PGShield: Maintaining Path Constraints in Property Graphs

Christopher Spinrath
Lyon 1 University, Liris CNRS
Lyon, France
christopher.spinrath@liris.cnrs.fr

Angela Bonifati
Lyon 1 University, Liris CNRS & IUF
Lyon, France
angela.bonifati@univ-lyon1.fr

Rachid Echahed
CNRS LIG, Univ. Grenoble Alpes
Grenoble, France
rachid.echahed@imag.fr

ABSTRACT

We present PGShield, the first system that prevents updates that would violate path constraints on property graphs. It eliminates the need for costly post error detection and repair, that turns out to be a time-consuming task in graph databases. Indeed, violations may appear along long and arbitrarily complex paths. PGShield extends the recently standardized GQL query language with new clauses that allow users to define path constraints, apply updates safely, and receive feedback when an update cannot be performed due to a constraint violation. Internally, PGShield annotates the graph with metadata serving as an index to detect potential violations efficiently before changes are applied. We demonstrate PGShield on real-world property graph datasets and constraints.

PVLDB Reference Format:
Christopher Spinrath, Angela Bonifati, and Rachid Echahed. PGShield: Maintaining Path Constraints in Property Graphs. PVLDB, 19(12): 4590 - 4593, 2026.
doi:10.14778/3827998.3828073

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://doi.org/10.5281/zenodo.20830104.

1 INTRODUCTION

In practice, datasets frequently contain inconsistencies and errors. Such issues may arise from various sources, including data integration processes, modifications to the database schema driven by evolving requirements, or the inadvertent insertion of incorrect data. Consequently, error repairing has been studied extensively in the literature for various database models [1].

However, databases are subject to change: users can insert or remove data, and therefore also introduce new errors. Of course, this could be addressed by running an error detection, or even error repair algorithm, after each change. But classically these algorithms consider the whole database, and running them for each change might be too costly [5]. Moreover, users should be informed as soon as possible if a change was successful or not, provided with feedback on failures, and the means to rectify the situation.

We go beyond prior work [5] for guaranteeing integrity of property graphs and demonstrate PGShield, a novel graph database extension which prevents the introduction of errors into a clean or previously repaired database, that is, a database free of errors. It implements, to the best of our knowledge, a novel index-based constraint verification procedure, not published before, to detect whether changes would lead to errors in the database, and to provide feedback to users.

PGShield uses *property graphs* as underlying data model, which is part of the recent ISO/GQL standard [3]. In a nutshell, property graphs are multi-graphs, in which every node *and* edge is associated with (possibly multiple) labels and key-value pairs, called *properties*.

PGShield supports *path constraints* which forbid certain paths from existing in the graph. For instance, consider a database which stores officers and intermediaries of companies and their connections. A user can register a constraint with PGShield stating that officers cannot be presidents of intermediaries, neither directly nor indirectly via `officer_of` edges. More precisely, PGShield extends the standardized query language GQL for property graphs with a `CREATE PATH CONSTRAINT` such that the user can register the constraint with the following query using a GQL-like syntax.

```
CREATE PATH CONSTRAINT no_president_of_intermediary DENY
  (:Officer)-[:president_of]->()-[:officer_of]->*(:Intermediary)
```

Here $*$ is the classical Kleene-star operator which allows for zero or more repetitions.

Upon initialization of the constraint, PGShield annotates each node with *metadata* indicating, among others, whether

(1) it is reachable by a path of the form
```
(:Officer)-[:president_of]->()-[:officer_of]->*()
```
(2) it can reach an `Intermediary` node using `officer_of` edges.

Users can then use PGShield's `TRY INSERT` clause to attempt to insert new `officer_of` edges:

```
MATCH (x:Officer), (y:Entity) TRY INSERT (x)-[:officer_of]->(y)
```

To verify an insertion between two nodes v_1 and v_2, PGShield uses their metadata as an index to determine whether the insertion would create a path that violates the constraint, i.e., whether (1) holds for v_1 and (2) holds for v_2. By contrast, a naive approach would apply the change and then search the database for an arbitrarily long violating path. If a violation is detected, PGShield rejects the insertion and returns informative feedback to the user. After a successful insertion, the metadata must be updated to enable validation of subsequent insertions.

