Graph Pattern Matching in GQL and SQL/PGQ

ALIN DEUTSCH, UCSD & TigerGraph, USA

NADIME FRANCIS, LIGM, U Eiffel, CNRS, France

ALASTAIR GREEN, LDBC & Birkbeck, UK

KEITH HARE, JCC Consulting & Neo4j, USA

BEI LI, Google, USA

LEONID LIBKIN, U Edinburgh & ENS-Paris & Neo4j, UK & France

TOBIAS LINDAAKER, DataStax, Sweden

VICTOR MARSAULT, LIGM, U Eiffel, ENPC, CNRS, France

WIM MARTENS, University of Bayreuth, Germany

JAN MICHELS, Oracle, USA

FILIP MURLAK, U Warsaw, Poland

STEFAN PLANTIKOW, Neo4j, Germany

PETRA SELMER, Neo4j, UK

HANNES VOIGT, Neo4j, Germany

OSKAR VAN REST, Oracle, USA

DOMAGOJ VRGOČ, PUC Chile & IMFD, Chile

MINGXI WU, TigerGraph, USA

FRED ZEMKE, Oracle, USA

As graph databases become widespread, JTC1—the committee in joint charge of information technology standards for the International Organization for Standardization (ISO), and International Electrotechnical Commission (IEC)—has approved a project to create GQL, a standard property graph query language. This complements a project to extend SQL with a new part, SQL/PGQ, which specifies how to define graph views over an SQL tabular schema, and to run read-only queries against them.

Both projects have been assigned to the ISO/IEC JTC1 SC32 working group for Database Languages, WG3, which continues to maintain and enhance SQL as a whole. This common responsibility helps enforce a policy that the identical core of both PGQ and GQL is a graph pattern matching sub-language, here termed GPML.

The WG3 design process is also analyzed by an academic working group, part of the Linked Data Benchmark Council (LDBC), whose task is to produce a formal semantics of these graph data languages, which complements their standard specifications.

This paper, written by members of WG3 and LDBC, presents the key elements of the GPML of SQL/PGQ and GQL in advance of the publication of these new standards.

1 INTRODUCTION

Graph databases are becoming widely used [46]. Many applications require graph-structured data for analysis: they are found in a variety of scientific domains, such as biology and chemistry[29]. Machine learning increasingly centres on graph networks [26, 33]. Business transaction records, and derived metrics and control data, are viewed as graphs to detect fraud patterns, understand market trends and customer behaviours, and to model data lineage and complex access control rules.

Graphs appeal to many kinds of users because the graph data model is akin to conceptual models, which reflect our intuitive view of information as the properties of entities and their relationships. Human knowledge, when gathered incrementally as linked data items, naturally falls into the shape of a graph: this is visible in open-source knowledge graphs, such as Wikidata and DBpedia, and in the internal knowledge bases that big tech companies build in order to inform their decision and service-oriented tasks.

Two of the most popular ways of representing graphs are the *Resource Description Framework* (RDF) [18], and *property graphs* [3]. In RDF, the data is modelled as a directed edge-labelled graph. On the other hand, property graph models the data as a mixed (that is, partially directed) multigraph. Both nodes and edges can be labelled, and also attributed (that is, associated with property/value pairs). In this paper we focus on property graphs.

The property graph data model has gained adoption in many commercial database systems. Examples include AgensGraph, Amazon Neptune, ArangoDB, CosmosDB, DataStax Enterprise Graph, HANA Graph, IBM Db2 Graph, JanusGraph, Neo4j, Oracle Server and PGX, RedisGraph, Sparksee, Stardog, and TigerGraph. Unlike RDF with its query language SPARQL [27], which is a W3C standard, property graph systems possess disparate storage models and querying facilities, present either in the form of APIs like Gremlin, or, since 2010, in the form of declarative graph query languages including Cypher from Neo4j [23], GSQL from TigerGraph [47], and PGQL from Oracle [30], as well as industry/academia prototypes such as G-CORE [2]. Pre-existing property graph data languages overlap substantially in core features, but each has also introduced unique innovations. The 2015 openCypher project has led to a widening industrial use of Cypher as a language for property graphs, but did not succeed on its own in establishing a standard.

As graph data management has expanded as a product category, the lack of a standard query language (and associated sub-languages for schema definition) for property graphs has been felt more strongly. The situation resembles the early days of the relational/tabular data model, which led to the standardization of SQL, a hugely successful lingua franca. In 2019 the Joint Technical Committee 1 of ISO/IEC, which defines information technology standards for the International Organization for Standardization, and International Electrotechnical Commission, approved a project to create GQL, a standard property graph query language with full CRUD (create/read/update/delete) and catalog capability. GQL builds on prior graph languages, as well as a new part 16 of SQL, in development since 2017, called SQL/PGQ. PGQ (short for *property graph queries*) specifies how to define graph views over an SQL tabular schema, and to run read-only queries over such views, that can be projected by an SQL SELECT statement.

GQL and SQL/PGQ share a common data model, and a common *graph pattern matching language*, called *GPML* throughout the paper. Both language projects have been assigned to the ISO/IEC JTC1 SC32 (Sub-committee 32) Working Group for Database Languages (WG3) which continues to be responsible for maintaining and enhancing SQL as a whole. This structure serves a policy that GPML be kept identical in GQL and SQL/PGQ. The common GPML sublanguage is the central subject of this paper.

Graph query languages extract data from a graph that matches a graph pattern. In GPML a graph pattern is a collection of path patterns, which when applied to a property graph, results in a set of *path bindings*, each mapping variables in the expression to graph elements (node and edges) forming a path in the graph. These *variable bindings* can be used to refer to the graph elements and the values of their properties. The way in which path bindings are projected to produce query results varies between SQL/PGQ and GQL, but the set of path bindings matching a path pattern for a given graph (prior to result projection) will be the same for both languages.

The standards process for both languages is governed by WG3, whose expert members represent the national standards bodies of China, Denmark, Finland, Germany, Japan, Korea, the Netherlands, Sweden, the UK, and the USA. In addition WG3 has a liaison relationship with Linked Data Benchmark Council (LDBC). LDBC is a consortium of industrial companies, research institutes, academic researchers and consultants. LDBC defines benchmark standards for graph data workloads, but also supports the work of WG3 on GQL and SQL/PGQ by adding the expertise of its members to suggest improvements, or new ideas that may ultimately be incorporated in the official standards (for example, the Property Graph Schema Working Group in LDBC). As part of this process, an academic group was created under the auspices of LDBC, called FSWG (Formal Semantics Working Group). Members of this group previously provided a complete formal semantics of Cypher features [23]. The group's goal was to scrutinize and comment on the design of the GPML sublanguage, and to suggest improvements.

This paper gives an accessible summary of the GPML of GQL and SQL/PGQ. It does so before these two standards are published, and before vendors (including those which are represented in the author list) have released implementations of the features of GPML—but at an advanced stage of consensus on its scope and design.

The paper is organized as follows. In Section 2 we review the property graph model and in Section 3 we give a brief overview of pattern matching facilities in existing languages and survey related work. Section 4 describes the main functionalities of the GPML of GQL and SQL/PGQ. In Section 5 we explain how the language ensures finiteness of outputs, as there could be infinitely many cyclic paths in a graph. Section 6 provides a detailed explanation of the pattern-matching algorithm by means of an example. Section 7 describes future plans in terms of the Standard release, industrial implementations, and new research questions that the GPML poses.
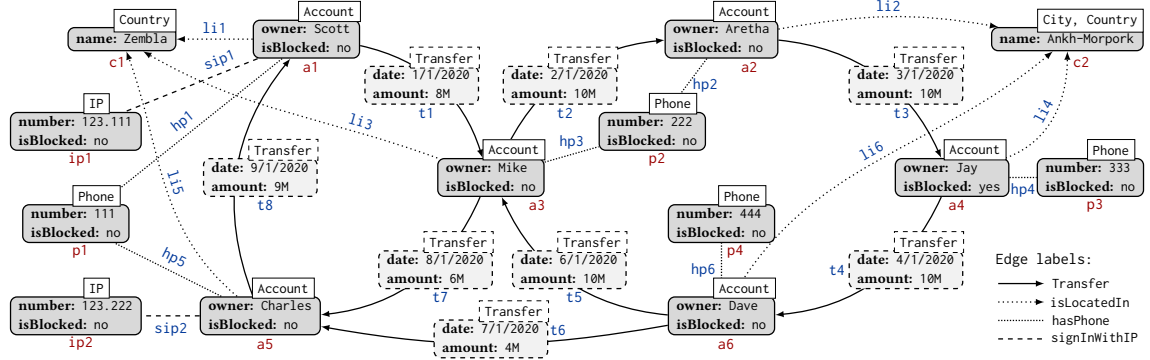
Fig. 1. A property graph with information on bank accounts, their location, and financial transations.

2 PROPERTY GRAPHS

To describe property graphs, we use a common application scenario involving banking and financial trans-
actions (used, e.g., for fraud detection). Figure 1 is an example of a property graph containing informa-
tion about bank accounts, their location, their associated phone numbers and IP addresses, and financial
transactions between them. The graph contains *nodes*, which are connected by *edges*. Nodes and edges
are identified by *node identifiers* (a1, ..., a6, c1, c2, p1,...,p4, ip1, ip2) and *edge identifiers* (t1,...,t8,
li1,...,li6,hp1,...,hp6,sip1,sip2), respectively. Furthermore, both nodes and edges can carry *labels* (e.g.,
Account, City, Transfer, isLocatedIn) and *property/value pairs* (e.g., owner/Mike, isBlocked/no). In figures,
we will depict node information in solid boxes, and edge information in dashed boxes. We use the umbrella
term *element* to refer to nodes or edges. When clear from the context, we do not distinguish between node
identifiers and nodes (similarly for edges).

In graph-theoretic literature, graphs are usually defined as pairs $\langle V, E \rangle$ of vertices V and edges E which
are either two-element subsets of V for undirected graphs, or pairs of vertices for directed graphs. In con-
trast, property graphs are *multigraphs* (there can be multiple edges between two endpoint nodes); *pseudo-
graphs* (there can be an edge looping from a node to itself); they are *mixed*, or *partially directed*: an edge can
be *undirected*, or can have source and target nodes, in which case it is *directed* from the source to the target.
They are also *attributed*: graph elements can have attributes (a disjoint union of labels and properties).

This is summarized in the following definition. Assume that $\mathcal{L}$, $\mathcal{P}$, and Val are countably infinite sets,
containing *labels*, *property names*, and *property values*, respectively.

