Master Thesis

Literature Study: Integrating SQL/PGQ in DuckDB

by

Daniël ten Wolde

(2619049)

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

June 1, 2022

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

Literature Study: Integrating SQL/PGQ in DuckDB

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

1 INTRODUCTION

In this literature study we will be looking at, and attempt to answer the question: "What is the current state of graph query languages and what are their strengths and weaknesses?". We will start with an introduction on the different types of graph models. This is relevant as different graph query languages base themselves on a certain model. To the surprise of some [**?**]insert more citations here, there current does not exist a standard graph query language. This literature study will also cover why this is the case, and where the future will likely lie in terms of a standard language. There is ongoing development in the area of creating a standardized language referred to as Graph Query Language (GQL)insert citation.

The following Scopus query was used to find papers relevant to the research question. We will be selecting papers that have been cited the most, or have recently been published, thus not allowing for many citations. The query contains papers mentioning "graph pattern matching" and "path finding", this is relevant since these are key features found in graph query languages compared to Structured Query Language (SQL) which is used in relational database management systems (DBMSs).

1.1 Scopus Query

TITLE-ABS-KEY(DATABASE) AND (TITLE-ABS-KEY("*GRAPH PATTERN MATCHING*") OR TITLE-ABS-KEY("*PATH FINDING*") OR TITLE-ABS-KEY("*GRAPH QUERY LANGUAGE*")) AND (LIMIT-TO (PUBYEAR,2022) OR LIMIT-TO (PUBYEAR,2021) OR LIMIT-TO (PUBYEAR,2020) OR LIMIT-TO (PUBYEAR,2019) OR LIMIT-TO (PUBYEAR,2018)) AND (LIMIT-TO (SUBJAREA,"COMP")) AND (LIMIT-TO (LANGUAGE,"English"))

1.2 Inclusion/Exclusion criteria

2 THREATS TO VALIDITY

2.1 Internal Validity

2.2 External Validity

2.3 Construct Validity

2.4 Conclusion Validity

3 BACKGROUND

3.1 Graph models

3.2 Cypher

3.3 G-CORE

3.4 SPARQL

3.5 PGQL

3.6 SQL/PGQ

3.7 GQL

3.8 Graph Pattern Matching in GQL and SQL/PGQ

https://arxiv.org/pdf/2112.06217.pdf GQL and SQL/PGQ share a common graph pattern matching language (GPML) -> Extracts data from a graph which matches a graph pattern -> result is a set of path bindings where each variable is mapped to a node or edge in the graph. The main difference between GQL en SQL/PGQ is the way the data is represented, with the latter showing it in table form. A brief introduction of graphs is given. They consist of nodes which are connected with edges. Both can carry labels (Account, City, Transfer) and key/value pairs (isBlocked/No). Graphs can be either directed, partially directed or undirected. There can be multiple edges between two nodes. Self-loops are also possible. A path is an alternating sequence of nodes and edges, starting and ending with a node and subsequent nodes are connected with edges. Several conjuctive regular path queries (CRPQs) exist, such as SPARQL, Cypher, Oracle PGQL and TigerGraph's GSQL

A GPML pattern is as follows MATCH pattern optionally followed by a WHERE MATCH (x:Account WHERE x.isBlocked = 'No') MATCH(x) -> x here is called a element variable (node patterns) MATCH -[e]-> -> e (edge patterns)

There are also quantifiers, similar to regex m,n between m and n repetitions m, m or more repetitions * 0 or more repetitions + 1 or more repetitions.

Two forms of union path pattern union (using |) multiset alternation (using |+|) Conditional variables (Need to understand better)

To figure out the direction of an edge (What happens here with undirected edges, what is the the outcome of IS SOURCE OF and

IS DESTINATION OF e IS DIRECTED -> true if e is bound to a directed edge s IS SOURCE OF e -> true if s is bound to the source of e d IS DESTINATION OF e -> true if d is bound to the destination of e

GPML queries must demonstrably terminate, this is done using restrictors and selectors. Restrictor: Path predicate (imposes a condition stating which paths are acceptable) such that the number of matches cannot be infinite) TRAIL no repeated edges ACYCLIC no repeated nodes SIMPLE no repeated nodes, except that the first and last nodes may be the same Selector: Algorithm that conceptually partitions the solution space on the endpoints and selects a finite set of matches from each partition. ANY SHORTEST selects one path with shortest length from each partition ALL SHORTEST selects all paths in partition that have minimal length ANY selects one path in each partition arbitrarily ANY K selects arbitrary path ks in each partition SHORTEST K select shortest k paths SHORTEST K GROUP Groups with same length k, selects all paths in the first k groups.

