nodes are divided into buckets based on Δ. Lighter nodes (weight $<$ Δ) are all traversed in the same iteration (can be done parallelly) while heavy nodes (weight $>$ Δ) are only traversed when all edges of lighter nodes have been relaxed. The performance of the algorithm heavily depends on the parameter (Δ); $\Delta = 1$ and integer edge weights leads to a version of Dijkstra's (16) algorithm and Δ = max (node degree) leads to Bellman-Ford. The idea of the algorithm is to find a compromise between the two extremes, however the authors note that doing so for a parallel version of the algorithm is not always possible. The authors analyse the run time complexity for various graph and edge weight possibilities to show a high number of cases where the algorithm performs in linear time on directed graphs with constant maximum degree.

6.4.4 Fast Exact Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling

Akiba *et al.* (60) present a new method for shortest-path distance queries by using a combination of pruning on BFSs and *distance aware-2 hop labelling.* A 2 hope labelling is based on the idea of pre-computing shortest paths between every pair of vertice (u,v) and an intermediate node w between that pair. The authors run BFS for each vertex in the order v_1, v_2,... v_n, given n vertices in the graph, and add the distance information to labels

of the visited vertices. At every new BFS iteration, there is a set of vertices S for which the minimum distance can be correctly computed if the shortest path passes through a node in S. For a BFS starting from v visiting u, if there is a vertex w∈S such that d(v, u) = d(v, w) + d(w, u), then the authors prune u. Using this pruning leads to a reduction in search space, specially for the latter iterations of BFSs. Furthermore, the authors note that the order of vertices is important; their experiments show visiting vertices with a higher degree or closeness-centrality first leads to a faster execution time than visiting vertices in random order. Furthermore, the authors use bit-operations for optimising the algorithm, and suggest variants for handling directed and weighted edges. This is a useful algorithm based on BFS for all pair-shortest path queries, however SQL/PGQ more commonly uses many pair source-destination queries which are a subset of all pair shortest-path problem.

6.4.5 Parallel Array Based Single and Multi Source BFS on Large Dense Graphs

Kaufmann *et al.* (32) extend the ideas of MSBFS defined in Section 2.3 and present multi-threaded single-source and multi-source BFS algorithms. These algorithms have a lower memory requirement than MSBFS as running a separate instance of MSBFS on each core leads to high memory usage, and multiple sources are required for the speedup given by MSBFS. The authors present parallelised bottom-up and top-down versions of the algorithms separated by a barrier using atomic instructions instead of locking. Furthermore, they found that by static partitioning of vertices and degree ordered vertex relabelling, the majority of the work was taken up by the initial worker threads. Even if the partitioning ends up being balanced, due to small-world networks, a run time skew would be observed in each iteration (32). Instead, they use fine-grained tasks with work stealing to achieve a better balance of vertices, and distribute degree-ordered vertices in a round-robin scheduling across worker threads.

6.4.6 Efficient Batched Distances Closeness Centrality, Betweenness Centrality in Unweighted and Weighted Graphs

Centrality algorithms determine the importance of vertices in a graph. There are three types of centralities discussed by the authors:

- Degree centrality: number of incident edges on the vertices.
- Closeness centrality: ranks vertices by average geodesic distance (shortest paths) to all other vertices.

- Betweenness centrality: number of shortest paths between any two vertices that a vertex is on.

Closeness and betweenness are computationally expensive (due to computation of all pairs geodesic computations), therefore Then *et al.* (61) suggest using batched execution to reduce random access and leverage vectorised execution for faster computation. The authors utilise bitfields and bitwise operations, store data with spatial locality and vectorised execution on modern CPU caches for performance improvements. The authors present MS-BFS (Section 2.3), batched versions of Bellman Ford, and Brandes' (62) algorithm for the different centrality computations in weighted and unweighted graphs (61).

6.4.7 GraphBLAS Solution to SIGMOD 2014 Programming Contest Using Multi-Source BFS

Elekes *et al.* (63) implement and modify the MSBFS algorithm to solve the graph queries involved in the 2014 SIGMOD Programming contest[1]. The authors use an adjacency matrix representation to store the graphs with seen, visit and visit_next all being matrices too. This is a different approach where matrix multiplication is used to calculate MSBFS, however it is 1-2 orders of magnitude slower than the top solutions of the contest that employed a more low-level approach.

[1] https://github.com/ldbc/sigmod2014-contest-graphblas

7

Conclusion

In this thesis, we explored architecting a minimal extension of SQL/PGQ in DuckDB. We first scoped down the specification to reduce the changes being introduced in the underlying database, improve performance and decrease implementation complexity while still retaining the core features for graph query processing. We explored different design choices for the implementation and researched the literature for a reachability algorithm that scales for our use case. The main contribution of the thesis is that we were able to find a pluggable way to integrate SQL/PGQ in a database.