Definition 2.1 (Property Graph). A *property graph* is defined as a tuple $G = (N, E, \rho, \lambda, \pi)$ where:

- N is a finite set of node identifiers;
- E is a finite set of edge identifiers; such that $N \cap E = \emptyset$;
- $\rho : E \rightarrow (N \times N) \cup \{\{u, v\} \mid u, v \in N\}$ is a total function mapping edges to ordered or unordered
  pairs of nodes;

Account		
ID	owner	isBlocked
a1	Scott	no
a2	Aretha	no
a3	Mike	no
…		

Transfer				
ID	A_ID1	A_ID2	date	amount
t1	a1	a3	1/1/2020	8M
t2	a3	a2	2/1/2020	10M
t3	a2	a4	3/1/2020	10M
…				

signInWithIP		
ID	A_ID	s_ID
sip1	a1	ip1
sip2	a5	ip2

Country	
ID	name
c1	Zembla

CityCountry	
ID	name
c2	Ankh-Morpork

Fig. 2. Excerpts of five tables of the tabular representation of the property graph in Figure 1. The tables Transfer and signInWithIP represent edges, the others represent nodes.

- $\lambda : (N \cup E) \to 2^{\mathcal{L}}$ is a total function mapping node and edge identifiers to sets of labels (including the empty set);
- $\pi : (N \cup E) \times \mathcal{P} \rightharpoonup \text{Val}$ is a partial function mapping elements and property names to property values.

We call an edge e *directed* if $\rho(e) \in N \times N$ and *undirected* if $\rho(e) \in \{\{u, v\} \mid u, v \in N\}$. In both cases we say that e connects u and v. Notice that both directed and undirected edges allow $u = v$, in which case the edge is a *self-loop*. Furthermore, the definition allows both directed and undirected edges to carry labels and data. Indeed, λ and π can be defined for directed and undirected edges. The definition does not preclude having two different edges connecting the same nodes. For more details we refer the reader to [24].

A property graph has a *graph representation*, which is illustrated in Figure 1, but it also has a *tabular representation*, which is illustrated in Figure 2. In the graph representation, we depicted the nodes in red and the edges in blue. We depicted the properties and values associated to nodes in grey rounded rectangles (which are dashed for edges) and most of the labels in white rectangles. To avoid clutter however, we have omitted the labels on some edges (namely `isLocatedIn`, `hasPhone`, and `signInWithIP`). The tabular representation has a relation for every combination of labels that appears on some node or edge in the graph. For instance, every label that appears in Figure 1 is a relation name in the tabular representation, except `City`, which does not appear by itself. It does appear together with Country (on node c2), so the tabular representation has a relation named CityCountry, which contains node c2.

A *path* is an alternating sequence of nodes and edges such that: (1) it starts and ends with a node; and (2) subsequent nodes in the sequence are connected by the edge between them.[1] We write paths as path(c1,li1,a1,t1,a3,hp3,p2) indicating a path from node c1 to node p2 that first follows edge li1 in reverse direction, then the edge t1 in forward direction, and then the undirected edge hp3.

---

[1]As is usual in the graph database literature [8, 38, 40, 49], we use the term path to denote what is called *walk* in the graph theory literature [12].
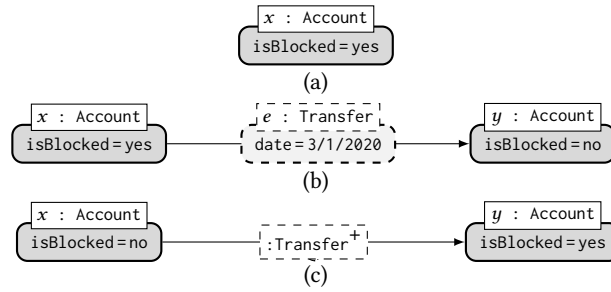
Fig. 3. Node pattern, edge pattern, and a path pattern.

3   GRAPH PATTERN MATCHING TODAY

As identified in a recent survey of modern graph query languages [3], when querying graph databases, one usually starts by selecting atomic graph elements, such as nodes, edges, or paths. Coming back to our example from Figure 1, basic elements we might want to select are, for instance:

- All the nodes representing a blocked account;
- All the edges representing a money transfer from a blocked to a non-blocked account that occurred on a specific date;
- All the paths whose edges represent money transfers, and that start in a non-blocked account, but end in a blocked one.

In an abstract sense, we could represent these elements as patterns shown in Figure 3. In addition to standard elements of a property graph, these graph patterns also attach variables x, y, e, etc. to different parts of the patterns. To match such a pattern in a property graph, we need a mapping that links the node patterns to nodes in the graph, and similarly with edge and path patterns. For instance, the pattern (a) from Figure 3 matches to an Account node such that the value of the property isBlocked is set to yes. Similarly, the pattern (b) of Figure 3 matches the node variables x and y to two accounts (the first one blocked, and the second one not), while e matches an edge of type Transfer connecting these two nodes, with the value of date set to 3/1/2020.

The most interesting pattern, shown in Figure 3(c), specifies a path of arbitrary length. Starting from [37], specifying paths via regular expressions over edge types has been well established both in the research community [3, 8, 15, 49] and in the industry [23, 30, 42, 47]. Several ways of defining valid paths (simple, arbitrary, etc.) have been proposed [23, 38]; we discuss them in later sections.

With these basic building blocks at hand, we can define more complex graph patterns. For instance, assume that in the property graph from Figure 1, we want to identify all pairs of owners with accounts that are located in Ankh-Morpork, and are connected by a path of (arbitrarily many) money transfers. Additionally, we might want to stipulate that the first account is blocked, while the second one is not blocked. Following our graphical representation, one can visualize this query as the graph pattern in Figure
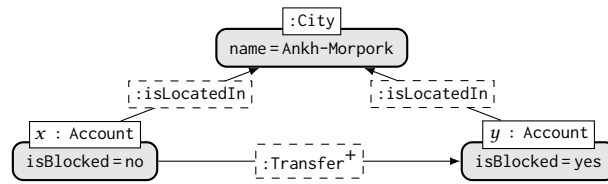
Fig. 4. Pattern of fraudulent accounts in Ankh-Morpork.

4. Notice that the pattern only specifies what we want to match in the graph, not what we want to output (the owners of x and y).

In general, a graph pattern specifies a set of nodes and their connections via edges and paths, filtered according to labels and values of some of their properties. Such a graph-shaped query gets matched to the graph database to obtain a (collection of) result(s). Graph patterns form the core of most existing graph query languages, and have been present in academic research (under the name of conjunctive regular path queries, or CRPQs [8, 14, 49]), in W3C recommendations (e.g. SPARQL [42]), and in existing property graph engines [23, 30, 40, 47]. We next summarize the support for graph patterns in SPARQL, Neo4j's Cypher, Oracle PGQL, and TigerGraph's GSQL. For historical reasons, we begin with SPARQL.

The SPARQL standard [42] prescribes graph pattern matching on edge-labeled graphs represented as a single ternary relation. Neither labels nor properties are present in SPARQL, and matching is done only between nodes and constants. Path queries are supported by means of property paths [27] that permit full 2RPQs [15] under arbitrary path semantics. In order to avoid infinite results when cycles are present, SPARQL prescribes that one can only check for the existence of a path between a pair of nodes, but cannot count the number of such paths, or try to reconstruct them [6, 32]. A much simplified version of the query from Figure 4 in SPARQL (assuming cities are identified by their names), would be the following:

```
SELECT ?x, ?y
WHERE { ?x isLocatedIn "Ankh-Morpork".
        ?y isLocatedIn "Ankh-Morpork".
        ?x Transfer+ ?y }
```

The most common query language for property graphs is Cypher [23] that originated at Neo4j and has since been implemented by vendors such as Amazon, Agens Graph, Katana Graph, Memgraph, RedisGraph, and SAP HANA [41]. The owners of the accounts in the query from Figure 4 can be found with Cypher as follows:

```
MATCH (a:Account {isBlocked:'no'})-[:isLocatedIn]->
      (g:City {name:'Ankh-Morpork'})<-[:isLocatedIn]-
      (b:Account {isBlocked:'yes'}),
  p = (a)-[:Transfer*1..]->(b)
RETURN a.owner, b.owner
```

The query first matches the variables `a` and `b` to an unblocked and a blocked account located in Ankh-Morpork, using its "ASCII-art" syntax for specifying edges. It then matches `p` to a whole path of transfers of arbitrary length from `a` to `b`, and returns the owners. Unlike SPARQL, Cypher allows returning paths: we can return `p` simply by including it in the **RETURN** clause.

Cypher's pattern matching has multiple other features. It permits label disjunctions and inline predicates in variable-length path patterns: **MATCH (**a**)-[:**X|Y*{weight:1}**]->(**b**)**. It supports testing for the presence or absence of a path relative to an element specified in a match: **MATCH (**a:Person**)->(:**Cat**)** **WHERE NOT (**a**)->(:**Dog**)**. It also allows custom graph traversals by calling user-defined procedures via **CALL**, and complex predicates involving nested data, especially paths. Cypher also allows additional operations on matched paths such as returning a single shortest or all shortest paths, returning all nodes or edges in a path, the length of a path, as well as the number of paths found.

PGQL (Property Graph Query Language) [48] is an SQL-like graph pattern matching query language that is open-sourced by Oracle. Where features overlap, PGQL follows SQL's syntax and semantics, such as in **SELECT**, **WHERE**, **GROUP BY**, **HAVING**, and **ORDER BY**, as well as for functions, aggregations, predicates, existential and scalar subqueries. The core of a PGQL query is the graph pattern, which is spread over multiple **MATCH** clauses that are syntactically placed inside the **FROM** clause. The owners of the accounts of query in Figure 4 can be found as follows.

```
SELECT x.owner AS A, y.owner AS B
FROM
  MATCH (x:Account)-[:isLocatedIn]->
        (g:City)<-[:isLocatedIn]-(y:Account),
  MATCH ANY (x)-[e:Transfer]->+(y)
WHERE x.isBlocked='no' AND y.isBlocked='yes'
  AND g.name='Ankh-Morpork'
```

We can return the whole path of `Transfer` edges between `x` and `y` by replacing the first line of the query with **SELECT** x.owner **AS** A, y.owner **AS** B, LISTAGG(e.ID,', '), assuming that e.ID is the ID of edge e. Here, the variable `e` is treated as a *group variable* that matches subsequent edges of the path and can be used to aggregate data along paths of variable length. For example, one can compute the length of the path using COUNT(e), or filter out paths with repeated edges using **WHERE** COUNT(e) = COUNT(DISTINCT e). The aggregate LISTAGG(e.ID,', ') produces a comma-separated list of values encoded as a single string of characters. **ANY** is used to obtain an arbitrary single path from `x` to `y`. PGQL supports also **ANY SHORTEST**, **ALL SHORTEST**, **TOP** k **SHORTEST**, **ANY CHEAPEST**, and **TOP** k **CHEAPEST**, allowing for retrieving multiple paths between a pair of nodes. **ALL** gives all paths but requires an upper bound on the path length, e.g., {1,4} instead of +.