Key steps in pattern matching execution model Normalisation Expansion Rigid-pattern matching Reduction and deduplication

Graph database applications with SQL/PGQ https://download.oracle.com/otndocs/products/spatial/pdf/AnD2020/AD_Develop_Graph_Apps_SQL_PGQ.pdf Contains a good example on the use of graph view (Anti-Money Laundering) Query is shorter and more intuitive

3.9 Query, Analysis, and Benchmarking Techniques for Evolving Property Graphs of Software Systems by Gábor Szárnyas – PhD dissertation, BME (2019)

Conceptual graph data models [**?**] Untyped graphs: hold no type information Definition 1 (untyped graph or homogeneous graph) G = (V,E,src,trg) where V is a set of nodes, E a set of edges. Functions src,trg : $E \rightarrow V$ are total functions respectively assigning the source and target node to each edge. Definition 2 (graph elements) We refer to the union of nodes and edges as graph elements Definition 3 (weighted graphs) A weighted graph is a graph where each edge is assigned a weight $w \in \Re$ Definition 4 (edge-typed graph or multiplex graph) An edge-typed graph is defined as G = (V,E,src,trg,T,type), where T is a set of edge types and function type $E \rightarrow T$ assigns a single type to each edge. Useful since we still abstract away information regarding the node labels, but provides more insight compared to untyped graphs. Definition 5 (node-labelled graph) Defined as G = (V,E,src,trg,L,labels), where L is a set of node labels, and function labels : $V \rightarrow 2^L$ assigns a set of labels to each node Definition 6 (labelled graph) G = (V,E,src,trg,L,T,labels,type)

Definition 7 (property graph) G = (V,E,src,trg,L,T,labels,type,Pv,Pe) P_v: set of node properties. $p \in P_v$ is a function p : $V \rightarrow D$, which assigns a property value $d \in D$ to node $v \in V$, if v has property p, otherwise returns NULL P_e: set of edge properties. $p \in P_e$ is a function P : $E \rightarrow D$, which assigns a property value $d \in D$ to edge $e \in E$, if e has property p, otherwise returns NULL

Path property graphs mentioned in G-CORE, treating paths as first-class citizens by introducing an explicit set of paths in the data model, each path having its own set of labels and properties.

Practical graph data models Eclipse Modelling Framework (EMF) EClass represents classes. Identified by their name and may have several attributes and references. Can refer to supertype classes. EAttribute represents attributes which contain data elements of a class (contain data type) EDataType represents simple data types that are treated as atomic EReference represents unidirectional edge between EClasses identified by their name. LowerBound and UpperBound can be specified

Property graphs are common in graph databases. No or optional support provided for metamodelling features.

Semantic Graphs (RDF) Provides metadata data model for semantic web applications. Makes a statement about resources (nodes/objects) in the form of triples. A triple is composed of a subject, a predicate and an object. Both ontology (metamodel) and facts (instance model) are represented as triples and stored together in the knowledge base. Specialised databases for triplestores Semantic graphs can be modelled as labelled graphs.

Relation model RDBMSs dominating Database schema with attribute types and primary and foreign key relations and integrity constraints.

Schema inference algorithm. Lack of predefined schema is so prevalent that an algorithm has been designed to work around this problem by extracting the relevant part of the schema from the query specifications.

Nodes and edges are mapped to tuples Nodes are represented by 1-tuples Edges are represented by triple (source node, edge, target node) Often useful to specify the attribute names for each tuple

Unary operators Definition 8 (projection) π filters the attributes (columns) of a relation by only keeping a certain set of them. Can also be extended to perform computations and produce new attributes. Definition 9 (selection) σ filters the relation according to some criteria. $t = \sigma_\theta(r)$ where θ is a propositional formula. Selects all tuples in r for which holds. Definition 10 (duplicate elimination) δ eliminates duplicate tuples in a bag.