7.1 Research Questions

In the following section, we answer the research questions we had defined in Section 1.2.2.

7.1.1 RQ 1: How can SQL/PGQ be mapped to relational query operators including the new path-finding operator?

As defined in Section 3, we outline the approaches selected for mapping the SQL/PGQ clauses to relational operators where creation of property graphs is done by defining a similar execution pipeline as `CREATE VIEWS`, simple pattern matching is mapped and rewritten to a cross product query (Section 4.1), while the path-finding operator (Kleene star) is mapped to a `SELECT` query defined using custom UDFs. We give reasoning on why we chose the UDF approach for the Kleene star implementation and how it helps in architecting a minimal and maintainable extension (Section 3.2.2).

7.1.1.1 RQ 1a: What design decisions need to be made to scope down the extension for improving performance and feasibility?

In Section 3.1, we outlined the decisions to scope down the specification and the reasoning behind scoping elements out. We ensured that the basic support for simple pattern matching and path-finding was still part of the specification and the scoping down was done to reduce implementation complexity. Some of the scoped out features can be added as part of future work based on the SQL/PGQ adoption and features requested by users.

7.1.2 RQ 2: What path-finding algorithm is needed to minimally support SQL/PGQ?

We explore the literature and select MSBFS (25) for our use case of supporting path-finding and reachability. It is a batched algorithm that is useful for many source many destination queries, which are expected to occur frequently in SQL/PGQ queries using the Kleene star operator. With our experiments (Section 5.4), we find that MSBFS integrates well with the DuckDB vectorised execution model and batched algorithm scales well for graph algorithms.

7.1.2.1 RQ 2a: How can the algorithm be integrated in a vectorised query engine?

To integrate the algorithm, we make use of UDFs to have a pluggable extension and use their default parellelisation. As the algorithm is batched it fits well with the vectorised execution model of DuckDB. Furthermore, we try adding some sub-optimisations to the algorithm like running a BFS only for unique source vertices (Section 4.3.1), introducing dynamic states to reduce the amount of iterations (Section 4.3.2), and using SIMD hints along with Clang compilation to try SIMDise the `for` loops (Section 5.4.3). From our experiments, we observe the algorithm scales well with an increase in threads (Section 5.4).

7.1.2.2 RQ 2b: What data structure representation will be required for implementing the algorithm?

We utilise CSR structure with dense keys to represent the graphs and store them as large arrays in memory. Even large graphs (>1 million nodes) can be materialised in memory, and the creation of the structure is also done with DuckDB UDFs as part of a SQL query (Section 4.2). Furthermore, CSR creation is a relatively fast process as demonstrated in Section 5.3.

7.1.3 RQ 3: How can the performance of the system be systematically evaluated?

LDBC's SNB BI workload is used for evaluation as it generates a realistic social network dataset that is a common application for graph workloads. Query 13 of the BI workload is used to test the scalability of the approach as it involves a Kleene star operator with the Person vertices and Person_knows_Person edges (Section 5.2). We are able to test the scalability of our UDFs in DuckDB using this approach as seen in Section 5.

7.2 Future Work

As this thesis was the first minimal implementation, various directions for future work exist. Support can be added for weighted graphs, path queries, specially cheapest path calculation that will require a different algorithm than MSBFS. Worst-Case optimal join (64) support can be added to DuckDB as many pattern matching queries end up with cyclic paths and this is also useful for non-graph queries. Furthermore, the MSBFS implementation can be made SIMD friendly to take advantage of 512 bitsets in cache. Work can be done to support filtering in CSR to accomodate `WHERE` clause in the Kleene star query, reusing CSR for larger graphs and for implementing the parser as an extension module to DuckDB.

Acronyms

CSR Compressed Sparse Row. ii, iii, vi, 4, 9, 10, 15, 16, 17, 19, 21, 22, 23, 24, 25, 26, 29, 30, 31, 32, 46, 47, 48, 53, 54

DBMS Database Management system. 2, 4, 47

GDBMS Graph Database Management System. 1, 38, 45, 47

LDBC Linked Data Benchmark Council. 12, 41, 42

LSQB Labelled Subgraph Query Benchmark. 12

MSBFS Multi-Source Breadth First Search. vi, vii, 8, 27, 28, 33, 34, 36, 50, 51, 53, 54

PG Property graph. 5, 6

RDBMS Relational Database Management System. ii, 1, 2, 23, 38, 45, 46, 47

SQL/PGQ SQL Property Graph Queries. ii, iv, vi, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 15, 16, 17, 18, 19, 41, 47, 48, 50, 52, 53