The design philosophy behind TigerGraph's GSQL (Graph SQL) language [19, 47] is to flatten the learning curve for the largest community of potential adopters, namely SQL programmers. Like PGQL, GSQL

supports SQL-style **SELECT**, **WHERE**, **GROUP BY**, **HAVING**, **LIMIT** and **ORDER BY** clauses, aggregation, and graph patterns in the **FROM** clause. The running example query from Figure 4 is expressed in GSQL as follows, using table `T` to hold the result.

```
SELECT x.owner AS A, y.owner AS B INTO T
FROM Account:x -(isLocatedIn>)- City:g
     -(<isLocatedIn)- Account:y,
     :x -(Transfer>+)- :y
WHERE x.isBlocked='no' AND y.isBlocked='yes'
  AND g.name='Ankh-Morpork'
GROUP BY A, B
```

GSQL's default semantics is **ALL SHORTEST**, hence there is no upper bound on the + quantifier. While supporting group and path variables is on the near-future roadmap for GSQL, this is merely a matter of adding syntactic sugar as they are currently expressible using a GSQL-specific aggregation paradigm based on *accumulators*. These are containers that can be attached at query time to the vertices and written to during the pattern matching phase. Accumulator inputs are aggregated via pre- or user-defined binary operators. Accumulator-based aggregation was shown to enjoy expressivity and efficiency benefits over SQL-style aggregation [20].

4 GRAPH PATTERN MATCHING LANGUAGE

Recall that we use the acronym GPML to denote the common graph pattern matching language that is shared by SQL/PGQ and GQL. In the database domain there is no standard or product by those initials to our knowledge; it is just a convenient abbreviation for this paper. This section is devoted to explaining how GPML works. In general, a GPML pattern is an expression of the form:

MATCH *Pattern*

where *Pattern* is a pattern specifying the parts of our input property graph we want to explore. The **MATCH** clause may be optionally followed by a **WHERE** clause to filter the results; we discuss it later in the section. For example, the following expression retrieves all the accounts in our graph from Figure 1 that are not blocked:

MATCH (x:Account **WHERE** x.isBlocked='no'**)**

Intuitively, this pattern, which is an example of a *node pattern*, binds to x all the nodes in the graph which have the label `Account`, and whose `isBlocked` property has the value `no`. Another basic pattern, called the *edge pattern* extracts edges from our graph. For instance, the following pattern asks for all the edges which represent transfers with a value of more than 5 million:

MATCH -[e:Transfer **WHERE** e.amount>5M**]->**

We shall use shorthands such as 5M for readability. The variable e gets bound to an edge in our graph, giving us access to its data. We later explain how one can also retrieve the endpoints of the edge. Generally, node and edge patterns retrieve graph elements – nodes and edges – and can then be combined in more complex ways to define paths and patterns present in our graph. In what follows we specify how all of this is supported in GPML. We start by expanding on our definition of node and edge patterns.

4.1 Accessing Nodes and Edges

Node patterns. The most basic query pattern allows the user to fetch nodes inside a property graph. For instance, to extract all of the nodes of the graph, the user can write the following[2]

$$\textbf{MATCH (}x\textbf{)}$$

The variable x here is called an *element variable*, since it binds to a graph element. In the case of x, it will be bound to a node in the graph, since it is placed inside the **( )** braces, which signify that we are talking about a node. For instance, if evaluated on the property graph from Section 2, this query will return bindings that map x to accounts, cities, phones, and IPs.[3]

Of course, one rarely wants to obtain all the nodes in a graph, and usually will want to restrict the obtained results in some way. A natural way to do this is to look only for nodes with a specific label. GPML does this via *label expressions*. For instance, to return nodes that correspond to accounts, one writes:

$$\textbf{MATCH (}x\texttt{:Account}\textbf{)}$$

Here the label is explicitly specified, and only nodes with the label Account will now be bound to x. More generally, label expressions allow conjunctions (&), disjunctions (|), negations (!), and grouping of individual labels by using parentheses. For instance, to capture all the nodes that are either accounts, or IP addresses, we would write **MATCH (**x:Account|IP**)**. There is also a wildcard symbol % matching any label; e.g., pattern **(**:!%**)** matches nodes that have no labels.

Further filters on a node restrict the values of some of its properties. As seen earlier, if we are interested only in those accounts that are not blocked, the pattern **(**x:Account**)** changes to **(**x:Account **WHERE** x.isBlocked='no'**)**. The **WHERE** clause can be put outside **MATCH** to be used as a postfilter for produced matches:

$$\textbf{MATCH (}x\texttt{:Account}\textbf{)}$$
$$\textbf{WHERE } \texttt{x.isBlocked='no'}$$

The **WHERE** clause can support a host of search conditions, and these may be combined into logical statements using **AND**, **OR**, and **NOT**.

---

[2]The ascii art **()** used for nodes is meant to be suggestive of how nodes are usually drawn as circles in visual representation of a graph.

[3]For now we model query evaluation as a process taking a property graph and a query pattern as input, and producing as output a multiset of *bindings* from the set of variables to the set containing graph elements and property values. In Section 6 we extend this to capture more advanced objects such as arbitrary length paths.

We note that each of the restricting elements in a node pattern (e.g. the variable, the label specification, or the **WHERE** condition) is optional. That is, any of them can be either present or absent. This makes the following the simplest possible node pattern: **MATCH ()**. Since there is no variable, there is no syntax to reference the nodes bound to this pattern; consequently it is of little use standing alone as the sole element pattern as in this example. However, below we explain how such a pattern can be combined with edge patterns in order to extract paths and patterns in the graph. Intuitively, this construct will allow us to have a placeholder for any node in the graph, thus allowing us to link it with other graph elements. It is of more use when concatenated with edge patterns, as in

$$\textbf{MATCH (}x\textbf{)-[:}\texttt{Transfer}\textbf{]->()-[:}\texttt{isLocatedIn}\textbf{]->(}y\textbf{)}$$

In the preceding example, the user is only interested in the bindings to x and y, consequently the user omitted the variables from the two edge patterns and the middle node pattern.

Edge patterns. These let us explore an edge connecting two nodes. A basic way of doing this is via the pattern[4]

$$\textbf{MATCH -[}\texttt{e}\textbf{]->}$$

which searches all the directed edges in the graph, and binds them to the element variable e. That is, the bindings generated by this pattern will map e to any identifier of a directed edge in the graph. If we wish to use specific types of edges (e.g. undirected), this can be done by special symbols used in the edge pattern. For example, all undirected edges can be recovered by **MATCH ~[**e**]~**.

The full specification of possible edge direction restrictions and their combinations is provided in Figure 5. In the figure, *spec* is an optional element of the form e:labelExpr **WHERE** condition, where labelExpr and condition are as in node patterns above; when *spec* is omitted, the abbreviated version can be used. As an example, if we want edges that are either undirected, or directed from right to left, we could write **<~[**e**]~**.

Similarly to node patterns, we can restrict which edges we wish to return by specifying their labels, or using the **WHERE** clause, as was shown earlier with the example that restricted Transfer edges to those with the amount exceeding 5 million. Both label expressions and the **WHERE** clause can use the same constructs as in the case of node patterns. Similarly to node patterns, edge patterns need not specify an element variable. This will become useful later on when defining more complex patterns.

4.2   Building Path Patterns by Concatenation

Node and edge patterns can be chained together to form a *path pattern*. The most natural way to do this is to ask for edges in the graph, together with their source and target nodes, as in:

$$\textbf{MATCH (}x\textbf{)-[}\texttt{e}\textbf{]->(}y\textbf{)}$$

---

[4]Here again we use the ascii art **-[]->** to mimic how one draws an edge in a graph.

Orientation	Edge pattern	Abbreviation
Pointing left	**<-[** *spec* **]-**	**<-**
Undirected	**~[** *spec* **]~**	**~**
Pointing right	**-[** *spec* **]->**	**->**
Left or undirected	**<~[** *spec* **]~**	**<~**
Undirected or right	**~[** *spec* **]~>**	**~>**
Left or right	**<-[** *spec* **]->**	**<->**
Left, undirected or right	**-[** *spec* **]-**	**-**

Fig. 5. Table of edge patterns.

The bindings generated by this pattern will map `e` to an edge in the property graph, and `x` and `y` to its source and target node, respectively. If we do not specify direction and write `(x)-[e]-(y)`, then each edge will be returned twice, once for each direction in which it is traversed. Edge direction, edge (un)directedness, or any filter on nodes/edges can be used as in the examples above. For instance, to ask for a source account of every transfer that reaches an account with the `owner` equal to `'Aretha'`, we write:

MATCH (`y` **WHERE** `y.owner='Aretha'`**)<-[**`e:Transfer`**]-(**`x`**)**

Even if the edge pattern is ambiguous about the orientation of `e`, we may wish to refer to this orientation in a postfilter. We can do this using the predicates `e` **IS DIRECTED**, `s` **IS SOURCE OF** `e`, and `d` **IS DESTINATION OF** `e`.

One can keep on alternating edge and node patterns to create more complex path patterns. For instance, the query

MATCH (`s`**)-[**`e`**]->(**`m`**)-[**`f`**]->(**`t`**)**

extracts all directed paths of length two in the graph. Variable `s` is bound to the id of the source node of the path, `t` is bound to the id of the target node, and `m` to the middle node. Similarly, variables `e` and `f` are bound to the two edge ids of the path, respectively. Executed on the graph in Fig. 1, one of the returned bindings is:

$s \mapsto$ a1, $e \mapsto$ t1, $m \mapsto$ a3, $f \mapsto$ t2, $t \mapsto$ a2.

We can still use the constructs previously described (filters, orientation, labels) in each individual edge or node pattern; for example:

MATCH (`p:Phone` **WHERE** `p.isBlocked='yes'`**)**
            ~[`e:hasPhone`**]~(**`a1:Account`**)**
            -[`t:Transfer` **WHERE** `t.amount>1M`**]->(**`a2`**)**

It searches for substantial transfers from accounts into which a login attempt was made from a blocked phone. The query still extracts paths of length two, but the first edge is undirected and the second is directed and in forward orientation. Hence, each returned binding maps `p`, `a1`, `a2` to nodes n_1, n_2, n_3, and `e`, `t` to edges e_1, e_2, such that e_1 links n_2 and n_1, while e_2 goes from n_2 to n_3.

Moreover, one may use the same variable multiple times in order to impose topological constraints on the matched paths via an implicit equi-join on the repeated variable. For instance in the query below, the variable `s` is used twice; hence this query finds "triangles" of accounts involved in money transfers; the reuse of variable `s` ensures that one starts and ends in the same node:

```
MATCH (s)-[:Transfer]->(s1)-[:Transfer]->(s2)-[:Transfer]->(s)
```

Path patterns permit the use of *path variables*: in the returned bindings, such a path variable is bound to a whole path. In

```
MATCH p = (s)-[:Transfer]->(s1)-[:Transfer]->(s2)-[:Transfer]->(s)
```

the variable `p` will be bound to paths of length three of `Transfer` edges that start and end in the same node. This variable `p` could be returned, or used to compute a more complex value, for instance an aggregate over the bound path.

