Design principles of property graph languages

A theoretical and experimental study

Alexandra Rogova

November 27th 2024

Data is everywhere

A large majority of it is stored in relational tables.

Sometimes the important information is not only the data itself but also how it's connected.

Connected relational data

Ingredient

ID
i1
i2
i3
i4

Recipe

ID
r1
r2
r3

Require

ID	Recipe	Ingredient
rq1	r1	i1
rq2	r1	i2
rq3	r1	i3
rq4	r3	r1
rq5	r3	i4
rq6	r3	r2

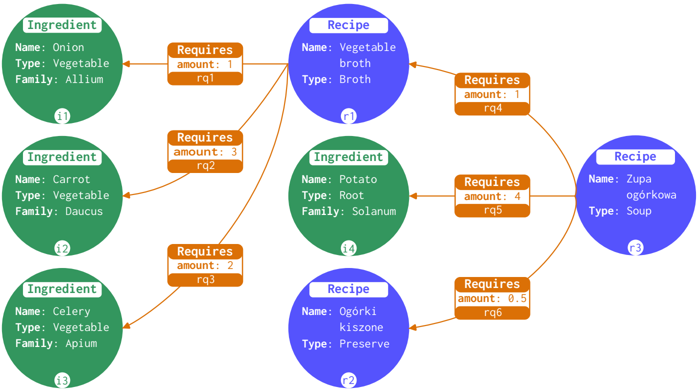
Properties

ID	Key	Value
i1	Name	Onion
i1	Type	Vegetable
i1	Famility	Alium
i2	Name	Carrot
...	...	...
r3	Name	Zupa ogórkowa
r3	Type	Soup
rq1	Amount	1
rq2	Amount	3
rq3	Amount	2
rq4	Amount	1
rq5	Amount	4
rq5	Amount	0.5

A collection of recipes encoded in the relational model.

Storing data as a graph

We can take inspiration from a natural representation:



What we have drawn is called a property graph.

What kind of questions do we want to ask about recipes?

1. Are there nuts in a zupa ogórkowa?

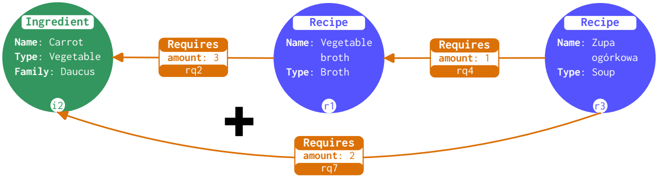
i.e. find a path from the recipe to a node of type “Nut”.

This is a reachability query.

Querying data

What kind of questions do we want to ask about recipes?

2. How many carrots are required to make a zupa ogórkowa?



i.e. find all nodes with name "Carrot" reachable from the recipe and sum the appropriate amounts.

This is a counting query.

What kind of questions do we want to ask about recipes?

3. What recipes can I make with this list of ingredients?

i.e. find a recipe node that can reach all ingredient nodes.

This is a back-and-forth reachability query.

Querying data

What kind of questions do we want to ask about graphs?

- Reachability
- Counting
- Back-and-forth reachability
- Many others (find cliques, covering, network flows, ...)

Can we write them in the standard relational query language SQL?

Technically yes.

Unweighted shorted path query in SQL:

```
WITH RECURSIVE paths(startNode, endNode, path, level, endNodeReached) AS (
    SELECT node1id AS startNode, node2id AS endNode,
          [node1id, node2id]::bigint[] AS path, 1 AS level,
          max(CASE WHEN p2.Type = 'Nut' THEN true ELSE false END) OVER
          (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS
          endNodeReached
    FROM requires
    JOIN Node n1 ON n1.id = requires.node1id
    JOIN Node n2 ON n2.id = requires.node2id WHERE n1.name = 'Zupa ogórkowa'
    UNION ALL
    SELECT paths.startNode AS startNode, node2id AS endNode,
           array_append(path, node2id) AS path, level + 1 AS level,
           max(CASE WHEN p2.Type = 'Nut' THEN true ELSE false END) OVER
           (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS
           endNodeReached
    FROM paths
    JOIN requires ON paths.endNode = requires.node1id
    JOIN Node n2 ON n2.id = requires.node2id WHERE n2.id != ALL(paths.path)
    AND NOT paths.endNodeReached)
SELECT path, level, endNodeReached AS recipe
FROM paths;
```

From Peter Boncz's tutorial at EDBT 2022

To avoid this complexity, graph specific query languages were developed.