UDF User Defined Function. ii, 17, 19, 31, 34, 43, 46, 48, 52, 53, 54

References

[1] Florian Holzschuher and René Peinl. **Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j**. In Giovanna Guerrini, editor, *Joint 2013 EDBT/ICDT Conferences, EDBT/ICDT '13, Genoa, Italy, March 22, 2013, Workshop Proceedings*, pages 195–204. ACM, 2013. 1

[2] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. **The ubiquity of large graphs and surprising challenges of graph processing: extended survey**. *VLDB J.*, **29**(2-3):595–618, 2020. 1, 2, 40

[3] Kangfei Zhao and Jeffrey Xu Yu. **All-in-One: Graph Processing in RDBMSs Revisited**. In *SIGMOD*, pages 1165–1180. ACM, 2017. 1

[4] Neo4j. **Neo4j**. `https://neo4j.com/`, 2022. 1, 5, 12, 41, 46

[5] **TigerGraph**. `https://www.tigergraph.com/`. 1, 5, 43, 46

[6] **Memgraph**. `https://memgraph.com/`. 1, 5, 12, 41

[7] 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 *SIGMOD*, pages 1433–1445. ACM, 2018. 1, 5, 41, 42

[8] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. **PGQL: a property graph query language**. In *GRADES at SIGMOD*. ACM, 2016. 1, 5, 42

[9] **The Gremlin Graph Traversal Machine and Language**. `https://tinkerpop.apache.org/gremlin.html`. 1, 41

[10] 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. 1, 38, 41

[11] Andrey Gubichev and Manuel Then. **Graph Pattern Matching - Do We Have to Reinvent the Wheel?** In *GRADES*, pages 8:1–8:7. CWI/ACM, 2014. 1, 44

[12] Konstantinos Xirogiannopoulos, Virinchi Srinivas, and Amol Deshpande. **GraphGen: Adaptive Graph Processing using Relational Databases**. In *GRADES at SIGMOD*, pages 9:1–9:7. ACM, 2017. 1, 46, 47

[13] 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 *SIGMOD*, pages 345–359. ACM, 2020. 1

[14] Alastair J. Green. **SQL ... and now GQL**. `https://www.linkedin.com/pulse/sql-now-gql-alastair-green/`, 2019. 1, 6, 7

[15] Renzo Angles. **The Property Graph Database Model**. In *AMW*, **2100** of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018. 1

[16] Edsger W. Dijkstra. **A note on two problems in connexion with graphs**. *Numerische Mathematik*, **1**:269–271, 1959. 2, 49

[17] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. **The Graph Story of the SAP HANA Database**. In *DBIS*, **P-214** of *LNI*, pages 403–420. GI, 2013. 2, 41

[18] Mark Raasveldt and Hannes Mühleisen. **DuckDB: an Embeddable Analytical Database**. In *SIGMOD*, pages 1981–1984. ACM, 2019. 2, 10

[19] Eric W. Weisstein. **Simple Graph**. `https://mathworld.wolfram.com/SimpleGraph.html`, 2022. 4

[20] Renzo Angles. **The Property Graph Database Model**. In *AMW*, **2100** of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018. 5

[21] Andy Witkowski Jan Michels. **Graph database applications with SQL/PGQ**. https://download.oracle.com/otndocs/products/spatial/pdf/AnD2020/AD_Develop_Graph_Apps_SQL_PGQ.pdf, 2020. 6

[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 *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 2246–2258. ACM, 2022. 6

[23] **GQL Standard**. https://www.gqlstandards.org/. 7

[24] Moritz Kaufmann, Manuel Then, Alfons Kemper, and Thomas Neumann. **Parallel Array-Based Single- and Multi-Source Breadth First Searches on Large Dense Graphs**. In Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Bress, editors, *EDBT*, pages 1–12. OpenProceedings.org, 2017. 8

[25] 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**. *VLDB*, **8**(4):449–460, 2014. 8, 26, 27, 53

[26] Brian Wheatman and Helen Xu. **Packed Compressed Sparse Row: A Dynamic Graph Representation**. In *2018 IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*, pages 1–7. IEEE, 2018. 9

[27] Hannes Mühleisen and Mark Raasveldt. **1000 Days of DuckDB Pareto Principle Still Holds**. https://dsdsd.da.cwi.nl/slides/dsdsd-duckdb.pdf, 2021. 10

[28] Mark Raasveldt. **Push-Based Execution in DuckDB**. https://dsdsd.da.cwi.nl/past_talks/duckdb-push-based-execution/, 2021. 11

[29] DuckDB Documentation Team. **DuckDB Docs SELECT Statement - RowIDs**. https://duckdb.org/docs/archive/0.3.1/sql/statements/select, 2022. 11

[30] 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. 12

[31] 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. 12

[32] Moritz Kaufmann, Manuel Then, Alfons Kemper, and Thomas Neumann. **Parallel Array-Based Single- and Multi-Source Breadth First Searches on Large Dense Graphs**. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 1–12. OpenProceedings.org, 2017. 16, 50