As in previous examples, the presence of variables in node or edge patterns is not compulsory. For instance, the query

```
MATCH (p:Phone)~[:hasPhone]~(s:Account)-[t:Transfer]->
      (d:Account)~[:hasPhone]~(p)
```

extracts the transfers between accounts that were accessed from the same phone. Since the `hasPhone` edges have no additional property, there is no need to return them. It thus returns two bindings:

$$p \mapsto p1, \quad s \mapsto a5, \quad t \mapsto t8, \quad d \mapsto a1$$
$$p \mapsto p2, \quad s \mapsto a3, \quad t \mapsto t2, \quad d \mapsto a2$$

4.3 Graph Patterns

Graph patterns combine several path patterns together. As an example, consider the fraud detection query looking for substantial transfers from accounts into which a login attempt was made from a blocked phone. It could be alternatively written as

```
MATCH (p:Phone WHERE p.isBlocked='yes')~[:hasPhone]~(s:Account),
      (s)-[t:Transfer WHERE t.amount>1M]->()
```

by splitting the path into two edges and reusing the variable `s` to indicate that these edges share a node. In general, we can put arbitrarily many path patterns together. For example, we can modify the above query by adding a condition that another login attempt into an account was made that did not use a phone:

```
MATCH (s:Account)-[:SignInWithIP]-(),
      (s)-[t:Transfer WHERE t.amount>1M]->(),
      (s)~[:hasPhone]~(p:Phone WHERE p.isBlocked='yes')
```

Quantifier	Description
{m,n}	between m and n repetitions
{m,}	m or more repetitions
*	equivalent to {0,}
+	equivalent to {1,}

Fig. 6. Table of quantifiers

This pattern looks for three edges coming out of s which is cumbersome to write as a single path. In such graph patterns, each path pattern produces a set of mappings, which are then joined using variables that are shared between them.

4.4 Quantifiers and Group Variables

GPML includes quantifiers similar to those in Perl and other common "regex" tools. Figure 6 lists the quantifiers of GPML. Quantifiers are written as postfix operators on either a single edge pattern or a parenthesized path pattern (a path pattern enclosed in parentheses or square brackets with an optional **WHERE** clause). For example, a path of length 2 to 5 of Transfer edges can be sought as follows:

MATCH (a:Account**)-[**:Transfer**]->**{2,5}**(**b:Account**)**

As an example using a parenthesized path pattern, consider the problem of finding paths of 2 to 5 Transfer edges between accounts with the same owner:

MATCH [**(**a:Account**)-[**:Transfer**]->(**b:Account**) WHERE** a.owner=b.owner]{2,5}

The pattern does not provide a strict alternation of node and edge patterns; it will be unrolled two to five times, to obtain a sequence of bindings of variables a b a b… The **WHERE** clause here applies to each such pair of a and b bindings separately. As variables a and b will occur multiple times in such a sequence, the notion of binding an element variable to a unique node or vertex is insufficient in this case, and must be expanded to embrace *path bindings* in which each variable in that sequence is mapped to a graph element. In addition, at the transition between groups, the previous binding of b must be the same as the next binding of a. More about the exact mechanism of such bindings will follow from the detailed example in Section 6.

As for a quantifier on a bare edge pattern, this is understood by supplying anonymous node patterns to its left and right. For example, if we are interested in pairs of accounts involved in a chain of large transfers of length between 2 and 5 we could write

MATCH (a:Account**)**
    [**()-[**t:Transfer**]->() WHERE** t.amount>1M]{2,5}
    (b:Account**)**

Variable references are categorized as either *singleton* or *group*. For example, in predicates such as x.isBlocked='yes', the variable x is referenced as a singleton. Intuitively, a reference is group if you

have to cross a quantifier to get from the reference to the declaration of the variable. To explain this, consider a modification of the above example where we are only interested in chains of transfers with the total value over 10 million:

```
MATCH (a:Account)
        [()-[t:Transfer]->() WHERE t.amount>1M]{2,5}
        (b:Account)
WHERE SUM(t.amount)>10M
```

In this example, the predicate in the edge pattern references t as a singleton, since one does not have to cross the quantifier {2,5} to reach its declaration which is in the same edge pattern. The predicate in the final **WHERE** clause, used here as a postfilter, references t in the aggregate **SUM**. In this location one has to cross the quantifier to reach the declaration of t, making it a group reference.

4.5   Path Pattern Union & Multiset Alternation

In GPML there are two forms of union, called path pattern union (with set semantics) and multiset alternation (with multiset semantics). Path pattern union is indicated by an infix vertical bar, whereas multiset alternation uses |+| as its infix operator. An example of path pattern union is

```
MATCH (c:City) | (c:Country)
```

In the sample graph, there are two Country nodes (c_1 and c_2) and one City node (c_2). Thus the first operand produces two results $c \mapsto c_1$ and $c \mapsto c_2$ and the second operand produces the single result $c \mapsto c_2$. Thus the operands produce a duplicate binding to c_2, which will be reduced to a single solution in the final result, which has one binding to c_1 and one binding to c_2.

Rewriting this example with multiset alternation, we have

```
MATCH (c:City) |+| (c:Country)
```

This pattern returns *three* results, one result binding c_1 and two results binding c_2.

Another example of deduplication using path pattern union is overlapping quantifiers, as in this example:

```
MATCH ->{1,5} | ->{3,7}
```

The two quantifiers overlap between 3 and 5; consequently when the results are deduplicated the query is equivalent to:

```
MATCH ->{1,7}
```

Using multiset alternation would not deduplicate the overlap in the quantifiers.

While this particular example is easy to deduplicate at compile time, a general algorithm to simplify arbitrary path pattern unions in the language that includes **WHERE** clauses, restrictors, nested quantifiers, etc., is not so clear and is likely to have a very high complexity, making it an unattractive option for the implementer. Run-time deduplication is also expensive, and from a user perspective, it may be acceptable to get a result that is a multiset, rather than waiting a long time for a set result. This is akin to the SQL

situation with **UNION ALL** chosen for its speed over **UNION DISTINCT** guaranteeing set semantics. For these reasons GPML provides both path pattern union and multiset alternation.

4.6   Conditional Variables

Consider the following query.

$$\texttt{MATCH } [\texttt{(x)->(y)}] \ | \ [\texttt{(x)->(z)}]$$

Node variable x will be bound if either operand of the path pattern union binds. Variables y and z, on the other hand, are only bound by one operand but not by the other. We say that x is an *unconditional singleton* whereas y and z are *conditional singletons*.

Implicit equi-joins on conditional singletons are disallowed, because they lack intuitive semantics. For instance, in the illegal query

$$\texttt{MATCH } [\texttt{(x)->(y)}] \ | \ [\texttt{(x)->(z)}], \ \texttt{(y)->(w)}$$

does the fact that y must bind in the second path pattern imply that y must bind in the path pattern union, thereby effectively excluding the results of the second operand? Implicit equi-joins on conditional singletons were forbidden in order to eliminate such doubts.

Conditional singletons are also introduced by the question mark operator, a postfix operator similar to the quantifiers; for example

$$\texttt{MATCH (x) } [\texttt{->(y)}]?$$

In most "regex" tools, the postfix question mark operator is equivalent to the quantifier {0,1}. In GPML the two operators have almost the same semantics. The difference is that the quantifier {0,1} exposes all variables as group, whereas the question mark operator exposes singletons as conditional singletons. This distinction was made so that query-generating tools can solicit upper and lower bounds for a quantifier from the user without having to watch for {0,1} as a special case.

To provide an example of the use of conditional variables in our running example, consider a query asking for accounts from which mobney was transferred to either a blocked account or an account with a login from a blocked phone. It could be written as path patetrn union:

```
MATCH [(x:Account)-[:Transfer]->(y:Account WHERE y.isBlocked='yes')] |
      [(x:Account)-[:Transfer]->()-[:hasPhone]-(p WHERE p.isBlocked='yes')]
```

Alternatively, one can use the question mark operator:

```
MATCH (x:Account)-[:Transfer]->(y:Account) [-(:hasPhone)-(p)]?
WHERE y.isBlocked='yes' OR p.isBlocked='yes'
```

If the optional part of the pattern is not matched, then the condition p.isBlocked='yes' is not true, and thus the account must be blocked to ensure that the condition in **WHERE** is true.

4.7 Graphical Predicates

In addition to familiar predicates such as comparison (e.g., equal to, greater than, etc.) and **IS NULL**, GPML specifies some predicates relevant to graphs (applicable to singleton variable references).

Many of the edge patterns are ambiguous about the orientation of an edge. For example, `-[e]-` matches a directed edge pointing either left or right, or an undirected edge. The user may wish to interrogate the orientation of the edge bound to `e`. For this, GPML provides the following predicates:

> `e IS DIRECTED` is true if `e` is bound to a directed edge;
>
> `s IS SOURCE OF e` is true if `s` is bound to the source of `e`;
>
> `d IS DESTINATION OF e` is true if `d` is bound to the target of `e`.

SQL/PGQ is not able to use the equals sign to test for equality of element references, because `p=q` is interpreted as an equality test on two column values, therefore `p` and `q` cannot be interpreted as element references. GQL, on the other hand, will permit such equality tests of element references. To provide a portable way to test equality of element references, GPML provides the following predicates:

- `SAME(p, q, ...)`: test whether all element references in the list are bound to the same graph element.
- `ALL_DIFFERENT(p, q, ...)`: test whether the element references in the list are pairwise distinct.

The element references listed in `SAME` or `ALL_DIFFERENT` must be unconditional singletons because the semantics of such expressions for conditional singletons would be ambiguous.

5 ASSURING TERMINATION

Written without any restrictions, GPML queries may not terminate as they will return infinitely many matches. Consider for example,

$$\textbf{MATCH } p = \textbf{(}a \textbf{ WHERE } a.owner='Dave'\textbf{)-[}t:Transfer\textbf{]->}*$$
$$\textbf{(}b \textbf{ WHERE } b.owner='Aretha'\textbf{)}$$

It asks for paths with any number of transfers between the account owned by `Dave` and the account owned by `Aretha`. If evaluated over the graph of Figure 1, this query would have an infinite number of matches, since the path matched by `p` could include a `Transfer` loop (e.g., path(a3,t7,a5,t8,a1,t1,a3)) any number of times.

To prevent this behaviour, GPML queries must *demonstrably terminate*; in particular, the number of matches must be finite. To achieve this, GPML uses *restrictors* and *selectors*. Every unbounded quantifier (such as `*` above) must be contained in the scope of either a restrictor or a selector or both.