Binary operators Definition 11 (union) produces the set union of the tuples in the relations Definition 12 (bag union) bag union of two relations produces the bag (multiset) union of the tuples in the relations (uniqueness not required). Definition 13 (set minus) On two relations, removes the tuples present in the second relation from the first relation. Definition 14 (Cartesian product) $t = rxs$, where t holds all tuples that are concatenations of exactly one tuple from r and exactly one tuple from s.

$$r \times s = (r1, ..., r_m, s1, ..., s_n) | (r1, ..., r_m) \in r \wedge (s1, ..., s_n) \in s \quad (1)$$

Definition 15 (join or natural join) Producing the cartesian product of the relations, then filtering those tuples which are equal on the attributes that share a common name. When a tuple appears in both, we only keep a single one. Definition 16 (semijoin) takes its left input and only keeps tuples which have a matching pair on its right input in a join. Definition 17 (antijoin) Collects tuples from the left relation r which have no matching pair in the right relation s Definition 18 (left outer join) Produces the join of its input relations, then adds tuples from the left relation that did not have a pair in the right relation and pads them with NULL values.

Definition 19 (theta-joins) Performs the join according to a predicate θ a propositional formula

Definition 20 (theta-antijoin)

Graph pattern matching against a graph instance is widely known as the graph isomorphism problem. Isomorphism-based semantics

constrains some kinds of repetitions: No repeated node semantics → fully isomorphic matching and identical to restrictions used in the graph isomorphism problem No repeated edge semantics is known as edge-isomorphic matching Homomorphism based semantics defining no constraints on repetitions

Definition 21 (walk) sequence of nodes and edges, with both endpoints of an edge appearing adjacent to it in the sequence Definition 22 (trail) a walk with no repetitions of edges Definition 23 (path (strict) or simple path) a trail with no repetition of nodes. Definition 24 (path (relaxed)) same as walk Definition 25 (shortest path) consists of a walk with the least amount of possible edges between two nodes Can also be weighted if the edges contain weights Definition 26 (distance) The length of a shortest path between two nodes Definition 27 (cycle or circle) a walk that starts and ends in the same node

Graph queries: queries that contain graph patterns and / or path expressions. Recursive queries: queries that reference themselves and are evaluated up to reaching a fixpoint. Regular path queries (RPQs) are a restricted category of recursive queries that define paths using regular expressions. Allowing a concise formulation of graph queries, however little support so far. Path unwinding: allow users to define additional operations over a path in the graph

Support for WITH RECURSIVE was added in SQL:1999

Graph Query languages Cypher Supported by Neo4j. Making graph patterns with ascii-style Supports paths, transitive reachability or shortest paths. Supports path unwinding

PGQL Oracle labs Combines SQL with graph pattern matching using Cypher-like syntax. Reachability and path finding queries are supported.

G-CORE Designed to meet two key goals Allow composition of queries → queries return graphs as their results Treat paths as first class citizens → defined over the path property graph data model. Supports RPQs

SPARQL For RDF Supports variants of RPQs with property paths allowing sequences, alternatives, inversions, and transitive edges. Only endpoints can be accessed.

VIATRA Query Language Defining graph patterns based on Datalog Supports recursive queries in the form of recursive pattern calls.

Two main categories of graph processing: Graph query workloads: typically run graph pattern matching and navigation operations on semantic or property graphs. Falls in different categories depending on the scope of the graph query Local graph queries (For example on the neighbourhood of a node), more like OLTP style database systems Global graph queries (OLAP) touch on a significant portion of the data, performing more complex aggregations on the data. Graph analysis workloads: define analytical computations on graphs with little to no information stored as attributes Used to derive graph metrics, such as centrality, clustering and connectivity

3.10 Cypher: An Evolving Query Language for Property Graphs

Cyper property graph query language originally designed as part of Neo4j graph database [**?**]. Used in domains such as master data and knowledge management, recommendation engines, fraud detection, IT operations and network management, authorization and access control, bioinformatics, social networks, software system analysis and investigative journalism. Advantage of explicit support for graph data In this paper Cypher 9 is discussed Data model → property graphs, consisting of nodes and relationships (edges) The language has a fully formalised semantics of its core constructs. Given the (at the time) lack of standard for the language, there was a need to provide this.

Linear queries: Input is a property graph, output is a table. Instead of a SELECT statement at the beginning, there is a RETURN statement at the end of a query. Queries can be merged together using a WITH statement, where the query before the WITH is the driving table for the query after the WITH. UNION is also supported.