[33] Dr. Thomas Neumann. **Modern Databases: Algebraic Operators**. https://db.in.tum.de/teaching/ss22/moderndbs/chapter6.pdf?lang=en, 2022. 27

[34] **CWI Scilens Cluster**. https://projects.cwi.nl/scilens/. 30

[35] E. PRUDHOMMEAUX. **SPARQL query language for RDF**. *http://www.w3.org/TR/rdf-sparql-query/*, 2008. 38

[36] Moshe Y. Vardi. **The Complexity of Relational Query Languages (Extended Abstract)**. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 137–146. ACM, 1982. 39, 43

[37] 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 *SIGMOD*, pages 1421–1432. ACM, 2018. 41, 42, 43

[38] 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 *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. 41

[39] **JanusGraph**. `https://janusgraph.org/`. 41, 46

[40] Peter Boncz. **A survey on graph query languages**. `https://www.youtube.com/watch?v=oJmuRM9xpdU`, 2020. 41

[41] Marko A. Rodriguez. **The Gremlin graph traversal machine and language (invited talk)**. In *Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 25-30, 2015*, pages 1–10. ACM, 2015. 41

[42] pgx. **Orcal PGX**. `https://docs.oracle.com/en/database/oracle/property-graph/22.1/spgdg/using-graph-server-pgx.html`, 2022. 42

[43] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. **Aggregation Support for Modern Graph Analytics in TigerGraph**. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 377–392. ACM, 2020. 43

[44] Dean De Leo and Peter A. Boncz. **Extending SQL for Computing Shortest Paths**. In *GRADES@SIGMOD*, pages 10:1–10:8. ACM, 2017. 44

[45] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. **GRAIL: a scalable index for reachability queries in very large graphs**. *VLDB J.*, **21**(4):509–534, 2012. 45

[46] 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. 45

[47] 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 *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. 45

[48] Mohamed S. Hassan et al. **GRFusion: Graphs as First-Class Citizens in Main-Memory Relational Database Systems**. In *SIGMOD*, pages 1789–1792. ACM, 2018. 45

[49] Tianqi Zheng. **SQLGraph: When ClickHouse marries graph processing**. `https://presentations.clickhouse.com/meetup24/2.%20SQLGraph%20--%20When%20ClickHouse%20marries%20graph%20processing%20Amoisbird.pdf`, 2019. 46

[50] Konstantinos Xirogiannopoulos. *Enabling Graph Analysis Over Relational Databases*. PhD thesis, University of Maryland, College Park, MD, USA, 2019. 46

[51] Guodong Jin, Nafisa Anzum, and Semih Salihoglu. **GRainDB: A Relational-core Graph-Relational DBMS**. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. 47, 48

[52] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. 48

[53] Vipin Kumar and Vineet Singh. **Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem: A Summary of Results**. In *Proceedings of the 1990 International Conference on Parallel Processing, Urbana-Champaign, IL, USA, August 1990. Volume 3: Algorithms and Applications*, pages 136–140. Pennsylvania State University Press, 1990. 48

[54] **Chapter 1: Introduction To Mpi-2**. *Int. J. High Perform. Comput. Appl.*, **12**(1-2):12–14, 1998. 48

[55] 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. 49

[56] Sergio Cabello and Erin W. Chambers. **Multiple source shortest paths in a genus g graph**. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 89–97. SIAM, 2007. 49

[57] Leonidas Guibas. **Kinetic Data Structures - A State of the Art Report**. *Proc. Workshop Algorithmic Found. Robot.*, 08 1998. 49

[58] Eric W. Weisstein. **Graph Genus**. `https://mathworld.wolfram.com/GraphGenus.html`, 2022. 49

[59] Ulrich Meyer and Peter Sanders. **[Delta]-stepping: a parallelizable shortest path algorithm**. *J. Algorithms*, **49**(1):114–152, 2003. 49

[60] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. **Fast exact shortest-path distance queries on large networks by pruned landmark labeling**. In *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. 49

[61] 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. 51

[62] Ulrik Brandes. **A faster algorithm for betweenness centrality**. *Journal of mathematical sociology*, **25**(2):163–177, 2001. 51

[63] Márton Elekes, Attila Nagy, Dávid Sándor, János Benjamin Antal, Timothy A. Davis, and Gábor Szárnyas. **A GraphBLAS solution to the SIGMOD 2014 Programming Contest using multi-source BFS**. In *2020 IEEE High Performance Extreme Computing Conference, HPEC 2020, Waltham, MA, USA, September 22-24, 2020*, pages 1–7. IEEE, 2020. 51

[64] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. **Worst-case Optimal Join Algorithms**. *J. ACM*, **65**(3):16:1–16:40, 2018. 54

Appendix