5.1 Restrictors and Selectors

Restrictors. A restrictor is a path predicate (that is, it imposes a condition stating which paths are acceptable) such that the number of matches cannot be infinite. For instance, **TRAIL** forbids the matched path

to repeat edges; there are only finitely many edges, therefore there are only finitely many paths with no repeated edges in a graph. Restrictors are listed in Figure 7. Restrictors may be placed either at the head of a path pattern, or at the head of a parenthesized path pattern. If the above example is written as

> **MATCH TRAIL** p = **(**a **WHERE** a.owner='Dave'**)−[**t:Transfer**]->**⋆
>   **(**b **WHERE** b.owner='Aretha'**)**

then, executed on the graph of Fig. 1, returns three bindings for p:

$$\text{path}(\text{a6},\text{t5},\text{a3},\text{t2},\text{a2})$$

$$\text{path}(\text{a6},\text{t6},\text{a5},\text{t8},\text{a1},\text{t1},\text{a3},\text{t2},\text{a2})$$

$$\text{path}(\text{a6},\text{t5},\text{a3},\text{t7},\text{a5},\text{t8},\text{a1},\text{t1},\text{a3},\text{t2},\text{a2})$$

Note that the last path repeats the node a3; it is allowed by **TRAIL** but would be forbidden by the restrictor **ACYCLIC**. Note also that

$$\text{path}(\text{a6},\text{t5},\text{a3},\text{t2},\text{a2},\text{t3},\text{a4},\text{t4},\text{a6},\text{t5},\text{a3},\text{t2},\text{a2})$$

which traverses the `Transfer` cycle starting and ending in a6, is not a trail, and is thus not returned.

Selectors. A selector is an algorithm that conceptually partitions the solution space on the endpoints and selects a finite set of matches from each partition. For instance, **ALL SHORTEST** keeps all paths having the same shortest length within each partition defined by a pair of endpoints for which any solution exists. Note that the shortest length can differ from partition to partition. Selectors are listed in Figure 8. Selectors may only be placed at the head of a path pattern. As an example, applying a selector to the example from the beginning of Section 5 can be done as follows:

> **MATCH ANY SHORTEST**
> p = **(**a **WHERE** a.owner='Dave'**)−[**t:Transfer**]->**⋆**(**b **WHERE** b.owner='Aretha'**)**

Here we wish only one of the shortest paths between the nodes a6 and a2. In this case, there is only one shortest path between these nodes and thus p is bound to path(a6,t5,a3,t2,a2).

Combining restrictors and selectors. At the conceptual level, restrictors can be seen as operating during pattern matching while selectors operate afterwards. [5] That is, if combined, selectors are always applied *after* restrictors. For instance, consider the query:

> **MATCH ALL SHORTEST TRAIL**
> p = **(**a **WHERE** a.owner='Dave'**)−[**t:Transfer**]->**⋆
>   **(**b **WHERE** b.owner='Aretha'**)−[**r:Transfer**]->**⋆**(**c **WHERE** c.owner='Mike'**)**

---

[5]This is of course only an intuitive description: in the case of selectors, the engine will not compute an infinite number of matches and then only keep some. However, to make the specification declarative ("tell me what you want, not how to get it") the specification indeed describes how the infinite set is defined and then which finite selection from it should be returned.

Keyword	Description
TRAIL	No repeated edges.
ACYCLIC	No repeated nodes.
SIMPLE	No repeated nodes, except that the first and last nodes may be the same.

Fig. 7. Table of restrictors

It selects the shortest paths *among* the trails going from node a6 to node a3 and passing through a2. It returns two bindings for p:

$$\text{path}(a6,t5,a3,t2,a2,t3,a4,t4,a6,t6,a5,t8,a1,t1,a3)$$

$$\text{path}(a6,t6,a5,t8,a1,t1,a3,t2,a2,t3,a4,t4,a6,t5,a3)$$

The path path(a6,t5,a3,t2,a2,t3,a4,t4,a6,t5,a3) is not considered: it is shorter but it is not a trail.

This difference has one notable consequence. Consider a query Q with no selector or restrictor, and assume that Q has matches. Then, adding a selector to Q might reduce the number of matches, but the resulting query will always have at least one match. On the other hand, adding a restrictor to Q might yield a query with no matches at all.

For example, consider the query

```
MATCH (p:Account WHERE p.owner='Natalia')->{1,10}
      (q:Account WHERE q.owner='Mike')->{1,10}
      (r:Account WHERE r.owner='Scott')
```

In the example graph, path(a5,t8,a1,t1,a3,t7,a5,t8,a1) is a solution to this query, and in fact it is a shortest solution in the partition defined by the endpoints (a5,a1). However, it repeats the edge t8, so it fails the restrictor **TRAIL** (as well as **SIMPLE** and **ACYCLIC**. Thus adding a selector such as **ALL SHORTEST** will still have a result, whereas adding a restrictor such as **TRAIL** will have no result.

5.2   Prefilters and Postfilters of Selectors

When working with selectors, it is important to differentiate *prefilters* from *postfilters*. A prefilter is a predicate applied before selection; a postfilter is a predicate applied after selection. In GPML, postfilters are expressed in the final **WHERE** clause, whereas prefilters are expressed in element patterns or parenthesized path patterns within the selector's path pattern.

For example, suppose we want to find the shortest path from Scott to Charles passing through an account that is blocked:

```
MATCH ALL SHORTEST (p:Account WHERE p.owner='Scott')->+
                   (q:Account WHERE q.isBlocked='yes')->+
                   (r:Account WHERE r.owner='Charles')
```

Keyword	Description
ANY SHORTEST	Selects one path with shortest length from each partition. Non-deterministic.
ALL SHORTEST	Selects all paths in each partition that have the minimal length in the partition. Deterministic.
ANY	Selects one path in each partition arbitrarily. Non-deterministic.
ANY k	Selects arbitrary k paths in each partition (if fewer than k, then all are retained). Non-deterministic.
SHORTEST k	Selects the shortest k paths (if fewer than k, then all are retained). Non-deterministic.
SHORTEST k **GROUP**	Partitions by endpoints, sorts each partition by path length, groups paths with the same length, then selects all paths in the first k groups from each partition (if fewer than k, then all are retained). Deterministic.

Fig. 8. Table of selectors

Note that all predicates in this example are expressed in element patterns, therefore they are all prefilters. The only solution is the path(a1,t1,a3,t2,a2,t3,a4, t4,a6,t5,a3,t7,a5), in which q is bound to a4 (Jay, the only blocked account).

It would be a mistake to place the predicate on q in the final **WHERE** clause, like this:

```
MATCH ALL SHORTEST (p:Account WHERE p.owner='Scott')->+
                   (q:Account)->+
                   (r:Account WHERE r.owner='Charles')
WHERE q.isblocked='yes'
```

The shortest path from Scott to Charles is path(a1,t1,a3,t7,a5) with q bound to a3, which will be the result of the selector, but this result is then filtered out by the final **WHERE** clause because a3 is not blocked. Consequently this query finds no result. The original problem statement had the predicate *"passing through an* Account *that is blocked"* as a prefilter, therefore the query placing it as a postfilter is incorrect.

5.3   Aggregates of Unbounded Variables

There is another subtle way of having non-terminating queries that must be ruled out by GPML. Consider this query:

```
MATCH ALL SHORTEST [ (x)-[e]->*(y) WHERE COUNT(e.*)/(COUNT(e.*)+1)>1 ]
```

Note carefully that the predicate is within a parenthesized path pattern, therefore it is a prefilter. As a prefilter, the group variable e has not yet passed through the selector **ALL SHORTEST**; consequently the predicate sees e as effectively unbounded.

If a match has 0 edges, the quotient in the **WHERE** clause is 0; if it has 1 edge, the quotient is 1/2; if it has 2 edges, the quotient is 2/3; etc. The quotient can never exceed 1. This is easily seen by human reason, but generalizing this observation to any aggregate on an unbounded element reference is not obvious. For example, suppose the aggregate were **AVG**(e.x); the behavior of **AVG** on an unbounded collection of property references is not easy to anticipate. A few aggregates (**MAX**, **MIN**, **COUNT**) are monotonic, which might permit reasoning on simple expressions such as linear combinations, but it is currently expected that GPML will simply prohibit all predicates on unbounded groups to ensure termination.

The query can be made acceptable – i.e., turned into a terminating one – in several ways. First, the predicate might be converted to a postfilter, like this:

$$\texttt{MATCH ALL SHORTEST (x)-[e]->\textsubscript{*}(y)}$$
$$\texttt{WHERE COUNT(e.*)/(COUNT(e.*)+1) > 1}$$

Note that all we did was remove the square brackets, which moved the predicate out of the path pattern and into the final **WHERE** clause. This made e effectively bounded. Of course any results produced by the selector will be filtered out by the postfilter; therefore the result of this query is empty. However, turining the predicate into a postfilter and using a selector first ensures the finiteness of the result, and thus termination.

A different approach is to retain the prefilter but insure that e is effectively bounded within the selector. Certainly changing the quantifier to have a static upper bound such as {0,10} would do. Alternatively a restrictor could be used, like this:

MATCH ALL SHORTEST [**TRAIL (x)-[e]->**∗**(y) WHERE COUNT(**e.***)/(COUNT(**e.***)+1) > 1**]

Reasoning about this pattern we can see that the result will be empty. An implementation is of course free to implement algorithms that detect some queries that are unsatisfiable and thereby terminate quickly. Alternatively, the implementation could actually tackle the query as written and eventually arrive at the conclusion that the result set is empty.

6 EXECUTION MODEL BY EXAMPLE

We illustrate GPML evaluation using a step-by-step example. The output of the example query is a set, or a multiset, of *reduced path bindings*. A *path binding* is a sequence of *elementary bindings*, which in turn are pairs of a variable and a graph element. A *variable* is either a variable that occurs in the pattern, or a new variable corresponding to an anonymous node or edge. Each variable, anonymous or not, can be further indexed if it appears under the scope of a quantifier: the index indicates the iteration in which it appears. For the same variable a under the scope of a quantifier {1,} we can have variables $a^1, a^2, a^3, \ldots$ that correspond to different iterations of the quantifier. Obtained path bindings are then *reduced*, stripping all annotations, and *deduplicated*, to remove duplicates. If multiple path patterns appear after **MATCH**, the process is repeated for all of them, and the result is joined on identical singleton variables, and possibly

further filtered if there is a **WHERE** clause. Selectors and restrictors are used to ensure that the output is finite in the presence of quantifiers.

The key steps in the pattern matching execution model, as reflected in the forthcoming standard, are as follows.

Normalization GPML provides syntactic sugar to help write patterns; this step puts patterns in a canonical form.