Structure. Section 2 provides more details on the metadata serving as an index and how changes are verified. Section 3 explains PGShield's architecture, and its extension of GQL. Finally, we discuss the demonstration's scenarios in Section 4. More details and a video abstract are available in the supplemental material [4].

2 MAINTAINING PATH CONSTRAINTS

We discuss the constraints PGShield supports, and how it uses metadata. The underlying data model is that of property graphs:

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828073

that is, graphs consisting of nodes and directed edges, where there can be multiple edges between two nodes, each node and edge has zero or more labels, and each node and edge has zero or more properties, which are key-value pairs.

Path Constraints. PGShield supports path constraints which forbid certain paths from existing in the property graph. A path constraint is specified by a *path pattern*: Paths matching the pattern are forbidden. That is, a path matching the pattern is a *violation*.

To specify patterns, PGShield facilitates a fragment of the pattern sublanguage of GQL (and SQL/PGQ). A node can be matched by a pattern of the form `(:`φ`)` where φ is a (possibly empty) *label expression*. For example, `(:Officer)` matches all nodes labelled `Officer`, `(x:Officer&Intermediary)` all nodes labelled `Officer` as well as `Intermediary`, and `()` matches all nodes in the graph. In general, label expressions support arbitrarily nested conjunctions (`&`), disjunctions (`|`), and negations (`!`). Similarly, edge patterns have the form `-[:`φ`]->`. For example, `-[:officer_of]->` matches all edges labelled `officer_of`. Node and edge patterns can be combined into *path patterns* using *concatenation*, and the *Kleene-star* operator (for repetitions). For details on the pattern language we refer to [5], where they are called RGPC patterns (without unions).

We note that PGShield does not support constraints involving properties. Our demonstration scenarios (cf. Section 4) show that meaningful constraints can already be expressed without them.

Automata. PGShield uses finite-state automata for path patterns to compose the metadata and verify changes. Such an automaton reads nodes and edge labels in alternation, and recognizes precisely the paths matched by the pattern. For details, we refer again to [5].

Metadata. To speed up the validation of insertions, PGShield annotates each node and edge with metadata. For each constraint, the metadata of a node v consists of two sets S_v^{in} and S_v^{out}. Both are sets of states of the automaton for the constraint's pattern, and satisfy the following conditions. A state s is in S_v^{in} if and only if there is a path ending in v such that the automaton can be in state s after reading the labels of the path (starting in its initial state). Similarly, a state s is in S_v^{out} if and only if there is a path *starting* in v such that the automaton can be in an accepting state after reading the path starting in state s. The metadata for edges is defined analogously. This allows PGShield to quickly assert if an insertion would lead to a new violation: Suppose the user wants to insert an edge e between two nodes v and w. PGShield then checks whether the automaton can transition from a state in S_v^{in} to a state in S_w^{out} by reading the label(s) of e.[1] If this is the case, inserting e would result in a path accepted by the automaton, and thus a violation of the constraint. Otherwise, e can be inserted since it does not result in a violation. We emphasize that PGShield can thus verify quickly if an insertion does not violate a constraint solely by using the metadata stored at the endpoints of the new insertions. It does not have to scan the graph for potentially arbitrary-length paths. Finally, we note that deletions are always successful, since they cannot create new paths.

Computing and Updating Metadata. When a new constraint is registered for an existing graph, PGShield computes the metadata in an iterative process. The sets S_v^{in} are initialized with the states the automaton can transition into from its initial state by reading the label(s) of v, i.e. with a single transition. Then the sets S_e^{in} of outgoing edges e of a node v are updated by adding all states the automaton can assume by reading the label(s) of e starting in a state from S_v^{in}. Next, the sets S_v^{in} for nodes are updated similarly by using the sets S_e^{in} of incoming edges. This process is iterated until no state is added to any set anymore. The sets S_v^{out} are computed analogously, but the initial and accepting state of the automaton are swapped, and all transitions of the automaton and edges of the graph are taken in reverse direction.

Upon successful insertions, the iterative process is restarted but (initially) only for newly inserted nodes and edges and their neighbourhood in the graph. When deleting nodes or edges, PGShield uses a similar iterative process to test whether a state s can still be in a set S_v^{in} or S_v^{out}, and if not, removes s.