Pattern matching is done with ASCII art. Whatever binds to the pattern is introduced as a new row. The patterns express a restricted form of regular path queries: the concatenation and disjunction of single relationship types as well as variable length paths.

Data can also be modified, using CREATE, DELETE and SET. A MERGE can also be performed using a pattern. Overall the language is fairly similar to SQL, to allow users to switch easily between the two languages.

Query execution follows a conventional model, outlined by Volcano Optimizer Generator. Query planning based on the IDP algorithm using a cost model. Final query compilation uses either a simple tuple-at-a-time-iterator-based execution model, or compiles the query to Java bytecode with a push-based execution model.

Execution plan contains largely the same operators as RD engines, with an additional expand operator which finds pairs of nodes connected through an edge.

Several example queries are provided to create an understanding of the structure and execution of the queries.

A formal specification is provided for the core of Cypher. Key elements: Data model, including values, graphs and tables Query language, including expressions, patterns, clauses and queries Values can be simple or composite, such as lists or maps. Expressions denote values, patterns occur in MATCH statements. Queries are sequences of clauses. Each clause denotes a function from tables to tables.

A detailed description is provided of the values, the property graphs and tables. For paths of a set length, only one mapping exists of the free variables in which the path satisfies the pattern. In variable length paths this need not be the case. With rigid patterns, Cypher only looks in which relationship ids occur at most once. It cannot traverse the same relation more than once. Does not allow repeating relationship edges while matching patterns to avoid infinite paths.

A detailed description on the semantics of clauses is provided

Limitation is that it does not support multiple graphs (disconnected) Named queries are supported in Cypher 10 Temporal types (like DateTime) are currently being developed It is schema optional

3.11 G-CORE A Core for Future Graph Query Languages

G-CORE is a combined effort of academia and industry constructing the future of graph query languages [**?**]. Its main ideas are that the language should be composable, meaning that graphs are both the input and output of queries. Secondly, paths should be treated as

first-class citizens. LDBC has designed G-CORE, as well as various graph data management benchmarks. Graph databases are useful in a wide variety of fields. Three main challenges are identified about existing graph query languages Composability: Current implementations output tables of values, nodes or edges, while outputting graphs should make it easier for the user. Paths as first-class citizens: Providing the ability to return paths to the user enables post-processing paths within the query languages rather than in an ad-hoc manner Capturing the core of available languages: Since no standard exists, current languages should be taken into account and what they do well in order to create a standard. The following contributions are presented: Path Property Graph Model: the property graph is extended with paths. Paths can also have labels and kv pairs. Syntax and semantics of G-CORE Complexity results Every query returns a graph. Complexity analysis: each query q over input PPG G can be computed in polynomial time. G-CORE and SQL can be combined if instead of CONSTRUCT, SELECT is used and using FROM <table>

3.12 Knowledge Graphs

The paper by Hogan et al. [**?**] provides an in-depth description of the term "knowledge graph". Knowledge graphs are a graph based data model to capture knowledge in application scenarios that capture knowledge from a diverse source of data at a large scale. Graphs provide concise and intuitive abstraction over relational model. Allow postponing a schema. Graph query languages are better suited to the data (finding paths of arbitrary number of edges) Knowledge graphs aim to portray information of the real world. Where nodes imply subjects of interest, and edges possible relations between nodes. The standard data model used on directed edge-labelled graphs is Resource Description Framework (RDF). Within an RDF there are three types of nodes Internationalised Resource Identifiers (IRIs): Used for globally identifying entities and relations on the web. Literals: used to represent strings and other data type values Blank nodes: Used to denote the existence of an entity Heterogeneous Graphs: each node and edge is assigned one type. Edge is called homogeneous if between two nodes of the same type, otherwise heterogeneous.

Graphs can be stored in various ways, directed-edge labelled graphs can be stored in rds as either a single relation of arity three (triple table), binary relation for each property (vertical partitioning) or as n-ary relations for entities of a given type (property tables)

A number of semantics have been proposed for evaluating graph patterns. Homomorphism-based semantics → multiple variables can be mapped to same term Isomorphism-based semantics → nodes or edges are mapped to unique terms. Complex graph patterns allow tabular results of one or more graph patterns to be transformed using relational algebra. Navigational graph patterns (regular path queries)

To ensure data completion a form of validation needs to be done. Shapes graphs can target a set of nodes and specify constraints on those nodes. A node conforms to the shape if it satisfies all constraints. Fuzzy RDF allow annotating edges with a degree of truth Temporal RDF allows annotating edges with time intervals.