Expansion The pattern is expanded into a set of *rigid patterns* without any kind of disjunction. Intuitively, a rigid pattern is one that could be expressed by an SQL equi-join query. Formally, it is a pattern without quantifiers, union or multiset alternation. The expansion also annotates each rigid pattern to enable tracking the provenance of the syntax constructs.

Rigid-pattern matching For each rigid pattern, one computes a set of *path bindings*. Each elementary construct of the rigid pattern is computed independently and then the results are joined together on variables with the same name.

Reduction and deduplication The path bindings matched by the rigid patterns are *reduced* by removing annotations. The reduced path bindings are then collected into a set. This implies a *deduplication* step since different path bindings might become equal after reduction, and only one copy is kept.

6.1 Running Example

The remainder of this section gives a detailed account of how pattern matching is computed for the following query.

```
MATCH TRAIL (a WHERE a.owner='Jay')
              [-[b:Transfer WHERE b.amount>5M]->]+
              (a) [-[:isLocatedIn]->(c:City) |
                  -[:isLocatedIn]->(c:Country)]
```

This query finds sequences of transfers of arbitrary length that start and end with account owner `Jay`, as well as the location (city or country) of `Jay`. To ensure termination, the **TRAIL** mode is used.

6.2 Normalisation

The first step of normalisation makes each sequence of node and edge patterns consistent. More precisely, each such sequence must start and end with a node pattern; and alternate between node and edge patterns. In addition, syntactic sugar is expanded, e.g., quantifier + is replaced by {1, }. Hence, the pattern becomes:

```
(a WHERE a.owner='Jay')
[()-[b:Transfer WHERE b.amount>5M ]->()]{1,}
(a) [()-[:isLocatedIn]->(c:City) |
     ()-[:isLocatedIn]->(c:Country)]
```

We then introduce a fresh variable into each anonymous node and edge pattern (that is, a pattern that is not assigned to a variable). The fresh node and edge variables are denoted by $\square_x$ and $-_x$, respectively, for some index x. The pattern then becomes

```
(a WHERE a.owner='Jay')
[(□i)-[b:Transfer WHERE b.amount>5M]->(□ii)]{1,}
(a)[[(□iii)-[-i:isLocatedIn]->(c:City)] |
      [(□iv)-[-ii:isLocatedIn]->(c:Country)]]
```

6.3 Expansion

Expansion turns the pattern into a set of rigid patterns, which fix the number of iterations for each quantifier and a disjunct for each union or alternation. In our query, this amounts to choosing a disjunct in the union `|`, and expanding the quantifier `{1,}`. While unbounded quantifiers such as `{1,}` make the set infinite, techniques of Section 5 will make the evaluation feasible. The following pattern is one possible expansion, where we expanded the quantifier once and chose the left side of the path pattern union:

```
(a WHERE a.owner='Jay')
(□i¹)-[b¹:Transfer WHERE b¹.amount>5M]->(□ii¹)
(a)  (□iii)-[-i:isLocatedIn]->(c:City)
```

In general, one such pattern is obtained for each $n \in \mathbb{N} \setminus \{0\}$ and each $\ell \in \{\texttt{City}, \texttt{Country}\}$: we expand the quantifier n times, and choose either the `City` or the `Country` option of the path pattern union. Such a pattern is denoted by $\theta_{n,\ell}$. As an example, the pattern below is $\theta_{n,\texttt{City}}$.

```
(a WHERE a.owner='Jay')
(□i¹)-[b¹:Transfer WHERE b¹.amount>5M]->(□ii¹)
(□i²)-[b²:Transfer WHERE b².amount>5M]->(□ii²)
      ⋮              ⋮              ⋮
(□iⁿ)-[bⁿ:Transfer WHERE bⁿ.amount>5M]->(□iiⁿ)
(a)  (□iii)-[-i:isLocatedIn]->(c:City)
```

The prototype $\theta_{n,\texttt{Country}}$ is similar, only the last line changes.

Each group variable is marked by a superscript, corresponding to the iteration of the quantifier the variable is in.

The next step is a clean-up: every node pattern with an anonymous variable is deleted if it is adjacent to another node pattern. The resulting pattern is denoted by $\pi_{n,\ell}$. For instance, $\pi_{n,\texttt{City}}$ is:

```
(a WHERE a.owner='Jay')
```
$-$`[`b^1`:Transfer` **WHERE** b^1`.amount>5M]`$->$`(`$\square_{ii}^1$`)`
$-$`[`b^2`:Transfer` **WHERE** b^2`.amount>5M]`$->$`(`$\square_{ii}^2$`)`
$$\vdots \qquad\qquad\qquad \vdots \qquad\qquad\qquad \vdots$$
$-$`[`b^{n-1}`:Transfer` **WHERE** b^{n-1}`.amount>5M]`$->$`(`$\square_{ii}^{n-1}$`)`
$-$`[`b^n`:Transfer` **WHERE** b^n`.amount>5M]`$->$`(a)`
$-$`[`$-_i$`:isLocatedIn]`$->$`(c:City)`

6.4   Computation of Path Binding

The matches of rigid patterns are computed independently. The result of the computation is called a *path binding*, which in turn is a sequence of *elementary bindings*. An *elementary binding* is a pair of a variable and a graph element. It is convenient to portray these pairs as tables with two rows: the first row contains the variables, the second row contains the graph elements, and each column is an elementary binding. An example path binding is

$$\text{a} \quad b^1 \quad \square_{ii}^1$$
$$\text{a4} \quad \text{t4} \quad \text{a6}$$

Consider the rigid path pattern $\pi_{4,\text{City}}$. Each node-edge-node pattern is computed independently using the input graph, and then equi-joined on variables with the same name. The first node-edge-node part of $\pi_{4,\text{City}}$ is

`(a` **WHERE** `a.owner='Jay')`$-$`[`b^1`:Transfer` **WHERE** b^1`.amount>5M]`$->$`(`$\square_{ii}^1$`)`

In the graph it matches only one path binding, the one shown above. The second node-edge-node part is

`(`$\square_{ii}^1$`)`$-$`[`b^2`:Transfer` **WHERE** b^2`.amount>5M]`$->$`(`$\square_{ii}^2$`)`

and the path bindings it matches are given in the left column below. The path bindings matched by the third, fourth and fifth parts of $\pi_{4,\text{City}}$ are given in the three other columns.

$\square_{ii}^1$ b^2 $\square_{ii}^2$	$\square_{ii}^2$ b^2 $\square_{ii}^3$	$\square_{ii}^3$ b^4 a	a $-_i$ c
a6 t5 a3	a6 t5 a3	a6 t5 a3	a4 li4 c2
a3 t2 a2	a3 t2 a2	a3 t2 a2	a6 li6 c2
a2 t3 a4	a2 t3 a4	a2 t3 a4	a3 li3 c1
a2 t3 a4	a2 t3 a4	a2 t3 a4	a2 li2 c2
a3 t7 a5	a3 t7 a5	a3 t7 a5	a5 li5 c1
a5 t8 a1	a5 t8 a1	a5 t8 a1	a1 li1 c1
a1 t1 a3	a1 t1 a3	a1 t1 a3	

Note that the matches for each part (i.e., each column above) is computed independently.

The labels are matched and the **WHERE** clauses are checked at this stage. Thus, the edge (a6,t6,a5) does not appear anywhere above as it fails the **WHERE** condition, nor does the edge (ip1,sip1,a1) since it has neither the Transfer nor the isLocatedIn label.

Then the path bindings are concatenated by an implicit equi-join on variables with the same name. In the end there is only one path binding for $\pi_{4,\text{City}}$ given below.

$$\text{a} \ \text{b}^1 \ \square^1_{ii} \ \text{b}^2 \ \square^2_{ii} \ \text{b}^3 \ \square^3_{ii} \ \text{b}^4 \ \text{a} \ -_i \ \text{c}$$
$$\text{a4 t4 a6 t5 a3 t2 a2 t3 a4 li4 c2}$$

Note that variables with different subscript or superscript (e.g., b^1 and b^2) are not joined on.

Restrictors are also checked at this point. For example, $\pi_{8,\text{City}}$ has no match. Indeed, a path binding computed by the above join would use the loop (t4,t5,t2,t3) twice, and thus would not be a trail.

In the end, $\pi_{n,\ell}$ never has any matches unless $n = 4$ (as presented above) or $n = 7$. The patterns $\pi_{4,\text{City}}$, $\pi_{4,\text{Country}}$, $\pi_{7,\text{City}}$ and $\pi_{7,\text{Country}}$ each have one match given below.

$$\text{a} \ \text{b}^1 \ \square^1_{ii} \ \text{b}^2 \ \square^2_{ii} \ \text{b}^3 \ \square^3_{ii} \ \text{b}^4 \ \text{a} \ -_i \ \text{c}$$
$$\text{a4 t4 a6 t5 a3 t2 a2 t3 a4 li4 c2}$$

$$\text{a} \ \text{b}^1 \ \square^1_{ii} \ \text{b}^2 \ \square^2_{ii} \ \text{b}^3 \ \square^3_{ii} \ \text{b}^4 \ \text{a} \ -_{ii} \ \text{c}$$
$$\text{a4 t4 a6 t5 a3 t2 a2 t3 a4 li4 c2}$$

$$\text{a} \ \text{b}^1 \ \square^1_{ii} \ \text{b}^2 \ \square^2_{ii} \ \text{b}^3 \ \square^3_{ii} \ \text{b}^4 \ \square^4_{ii} \ \text{b}^5 \ \square^5_{ii} \ \text{b}^6 \ \square^6_{ii} \ \text{b}^7 \ \text{a} \ -_i \ \text{c}$$
$$\text{a4 t4 a6 t5 a3 t7 a5 t8 a1 t1 a3 t2 a2 t3 a4 li4 c2}$$

$$\text{a} \ \text{b}^1 \ \square^1_{ii} \ \text{b}^2 \ \square^2_{ii} \ \text{b}^3 \ \square^3_{ii} \ \text{b}^4 \ \square^4_{ii} \ \text{b}^5 \ \square^5_{ii} \ \text{b}^6 \ \square^6_{ii} \ \text{b}^7 \ \text{a} \ -_{ii} \ \text{c}$$
$$\text{a4 t4 a6 t5 a3 t7 a5 t8 a1 t1 a3 t2 a2 t3 a4 li4 c2}$$

The latter two path bindings are trails, but are not acyclic, since the node a3 appears twice. Thus, they would have been filtered out if we had used the restrictor **ACYCLIC** instead of **TRAIL**.