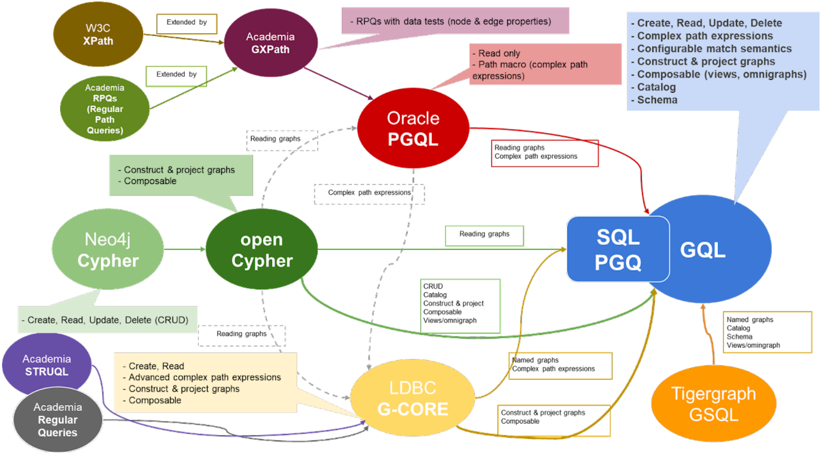
The most famous is Neo4j's Cypher.

The same unweighted shortest path query in Cypher:

```
MATCH p = ALL SHORTEST
          ({type: "Nut"})<-[:Requires*]-({name: "Zupa ogórkowa"})
RETURN p
```

This way of describing a graph shape is called pattern matching.

Other graph query languages



Standard graph query languages

In 2019, the International Organization for Standardization (ISO) decided to standardize the way we query graphs.

Two new languages were created:

- SQL/PGQ, an extension of SQL to query graphs stored in relational tables (the usual approach) and
- GQL, a brand new language completely separate from the relational model (this has never happened before)

Pattern matching is the same in GQL and SQL/PGQ.

Inspecting GQL and SQL/PGQ

While it's good that we now all speak the same language, the main question of this thesis was

Are GQL and SQL/PGQ good graph languages?

In particular:

- Can we write all common graph queries?
- Can we write them easily (unlike in SQL)?
- Is the time and space complexity of query evaluation sensible?
- Does it work fast enough in practice?

Case study : increasing path

Consider the following query:

"Match all paths s.t. the values along the edges are increasing."

I claim that

Theorem 1

1. There is no way to write it in graph pattern matching.
2. There is no way to write it GQL or SQL/PGQ such that it works in pratice.

How do we prove it?

The property graph model

A **property graph** is composed of:

Nodes

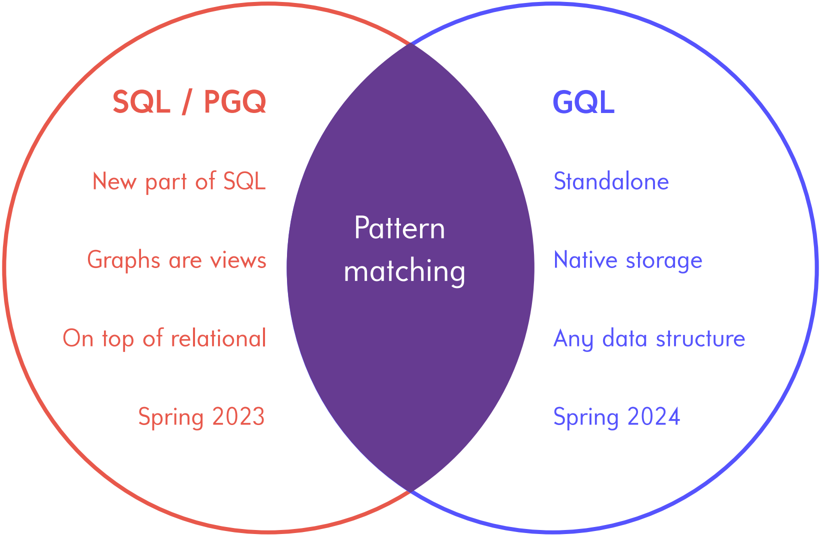
Directed edges

Undirected edges

Labels on both nodes and edges

Properties (key-value pairs) on both nodes and edges

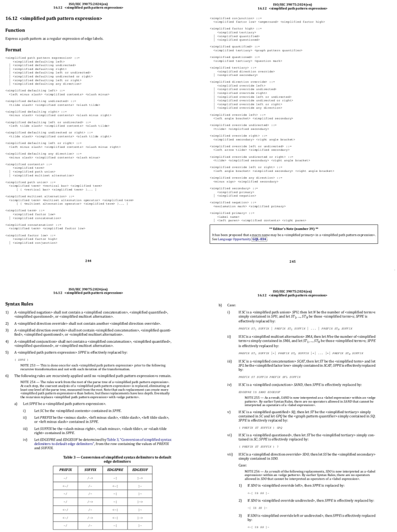
Formalizing GQL and SQL/PGQ



Formalizing GQL and SQL/PGQ

Both GQL and SQL/PGQ are

- Expensive ISO standards
- Hundreds of pages of formal grammar
- With pseudo-code semantics
- Trying to convey a simple Cypher-like syntax with all encompassing semantics



Definition of a "simplified path pattern expression"

Formalizing GQL and SQL/PGQ

We[1] read through the standards and produced:

- An explanation of GQL for the research community (*A Researcher's Digest of GQL*, ICDT 2023)
- A calculus that reflects all key pattern matching features of GQL (*GPC: A Pattern Calculus for Property Graphs*, PODS 2023)
- Two formal languages for Core GQL and Core PGQ that link them both to classical relational algebra (*GQL and SQL/PGQ: Theoretical Models and Expressive Power*[2], submitted to VLDB 2025)

[1] N. Francis, A. Gheerbrant, P. Guagliardo, L. Libkin, V. Marsault, W. Martens, F. Murlak, L. Peterfreund, A. R., D. Vrgoc

[2] A. Gheerbrant, L. Libkin, L. Peterfreund, A.R.

Core GQL/Core PGQ patt. match. syntax

A **pattern matching** expression is obtained from:

$$\phi \;:=\; \underbrace{([x])}_{\text{nodes}} \;\mid\; \overbrace{\xrightarrow{[x]} \;\mid\; \xleftarrow{[x]}}^{\text{edges}} \;\mid\; \underbrace{\phi_1\,\phi_2}_{\text{concatenation}} \;\mid\; \overbrace{\phi^{n..m}}^{\text{repetition}} \;\mid\; \underbrace{\phi_1 + \phi_2}_{\text{union}} \;\mid\; \overbrace{\phi\langle\theta\rangle}^{\text{conditions}}$$

where conditions θ are given by

$$\theta, \theta' \;:=\; \underbrace{x.k = x'.k'}_{\text{equality}} \mid \underbrace{x.k < x'.k'}_{\text{inequality}} \mid \underbrace{\ell(x)}_{\text{label}} \mid \underbrace{\theta \vee \theta' \mid \theta \wedge \theta' \mid \neg\theta}_{\text{boolean comb.}}$$

Core GQL/Core PGQ patt. match. semantics

The result of evaluating a pattern matching expression on a graph is a table called the driving table.

A driving table contains the graph elements assigned to each variable for all the answers to the query.

$$\Big[\!\Big[(x) \xrightarrow{y} (z)\langle x.Type = \text{"}Broth\text{"}\rangle\Big]\!\Big]_{\text{Recipes}} =$$

x	y	z
r1	rq1	i1
r1	rq2	i2
r1	rq3	i3

where `r1` is the vegetable broth node, `i1` the onion node, `i2` the carrot node and `i3` the celery node.

Inexpressibility of increasing path

Finding increasing node values is simple:

$$((x) \to (y)\langle x.k < y.k \rangle)^{0..\infty}$$

But if we try the same for edges:

$$\left(() \xrightarrow{x} () \xrightarrow{y} ()\langle x.k < y.k \rangle\right)^{0..\infty}$$

Some paths are wrongfully matched, e.g. $\bigcirc \xrightarrow{3} \bigcirc \xrightarrow{4} \bigcirc \xrightarrow{1} \bigcirc \xrightarrow{2} \bigcirc$

In a repeated pattern, only the last node is kept between iterations!

Proving inexpressibility of increasing path

Lemma: $\phi^{2..2}$ accepts the same paths as $\phi\phi$.

Proof idea of Theorem 1.1: Imagine there is a query Q that returns exactly all paths with increasing values in edges in any graph.