Ontologies can help with entailment, such as x is part of y, y is part of z, thus x is part of z. The most popular Ontology is Web Ontology Language (OWL)

Additional knowledge of a graph can be gathered using inductive reasoning. For example, if we observe that almost all capitals have international airports, we reason that Santiago must have an international airport. This need not be true however, so we perhaps assign a truth value of 0.959 accuracy to this prediction. Using supervised or unsupervised models, we can then create this inductive knowledge to make the knowledge graph more complete. Graph analysis can be performed, such as finding the node centrality, community, connectivity, node similarity, and graph summarisation.

The paper goes into detail about various machine learning methods which can be used to train models to interpret and further complete the dataset. Such as Graph Neural Networks, which is a graph where nodes are connected to their neighbours in the data graph. Convolutional Neural Networks, which are especially useful for image data. A difficulty lies in the interpretation of new unseen nodes or edges. A method that can help with this is symbolic learning, which can provide more general interpretations of the observed nodes and edges. An example of this is rule mining and axiom mining.

3.13 An early look at the LDBC Social Network Benchmark's Business Intelligence workload

The paper [**?**] provides an early look at the LDBC Social Network Benchmark's Business Intelligence workload. This workload tests graph data management systems on a graph business analytics workload. At the time, OLAP-like Business Intelligence workloads on graphs had been an somewhat unexplored area. The only comparable benchmark was the Berlin SPARQL Benchmark's BI use case, however this was found to be lacking as it did not have a true graph structure.

The workload consists of 25 read queries, carefully designed around chokepoints. They are designed in such a way that they represent realistic BI operations one would perform on a social network. The following chokepoints were chosen:

(1) Path pattern reuse
(2) Cardinality estimation of transitive paths
(3) Efficient execution of a transitive step
(4) Efficient evaluation of termination criteria of transitive queries
(5) Complex patterns
(6) Complex aggregations
(7) Windowing queries
(8) Query composition
(9) Dates and times
(10) Handling paths

A detailed query discussion shows that the order in which a query is executed can have several orders of magnitude difference in the execution time.

3.14 The property graph database model

The paper [**?**] starts of with a formal definition of a graph database: "A system specifically designed for managing graph-like data

following the basic principles of database systems. A database model has three main components:

(1) A set of data structure types
(2) A set of query operators
(3) A set of integrity rules

The paper mentions how the market of graph databases is fragmented, making research on methods and techniques more difficult. An informal definition of a property graph is given as a directed labelled multigraph with the special characteristic that each node or edge could maintain a set (possibly empty) of property value pairs. A formal definition is also given which is as follows: Assume that $\boldsymbol{L}$ is an infinite set of labels, $\boldsymbol{P}$ is an infinite set of property names, $\boldsymbol{V}$ is an infinite set of atomic values, and $\boldsymbol{T}$ is a finite set of datatypes. Given a set X, we assume that $SET^+(X)$ is the set of all finite subsets of X, excluding the empty set. Given a value $v \in \boldsymbol{V}$, the function type(v) returns the data type of v.

Definition 1: A property graph is a tuple $G = (N, E, \rho, \lambda, \sigma)$ where:

(1) N is a finite set of nodes.
(2) E is a finite set of edges such taht E has no elements in common with N .
(3) $\rho : E \rightarrow (N \times N)$ is a total function that associates each edge in E with a pair of nodes in N.
(4) $\lambda : (N \cup E) \rightarrow SET^+(\boldsymbol{L})$ is a partial function that associates a node/edge with a set of labels from L.
(5) $\sigma : (N \cup E) \rightarrow SET^+(\boldsymbol{P})$ is a partial function that associates nodes/edges with properties, and for each property it assigns a set of values from $\boldsymbol{V}$.

The papers goes into integrity constraints which are general statements and rules that define the set of consistent databasee states or changes of state or both. A graph schema can help define the types or nodes or edges and properties for such types.