6.5 Reduction and Deduplication

Reduction of path bindings strips variables from their annotations (subscripts and superscripts). In particular, it merges together all variables introduced in anonymous element patterns. The result of reduction in our example is show below:

$$\text{a} \ \text{b} \ \square \ \text{b} \ \square \ \text{b} \ \square \ \text{b} \ \text{a} \ - \ \text{c}$$
$$\text{a4 t4 a6 t5 a3 t2 a2 t3 a4 li4 c2}$$

$$\text{a} \ \text{b} \ \square \ \text{b} \ \square \ \text{b} \ \square \ \text{b} \ \text{a} \ - \ \text{c}$$
$$\text{a4 t4 a6 t5 a3 t2 a2 t3 a4 li4 c2}$$

$$\text{a} \ \text{b} \ \square \ \text{b} \ \square \ \text{b} \ \square \ \text{b} \ \square \ \text{b} \ \square \ \text{b} \ \square \ \text{b} \ \text{a} \ - \ \text{c}$$
$$\text{a4 t4 a6 t5 a3 t7 a5 t8 a1 t1 a3 t2 a2 t3 a4 li4 c2}$$

$$\text{a} \ \text{b} \ \square \ \text{b} \ \square \ \text{b} \ \square \ \text{b} \ \square \ \text{b} \ \square \ \text{b} \ \square \ \text{b} \ \text{a} \ - \ \text{c}$$
$$\text{a4 t4 a6 t5 a3 t7 a5 t8 a1 t1 a3 t2 a2 t3 a4 li4 c2}$$

All reduced path bindings are then collected together in a set. That is, at this point, *deduplication* occurs. If two different rigid patterns yield the same reduced path binding, then a single copy of the path binding is kept. Thus, the final result has only two distinct reduced path bindings.

```
        a  b  □  b  □  b  □  b  a  —  c
        a4 t4 a6 t5 a3 t2 a2 t3 a4 li4 c2
```

```
    a  b  □  b  □  b  □  b  □  b  □  b  □  b  a  —  c
    a4 t4 a6 t5 a3 t7 a5 t8 a1 t1 a3 t2 a2 t3 a4 li4 c2
```

The consequence of this deduplication is that our running query is equivalent to:

```
    MATCH TRAIL
       (a WHERE a.owner='Jay') [-[b:Transfer WHERE b.amount>5M]->]+
       (a)-[:isLocatedIn]->(c:City|Country)
```

Hence, a user does can turn a disjunction in the label expression into a path pattern union.

Having finished this detailed example, we now look at some other features of the pattern matching algorithm.

Using selectors. After deduplication, any selectors (if present) would be applied. Assume that in our running example we replaced the restrictor **TRAIL** with the selector **ALL SHORTEST**:

```
    MATCH ALL SHORTEST
       (a WHERE a.owner='Jay') [-[b:Transfer WHERE b.amount>5M]->]+
       (a) [-[:isLocatedIn]->(c:City) | -[:isLocatedIn]->(c:Country)]
```

While the number of reduced path bindings would be infinite (since `Transfer` loops may be taken arbitrarily many times without the trail restriction), the selector would keep the shortest reduced binding path for each pair of endpoints (in our case, a4 and c2), thus returning

```
        a  b  □  b  □  b  □  b  a  —  c
        a4 t4 a6 t5 a3 t2 a2 t3 a4 li4 c2
```

Multiple patterns. When more than one pattern, separated by commas, appear after **MATCH**, each path pattern is solved separately, resulting in finitely many solutions to each path pattern. At this point the cross product of the sets of path bindings is formed, and then filtered on the basis of implicit equi-joins on the global singleton variables and the final **WHERE** clause.

Path pattern union vs multiset alternation. The consequence of using deduplication is that our running query with path pattern union is equivalent to a query where the last edge pattern is replaced by `(a)-[:isLocatedIn]->(c:City|Country)`.

To avoid deduplication and to maintain four reduced path bindings in the output, one could use multiset alternation instead, replacing the pattern by

```
    (a) [-[:isLocatedIn]->(c:City) |+| -[:isLocatedIn]->(c:Country)]
```
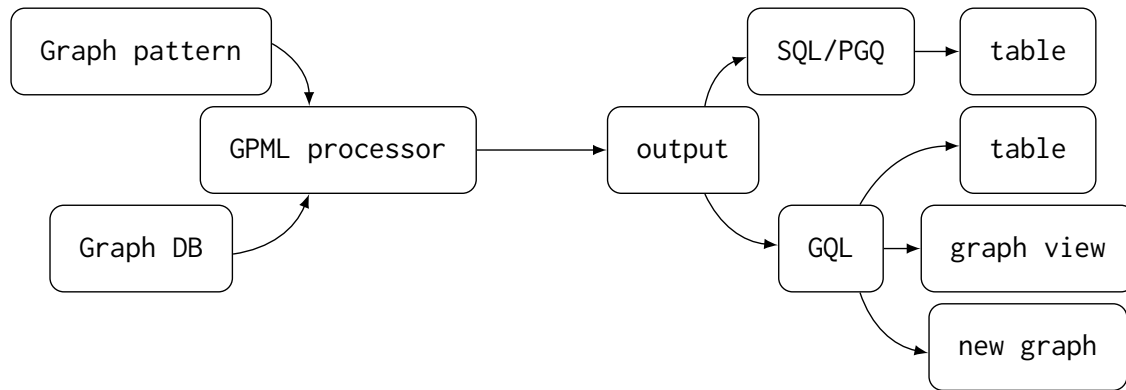
Fig. 9. Conceptual diagram of GPML, SQL/PGQ and GQL

6.6 Query Outputs

How should the result of pattern matching be represented to produce the output of a query? We have seen that executing a GPML statement results in a set of path bindings. Presenting this to the user depends on the host language, SQL/PGQ or GQL. Figure 9 shows the relationship between GPML and its two host languages.

The GPML processor is a software component within an implementation of either SQL/PGQ or GQL. The two inputs to the GPML processor are the graph pattern and the graph database. The output of the GPML processor is consumed by the host to produce the final output requested by the user. For SQL/PGQ, it will be a table, obtained from the computed reduced path bindings.

For GQL, the output could be more varied, including a graph view, or new graph. Indeed, each path binding defines a subgraph of the input graph given by its nodes and edges, together with annotations, given by variables assigned to them in the path binding. This opens up more possibilities for structuring query outputs. While in the initial release of the GQL standard, outputs will be in line with those of SQL/PGQ, it is anticipated that in the future more advanced options will be added.

7 LOOKING FORWARD

In this section we outline the ongoing work on the development of the SQL/PGQ and GQL standards and list several research problems that have arisen in the process of designing the GPML.

7.1 Standards Process: Steps and Timing

The SQL/PGQ and GQL standards are being developed in the international standards committee ISO/IEC JTC1 SC32 WG3 "Database Languages" with input from various national bodies. In particular, the US committee INCITS DM32 "Data Management and Interchange" and DM32's SQL/PGQ and GQL expert groups review all significant US change proposals before they are considered by WG3.

Date	SQL/PGQ	GQL
2017	Work started	
2018		Work started
2021-02-07	CD Ballot End	
2022-02-20		CD Ballot End
2022-12-04	DIS Ballot End	
2023-01-30	Final Text to ISO	
2023-03-13	**SQL/PGQ IS Published**	
2023-05-21		DIS Ballot End
2023-07-30		Final Text to ISO
2023-09-10		**GQL IS Published**

Fig. 10. SQL/PGQ and GQL Timeline

The ISO/IEC JTC1 process has a number of steps with ballots to transition between the stages. The high-level overview is:

- Initial effort – develop and expand the draft;
- Committee Draft (CD) Ballot – 12 weeks;
- Draft International Standard (DIS) Ballot – 20 weeks;
- International Standard (IS) published.

After each ballot, time is needed to resolve the comments submitted.

The current schedule for the progression of the SQL/PGQ and GQL standards is shown in Figure 10.[6] By the time the DIS ballot starts, the technical specification is fairly stable. Since GPML is the same for GQL and SQL/PGQ, GQL GPML will be fairly stable when SQL/PGQ begins DIS ballot. As SC32 WG3 makes progress on the drafts, it accumulates *Language Opportunities* (LOs). LOs are capabilities that are potentially useful, but are not yet ready for the current versions of the draft standards. Below we provide a sample of LOs pertaining to GPML:

- Constraining a graph pattern through the introduction of isomorphic match modes: for example, an *edge-isomorphic* match requires all edges matched across all constituent path patterns in the graph pattern to differ from each other.
- Queries on multiple graphs in a single concatenated **MATCH**.
- Path macros for multiple use in a query.
- Outputting the interleaving of bindings in nested quantifiers, such as `[[(p)->(q)]* ->(r)]*`.
- Cheapest path search, by adding weights to edges.
- Exporting a graph element or path binding to JSON.

---

[6]The schedule depends on work that has not been completed and so could change.

GQL also has LOs that go beyond those common with SQL/PGQ. Examples include property graph keys and constraints [4], system versioned graphs, and stored queries, procedures, and functions. As discussed in Section 6.6, formats such as JSON could potentially be used for returning a raw multi-path binding.

7.2   Research Questions

There are many open questions related to GPML. Given the novelty of the language, and its goal as the future standard of querying property graphs, it is important to full understand the expressiveness and complexity of its various fragments, in the spirit of many decades of research tradition in database query languages [5].

Some of the most intriguing questions concern processing unbounded paths. For instance, the innocently looking query below may not terminate.

```
MATCH (x)-[e]->*(y)
WHERE AVG(e.a)<1
KEEP ANY SHORTEST
```

With a sufficiently rich pattern language, termination will become an undecidable problem. A natural question is then whether there are interesting classes of predicates on aggregates of group variables for which termination can be guaranteed, or efficiently checked at compile time.

There are many questions related to the implementation of GPML. How does one solve efficiently shortest path queries with arbitrary regular expressions, not just $->*$ as in Dijkstra's algorithm? Can we handle more complex optimization problems, such as maximizing an objective function subject to an upper bound on the length or cost of the path (e.g., "What is the most scenic route to the airport in at most 2 hours?").

Another direction is to consider fully recursive graph patterns, permitting multiple self-references, not just a single one like in the $*$ operator. Such patterns might be used to search for trees and other structures more complex than paths. Is there intuitive syntax to express such patterns? What real-world problems might they address? What is the cost of adding them to GPML?

Yet another line of future works concerns extended capabilities and different uses of graph patterns. One of them concerns extending the language to capture the temporal aspect of data [39, 45]. A different line of work is to understand the underpinnings of languages defined by paths in property graphs, which are more than traditional regular languages over a finite alphabet, as they include property values coming from potentially infinite domains. Simple models for this based on data words [11] have already been studied in [31], but they disregarded operations on data, such as arithmetic or string operations. In an opposite direction, we may look at restrictions that will be useful in different applications, such as for example updating property graphs, which tends to be problematic [25] with a full power of pattern matching.

8   RELATED WORK

GPML is the culmination of a long evolution of graph query languages, including contributions in research, standards and practice, too rich to survey comprehensively here. We discuss related work only in broad strokes and refer the reader to existing surveys [3, 49].