By definition, Q must accept the path $\bigcirc \xrightarrow{1} \bigcirc \xrightarrow{2} \bigcirc \xrightarrow{3} \bigcirc$

But since Q necessarily contains repetition, by the lemma above extended to arbitrary repetition, it must also accept the path [1]

$$\bigcirc \xrightarrow{1} \bigcirc \xrightarrow{2} \bigcirc \xrightarrow{3} \bigcirc \xrightarrow{1} \bigcirc \xrightarrow{2} \bigcirc \xrightarrow{3} \bigcirc$$

As this path is not strictly increasing, we have a contradiction and so Q cannot exist.

[1] This is true only for pattern matching without $\leftarrow$

Real-life GQL and SQL/PGQ

GQL and SQL/PGQ are not limited to just pattern matching.

In fact, both languages can be described as

Calls to pattern matching within a variant of relational algebra

The increasing path query *could* be encoded as

(all paths) $\setminus$ (paths in which two consecutive values are decreasing)

But this forces the GDBMS to execute an inneficient algorithm:

first find all paths, then remove those that do not comply.

Graph queries in practice

Many graph problems are notoriously hard

- Finding a simple path/trail is NP-hard
- finding all maximal cliques requires exponential time!

Can graph systems execute these queries efficiently?

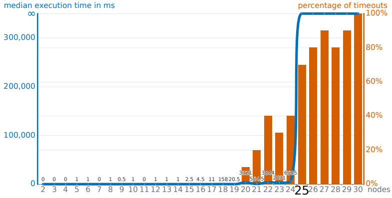
To find out we tested if this is the case for increasing path,we tested Neo4j's performance on random graphs generated with n nodes and probability p of any two nodes being connected.

We measured the median running time on 10 graph for each configuration, with a timeout set to 5 minutes.

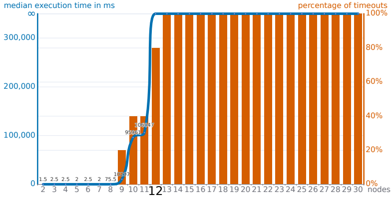
Neo4j performance – Increasing path

Cypher does not include the “\” operator so the increasing condition is verified by a reduce (fold) function:

```
MATCH p=()-[*2..]->()
WITH p, reduce(
    acc=relationships(p)[0].val,
    v in relationships(p) |
    CASE WHEN acc=-1 THEN -1
         WHEN v.val>=acc THEN v.val
         ELSE -1
    END) AS inc
WHERE NOT inc = -1
RETURN p
```



(a) $p = 0.1$



(b) $p = 0.3$

Fixing the increasing path query

Pattern matching cannot express the increasing path query.

We have to use the “\” operator in a way that is doomed to fail.

How can we fix this?

- Change the semantics of repetition
- Add a new "compare along the path" operator

General properties - Expressive power

The core of GQL and SQL/PGQ can be defined as relational algebra combined with pattern matching and the path restrictors simple, trail and shortest.

Theorem 2 - Expressive power

Core GQL and Core SQL/PGQ are at least as expressive as

- Unions of Conjunctions of Two-way Regular Path Queries,
- Nested Regular Expressions and
- Regular Queries.

General properties - Complexity

Definition 3 - The Enumerate Answers problem

Input: A graph G and a query Q

Output: Enumerate all answers to Q over G without repetitions.

Theorem 4 - Complexity

The complexity of GQL and SQL/PGQ is EXPSpace in combined complexity and PSpace in data complexity for the Enumerate Answers problem.

Are GQL and SQL/PGQ good?

We needed new tools to study GQL and SQL/PGQ.

We created formal models, simple yet powerful enough for thorough mathematical reasoning.

Thanks to these models we were able to formally prove the inexpressibility of the increasing path query.

We then confirmed experimentally that this is indeed a deficiency of the languages.

We also determined the expressive power of GQL and SQL/PGQ and their complexities for the Enumerate Answers problem.

What's left to do

Now that we have a defined core of pattern matching, we need to understand its behaviour:

- How complex are its queries for various problems (query answering, enumeration, containment, ...)?
- How do path or bag semantics affect this complexity?
- How far can we extend this core before its queries become intractable?

We have already identified some deficiencies,

- Are there other desirable queries that are inexpressible?
- Are some of the expressible queries dangerous?
- How can we change the core to avoid both issues?