Definition 2: A property graph schema is a tuple $S = (T_N, T_E, \beta, \delta$ where:

(1) $T_N \subset \boldsymbol{L}$ is a finite set of labels representing node types.
(2) $T_E \subset \boldsymbol{L}$ is a finite set of labels representing edge types, satisfying that T_E and T_N are disjoint.
(3) $\beta : (T_N \cup T_E) \times \boldsymbol{P} \rightarrow \boldsymbol{T}$ is a partial function that defines the properties for node and edge types, and the datatypes of the corresponding values.
(4) $\delta : (T_N, T_N) \rightarrow SET^+(T_E)$ is a partial function that defines the edge types allowed between a given pair of node types.

Definition 3: Given a property graph $G = (N, E, \rho, \lambda, \sigma)$ and a property graph schema $S = (T_N, T_E, \beta, \delta)$ we say that G is valid with respect to S when:

(1) For each node $n \in N$, it applies that $\lambda(n) \subseteq T_N$
(2) For each edge $e \in E$, it applies that $\lambda(e) \subseteq T_E$
(3) For each node or edge property $(o, p) = v$ in G, it satisfies that there exists $\beta(t_o, p) = dt$ in S such that $t_o \in \lambda(o)$ and $type(v) = dt$
(4) For each edge $e \in E$ such that $\rho(e) = (n, n')$, it satisfies that there exists $l_e \in \delta(t, t')$ such that $l_e \in \lambda(e), t \in \lambda(n)$ and $t' \in \lambda(n')$

Additional constraints can be defined, such as mandatory and optional properties / edges, unique attributes, and cardinality constraints.

A basic query language for the graph data model defined is given. The syntax is similar to SQL, SPARQL and Cypher. The main clauses are SELECT, FROM, MATCH and WHERE. Additionally, UNMATCH and UNION are also defined to support negation and union of graph patterns. Some examples of the semantics are given. A formal definition as given as in this paper can help standardize the graph query language, though the one given in this paper is still limited and would need to be further expanded upon.

3.15 Declarative and distributed graph analytics with GRADOOP

The paper [?] identifies two major categories of systmes that focus on the management and analysis of graph data: graph database systems and distributed graph processing systems. The first one mainly focusing on efficient storage and transactional processing with a provided declarative graph query language. They are however less suited for high-volume data analysis and graph mining. Graph processing systems, such as Google Pregel or GraphX are able to do this, however they lack expressive graph data models with support for heterogeneous entities and declarative graph operations. To combine the strength of both, GRADOOP was built. Its main contributions are:

- Extended Property Graph Model
- Composable graph operators
- Integration of graph algorithms
- Support for batch-oriented program execution.

A brief explanation of its core components is given, namely:

- Distributed Storage
- Distributed Execution Engine
- Extended Property Graph Model
- Graph Analytical Language
- Application Programming Interface
- Performance Evaluation

The extended property graph model does not force any scheme, however it is not clear why this is called 'Extended'. Graphs and vertices and edges are first-class citizens and can have their own properties.

A demonstration of the system is provided, however no real results are presented, making it difficult to estimate the performance of GRADOOP for now.

3.16 Two for One – Querying Property Graph Databases using SPARQL via Gremlinator

This paper [?] identifies that there is a lacking basic interoperability with the two query language of the data models most commonly used for Knowledge graphs. These query languages are SPARQL for RDF data models, and Gremlin, a Property Graph traversal language. The paper presents Gremlinator, which acts as a translator between SPARQL and Gremlin. This should increase the interoperability between the expressive data modelling that is RDF and efficient graph traversals that are possible with Property graphs. It claims the following advantages:

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

(2) It provides the foundation for a hybrid use of RDF triple stores and property graph store.

The paper mentions some related work, mainly regarding the work done on creating translating queries between SPARQL and SQL. It also mentions Cypher, the most well known graph query language of the Neo4j graph database.

Gremlinator supports both pattern matching and graph traversal queries and claims to be general than Cypher since it provides a common execution platform that supports any graph computing system. The execution pipeline is briefly discussed, though not very detailed it provides a good insight into the functionalities of the system. After this the paper addresses a limitation that Gremlinator currently does not support variables for the property predicate, though perhaps more limitations could have been discussed. A demonstration of 30 pre-defined SPARQL queries is provided for reference, the results seem promising, though not a lot of details are provided.