   GPML's graph patterns extend the celebrated language of Conjunctive Regular Path Queries (CRPQ) introduced in [17] and studied in a plethora of follow-up work (e.g. [9, 10, 14–16, 22]). CRPQs consist of variables (typically binding only to individual vertices and edges), path patterns specified by regular expressions over the edge labels, and comma-separated lists thereof. In their first incarnation, CRPQs operated over property-less graphs, but subsequent work considered graphs with data annotations (e.g. [31]), which are ancestors of today's property graphs. Notable early graph query languages derived from CRPQs include G [17], StruQL [21], Lorel [1] and WebSQL [36]. Syntactically, GPML graph patterns extend CRPQs by introducing group variables and variables binding to entire paths. The latter allows GPML to treat paths as first-class citizens, as advocated in the G-CORE proposal [2]. Semantically, graph patterns extend CRPQs by supporting a finer-grained notion of match, which permits binding variables to paths, using multiplicity-sensitive aggregation such as sum, count, and average, and restricting the number of possible returned paths. These features are original contributions of GPML.

   The *simple path* restrictor was already present in the first paper that introduced CRPQs [17]. However, after it was discovered that the evaluation problem for (C)RPQs with simple path and trail restrictors has a high computational complexity [38], they were put on hold by the research community. They were resurrected by practical query languages (e.g., Cypher [23] and early versions of SPARQL 1.1. [27]), which generated another wave of fundamental research [7, 34, 35] that sheds a clearer light on the relationship between queries that appear in real-life and the high complexity of simple paths and trails in the worst case.

   GPML patterns share many features with pattern standards developed for different data models. *XML documents* [13] correspond to tree-shaped, vertex-labeled graphs. The XPath [43] World-Wide Web Consortium (W3C) standard describes patterns that roughly correspond to GPML path patterns that can be combined into tree-shaped structures; the use of variables is limited. The XQuery [44] standard supports comma-separated lists of XPath patterns, stitched together with node variables. Like in GPML, XQuery matches are discriminated by the path bindings they correspond to, but the limited use of quantifiers and the tree shape of the graph avoid the query non-termination problem. *Knowledge Graphs* [28] corresponding to the *RDF* standard [18], also admit pattern-based querying, for instance using the SPARQL W3C standard query language [27]. Patterns consist once more of variables (binding to the RDF counterpart of nodes and edges) and regular path expressions. To deal with cycles, and potentially infinite number of matching paths [6, 32], SPARQL deploys the "endpoint semantics" of paths, and will only return start/end point of a path, instead of the matching path itself.

REFERENCES

[1] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. 1997. The Lorel Query Language for Semistructured Data. *Int. J. Digit. Libr.* 1, 1 (1997), 68–88. https://doi.org/10.1007/s007990050005

[2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *SIGMOD*. 1421–1432.

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

[4] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W. Hare, Jan Hidders, Victor E. Lee, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Josh Perryman, Ognjen Savkovic, Michael Schmidt, Juan F. Sequeda, Slawek Staworko, and Dominik Tomaszuk. 2021. PG-Keys: Keys for Property Graphs. In *SIGMOD '21: International Conference on Management of Data*. ACM, 2423–2436.

[5] Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens, and Andreas Pieris. 2021. *Principles of Databases.* Open source at https://github.com/pdm-book/community.

[6] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. 2012. Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *World Wide Web (WWW)*. 629–638.

[7] Guillaume Bagan, Angela Bonifati, and Benoît Groz. 2013. A trichotomy for regular simple path queries on graphs. In *Symposium on Principles of Database Systems (PODS)*, Richard Hull and Wenfei Fan (Eds.). ACM, 261–272.

[8] Pablo Barceló. 2013. Querying graph databases. In *Principles of Database Systems (PODS)*. 175–188.

[9] Pablo Barceló, Leonid Libkin, Anthony Widjaja Lin, and Peter T. Wood. 2012. Expressive languages for path queries over graph-structured data. *ACM Trans. Database Syst.* 37, 4 (2012), 31:1–31:46.

[10] Pablo Barceló, Leonid Libkin, and Juan L. Reutter. 2014. Querying regular graph patterns. *Journal of the ACM* 61, 1 (2014), 8:1–8:54.

[11] Mikolaj Bojanczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. 2011. Two-variable logic on data words. *ACM Trans. Comput. Log.* 12, 4 (2011), 27:1–27:26. https://doi.org/10.1145/1970398.1970403

[12] Béla Bollobás. 2013. *Modern Graph Theory.* Vol. 184. Springer Science & Business Media.

[13] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. 2008. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation. https://www.w3.org/TR/2008/REC-xml-20081126/

[14] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2000. Containment of Conjunctive Regular Path Queries with Inverse. In *Knowl. Representation & Reasoning (KR)*. 176–185.

[15] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2003. Reasoning on regular path queries. *SIGMOD Record* 32, 4 (2003), 83–92.

[16] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 1999. Rewriting of Regular Expressions and Regular Path Queries. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania, USA*, Victor Vianu and Christos H. Papadimitriou (Eds.). ACM Press, 194–204. https://doi.org/10.1145/303976.303996

[17] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. 1987. A Graphical Query Language Supporting Recursion. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*, Umeshwar Dayal and Irving L. Traiger (Eds.). ACM Press, 323–330. https://doi.org/10.1145/38713.38749

[18] Richard Cyganiak, David Wood, and Markus Lanthaler. 2014. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation. https://www.w3.org/TR/rdf11-concepts/

[19] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2019. TigerGraph: A Native MPP Graph Database. *CoRR* abs/1901.08248. arXiv:1901.08248 http://arxiv.org/abs/1901.08248

[20] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2020. 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*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 377–392. https://doi.org/10.1145/3318464.3386144

[21] Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. 1997. A Query Language for a Web-Site Management System. *SIGMOD Rec.* 26, 3 (1997), 4–11. https://doi.org/10.1145/262762.262763

[22] Diego Figueira, Adwait Godbole, Shankara Narayanan Krishna, Wim Martens, Matthias Niewerth, and Tina Trautner. 2020. Containment of Simple Conjunctive Regular Path Queries. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 371–380.

[23] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 1433–1445. https://doi.org/10.1145/3183713.3190657

[24] Alastair Green, Paolo Guagliardo, and Leonid Libkin. 2021. *Property graphs and paths in GQL: Mathematical definitions*. Technical Reports TR-2021-01. Linked Data Benchmark Council (LDBC). https://doi.org/10.54285/ldbc.TZJP7279

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

[26] William L. Hamilton. [n. d.]. Graph Representation Learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 14, 3 ([n. d.]), 1–159.

[27] Steve Harris and Andy Seaborne. 2013. SPARQL 1.1 Query Language. W3C Recommendation. http://www.w3.org/TR/sparql11-query/

[28] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutiérrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. 2021. Knowledge Graphs. *ACM Comput. Surv.* 54, 4 (2021), 71:1–71:37. https://doi.org/10.1145/3447772

[29] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Andrew J Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. 2021. Highly accurate protein structure prediction with AlphaFold. *Nature* 596, 7873 (Aug. 2021), 583–589.

[30] Property Graph Query Language. 2021. PGQL 1.4 Specification. https://pgql-lang.org/spec/1.4/

[31] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. 2016. Querying Graphs with Data. *Journal of the ACM* 63, 2 (2016), 14:1–14:53.

[32] Katja Losemann and Wim Martens. 2013. The complexity of regular expressions and property paths in SPARQL. *ACM Trans. Database Syst.* 38, 4 (2013), 24.

[33] Yao Ma and Jiliang Tang. 2021. *Deep Learning on Graphs*. Cambridge University Press.

[34] Wim Martens, Matthias Niewerth, and Tina Trautner. 2020. A Trichotomy for Regular Trail Queries. In *International Symposium on Theoretical Aspects of Computer Science (STACS) (LIPIcs, Vol. 154)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:16.

[35] Wim Martens and Tina Trautner. 2018. Evaluation and Enumeration Problems for Regular Path Queries. In *International Conference on Database Theory (ICDT) (LIPIcs, Vol. 98)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 19:1–19:21.

[36] Alberto O. Mendelzon, George A. Mihaila, and Tova Milo. 1996. Querying the World Wide Web. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, December 18-20, 1996, Miami Beach, Florida, USA*. IEEE Computer Society, 80–91. https://doi.org/10.1109/PDIS.1996.568671

[37] Alberto O. Mendelzon and Peter T. Wood. 1989. Finding Regular Simple Paths in Graph Databases. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*. 185–193.

[38] Alberto O. Mendelzon and Peter T. Wood. 1995. Finding Regular Simple Paths in Graph Databases. *SIAM J. Comput.* 24, 6 (1995), 1235–1258.

[39] Vera Zaychik Moffitt and Julia Stoyanovich. 2017. Temporal graph algebra. In *Proceedings of The 16th International Symposium on Database Programming Languages (DBPL)*, Tiark Rompf and Alexander Alexandrov (Eds.). ACM, 10:1–10:12. https://doi.org/10.1145/3122831.3122838

[40] openCypher. 2017. Cypher Query Language Reference, Version 9. https://github.com/opencypher/openCypher/blob/master/docs/openCypher9.pdf

[41] openCypher. 2021. Usage of Cypher. https://opencypher.org/projects/

[42] Eric Prud'hommeaux and Andy Seaborne. 2008. SPARQL Query Language for RDF. W3C Recommendation. http://www.w3.org/TR/rdf-sparql-query/

[43] Jonathan Robie, Michael Dyck, and Josh Spiegel. 2017. XML Path Language (XPath) 3.1. W3C Recommendation. https://www.w3.org/TR/xquery-31/

[44] Jonathan Robie, Michael Dyck, and Josh Spiegel. 2017. XQuery 3.1: An XML Query Language. W3C Recommendation. https://www.w3.org/TR/xquery-31/

[45] Christopher Rost, Kevin Gómez, Philip Fritzsche, Andreas Thor, and Erhard Rahm. 2021. Exploration and Analysis of Temporal Property Graphs. In *Proceedings of the 24th International Conference on Extending Database Technology (EDBT)*, Yannis Velegrakis, Demetris Zeinalipour-Yazti, Panos K. Chrysanthis, and Francesco Guerra (Eds.). OpenProceedings.org, 682–685. https://doi.org/10.5441/002/edbt.2021.83

[46] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. 2021. The future is big graphs: a community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71. https://doi.org/10.1145/3434642

[47] TigerGraph Team. 2021. TigerGraph Documentation – version 3.1. https://docs.tigergraph.com/

[48] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. 1–6.

[49] Peter T. Wood. 2012. Query languages for graph databases. *SIGMOD Record* 41, 1 (2012), 50–60.

This figure "sample-franklin.png" is available in "png" format from:

http://arxiv.org/ps/2112.06217v1