3.17 Efficient Batched Graph Analytics Through Algorithmic Transformation: Chapter 3

This chapter of the paper written by Then [**?**] introduces the Multi-Source breadth-first search (MS-BFS). It is based on the breadth-first search (BFS) which is able to traverse a graph from a given starting node. Due to the size of graphs, it can be computationally expensive to use a single instance of BFS. Optimizations, such as parallelization, have been researched to improve the performance of BFS, however this comes with additional complexity. MS-BFS is able to process a large number of BFSs concurrently in a single core. It exploits the properties of small-world networks, which implies that the distance between any two nodes is small compared to the size of the graph.

We will first provide an introduction to the BFS algorithm, as that is what MS-BFS is based upon. For BFS there are two main states for every node, discovered and explored. A discovered node has already been visited by the BFS, and explored means that both the node, its and edges and its neighbours have been visited. Pseudocode is provided in Listing 1 for reference.

The goal of MS-BFS is to create multiple independent BFSs on the same graph. This does however come with a number of issues and criteria:

- Memory locality worsens since the same node needs to be discovered by multiple BFSs leading to a higher number of cache misses
- Scalability should be properly supported to allow for a increased number of cores.
- Synchronization costs should be avoided as much as possible

An important observation from running multiple BFSs is that every node is discovered multiple times. Every time a node is explored, its set of neighbours must be traversed to check if they have already been discovered. This leads to a potential large amount of cache misses. To reduce these cache misses, MS-BFS shares the exploration of these nodes. This is important due to the small-world property mentioned previously, this makes that a large amount of nodes can be explored. When they are explored they are shared among the multiple BFSs that are ongoing.

Algorithm 1 BFS

```
1:  Input: G, s
2:  seen ← {s}
3:  visit ← {s}
4:  visitNext ← ∅
5:  while visit ≠ ∅ do
6:      for each v ∈ visit do
7:          for each n ∈ neighbours_v do
8:              if n ∉ seen then
9:                  seen ← seen ∪ {n}
10:                 visitNext ← visitNext ∪ {n}
11:                 Do BFS computation on n
12:             end if
13:         end for
14:     end for
15:     visit ← visitNext
16:     visitNext ← ∅
17: end while
```

Algorithm 2 MS-BFS

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

Further optimization can be applied to this algorithm, such as using bit operations to improve memory locality and avoid expensive set operations. Another optimization is to apply the direction-optimized BFS technique. During a typical execution, new nodes are discovered from the nodes found during the previous level, this is referred to as a top-down approach. However, a bottom-up approach, which uses the *seen* array of nodes that have not yet been discovered and looks for the discovered nodes, essentially working in the opposite direction of a top-down approach.

The performance of the MS-BFS algorithm was analysed using the LDBC Social Network dataset and a real-world dataset of Wikipedia and Twitter. The results show that MS-BFS has excellent scalability.

Furthermore, compared to the standard BFS algorithm it was shown that MS-BFS is roughly 80x faster on all datasets.

3.18 Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries

This paper by Besta et al. [**?**] provides a survey and taxonomy of graph database systems. In the paper, 45 graph database systems are presented.

The paper starts with a discussion on the classes of systems capable of storing and processing dynamic graphs. This includes NoSQL, which the Graph database Neo4j uses, but also wide-column stores, document stores and general key-value stores.

An explanation of the various types of graph models used by the surveyed systems is given. The simplest is a the *Simple Graph Model*, which is only capable of handling nodes and (weighted) edges that are either directed or undirected.

There is also the hypergraph model, which is able to connect an arbitrary amount of nodes to an edge.

A more rich version of the *Simple Graph Model* is the *Labeled Property Graph Model* also referred to as property graph. In this model the edges and nodes can be augmented with labels to define classes. In addition, a node or edge can contain any number of properties. These properties are stored as a key-value pair. Variants of the property graph exist, which for example limit the amount of labels an edge or node can have.

Finally, the Resource Description Framework (RDF) is another popular model to represent information. The model consists of a collection of *triples*. Each triple consists of a subject, predicate and object.

Ways exist to transform the property graph into RDF and vice versa.

The graph database systems were categorized on a number of categories:

- General backend
- Supported graph data models and representations
- Data organization
- Data distribution
- Query execution, transaction types, and supported query languages

A point worth mentioning is that almost all systems are closed source or do not provide all the details to their system, making detailed comparisons more difficult. Three large tables are provided to the reader indicating the level of support on the categories mentioned previously. The table regarding the supported query languages show how fragmented the field is. There are six different query languages, SPARQL, Gremlin, Cypher, SQL, GraphQL, and Progr. API (formulating queries using a native programming language such as C++). While most of these languages are fairly similar, most systems only support one. Some systems do not even support one of the six mentioned here, but support some different language.

A more in-depth analysis is provided for the different backend categories. For this we refer the reader to

3.19 GrainDB: A Relational-core Graph-Relational DBMS

This paper presents GRainDB [**?**], a system extending the RDBMS DuckDB by providing graph modeling, querying, and visualization capabilities. The authors modified the internals of DuckDB to integrate storage and query processing techniques to make the workload on graphs more efficient. These modifications include: predefined pointer-based joins, hybrid graph-relational data modeling and querying, and graph visualization. It appears to be a risky strategy to modify the internals of DuckDB, as this is still a relatively new database system in which the internals are still prone to change. By making modifications it could be that future versions of DuckDB become incompatible with the modified version and updating become difficult.

The team identifies advantages for both relations and graphs in terms of the data model. Relations are able to represent n-ary relationships for arbitrary values of n, and relations provide a natural structure to model normalized data. On the other hand, graph-based models are often closer to the mental model of users and are semi-structured.

However, the most popular query language, SQL, is considered very cumbersome to use for graph-based queries. Most notably, recursive queries are difficult to write and understand.

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

3.20 The case against specialized graph analytics engines

This paper by J. Fan et al. [**?**] provide an answer to the question whether RDBMSs should be used for graph analytics instead of creating specialized graph engines. The authors present Graph Analysis in Legacy (Grail) acting as a syntactic mapping layer for any queries. The authors compare Grail to two specialized graph engines, GraphLab and Giraph. Three algorithms are tested on these engines, namely Single Source Shortest Path (SSSP), PageRank, and Weakly Connected Component. A detailed description of Grail is provided including the API, the Translator, and the Optimizer. In addition, the computation model, the way nodes communicate with each other, of Grail is compared to those of Giraph and GraphLab. Some results from empirical evaluation are presented, obtained using datasets of various sizes. The sizes range from 9K nodes and 5M edges to 100M nodes and 3.3B edges. The Grail implementation appears slower on the smaller dataset, however is able to scale better compared to Giraph and GraphLab.

3.21 PGQL: A Property Graph Query Language

van Rest et al. propose[**?**] a new query language for the Property Graph data model: Property Graph Query Language (PGQL). Given the rise in popularity for graph technology, there is also a need for a well-designed query language for graphs. The authors argue that this query language should be able to support more graph-like queries, such as reachability, path finding, and path constructions. They mention the existence of SPARQL, which is based on the RDF

data model. Though the PG model is seen as a more natural data model. For the PG model Cypher exists, however Cypher is missing key query functionalities such as regular path queries and graph construction. Therefore PGQL was developed, which has a SQL-like syntax and functionality, with the addition of the aformentioned functionalities missing in Cypher. PGQL support two different pattern matching semantics: isomorphism and homomorphism. With isomorphism, two queries are not allowed to map to the same node or edge. With homomorphism this is allowed to happen. As previously mentioned, PGQL has SQL-like syntax and functionalities. For example, it allows for grouping and aggregation, order by, limit, union, not exists and even subqueries. These are supported since the result of a (sub-)query is also represented as table with variables and their bindings, thus they can be used in another query. In addition, many existing query language support Regular Path Queries (RPQs), finding a path that satisfies a certain regular expression. PGQL has extended this by adding support for general expressions over vertices and edges along paths. PGQL supports reachability queries as well as path finding queries. In PGQL it is also possible to create graph construction queries. These queries are able to extract entities from an input graph and enhance them with additional nodes, edges, or properties.

3.22 The ubiquity of large graphs and surprising challenges of graph processing: extended survey

Sahu et al. [**?**] mention a noticeable increase in the work on graph processing in research and practice. However, little is known how the graph data is actually used in practice and the challenges faced by users. Therefore a survey was conducted among 89 users, in which the goal was to provide answers to the 4-high level questions:

- What types of graph data do users have?
- What computations do users run on their graphs?
- Which software do users use to perform their computations?
- What are the major challenges users face when processing their graph data?

They find that the graphs represent a wide variety of entities. There is a ubiquity of very large graphs. These are often more than a billion edges. The scalability forms a pressing challenge. Visualization is a popular task in the processing pipeline. RDBMSs still play an important role in managing and processing graphs.