More details on metadata as well as on computing and updating it are available in the supplementary material [4].

3 SYSTEM ARCHITECTURE

PGShield is open-source [4] and implemented in Kotlin. It extends the fragment of GQL supported by the Neo4j graph database system with a new `TRY INSERT` clause that allows for specifying – in a declarative fashion – insertions, which are only performed when doing so does not lead to the violation of a path constraint. Moreover, it provides the means to manage, i.e. add, remove, list, or inspect, path constraints. It has been tested in conjunction with Neo4j 5.26, the vendor's web interface and Python driver. In the following, we present PGShield's components as shown in Figure 1 by explaining PGShield's three major workflows: creating path constraints, initializing metadata, and safely inserting nodes and edges.

Creating Path Constraints. PGShield extends GQL with a `CREATE PATH CONSTRAINT` clause, which can be used to create new path constraints, using any Neo4j client. However, instead of connecting directly to Neo4j, clients connect to PGShield's *rewrite proxy*. It rewrites PGShield's GQL extensions into custom procedure calls such that Neo4j can forward them to the other components which run as a plug-in within the Neo4j DBMS. For example,

```
CREATE PATH CONSTRAINT no_president_of_intermediary DENY
  (:Officer)-[:president_of]->()-[:officer_of]->*(:Intermediary)
```

is rewritten into the following call.

```
CALL pgshield.createConstraint("no_president_of_intermediary",
 "(:Officer)-[:president_of]->()-[:officer_of]->*(:Intermediary)")
```

The `CALL` clause will invoke PGShield's *automaton builder*, which uses the *path pattern parser* to build an automaton for the constraint. Automaton, pattern, and constraint name are then written to the persistent *constraint store* component.

Initializing Metadata. After creating a new constraint for an existing property graph, its metadata has to be initialized. In our example, this is achieved by submitting the following query.

```
INIT PATH CONSTRAINT no_president_of_intermediary
```

As before, the *rewrite proxy* will rewrite this clause into a procedure call. PGShield will then lookup the constraint's automaton in the *constraint store*, and then invoke the *metadata updater* component for the constraint with all nodes and edges marked as newly inserted. The component will then run the iterative procedure

[1] When inserting longer paths p, the automaton reads all the labels of p.

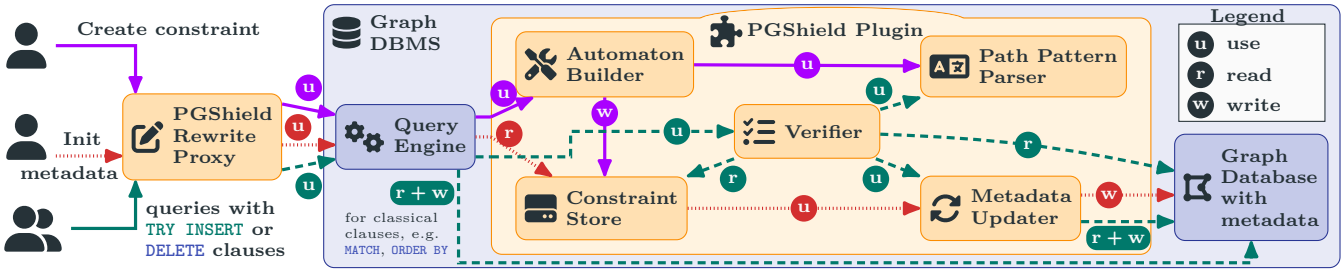

Figure 1: Architecture of the system with three possible workflows: creating a path constraint (solid edges), initialising metadata (dotted edges), and using `TRY INSERT` clauses (dashed edges). Components of the system are orange, third-party components blue.

discussed in Section 2 to compute the metadata, and store it as properties in the property graph.

Inserting Nodes and Edges. After the creation and initialization of (possibly multiple) constraints, users can enjoy PGShield's main feature: safely inserting new nodes and edges with the new `TRY INSERT` clause. Once again, `TRY INSERT` clauses are intercepted by the *rewrite proxy* and rewritten into custom procedure calls. However, since `TRY INSERT` clauses can be mixed with existing GQL clauses like `MATCH`, the rewriting is more complex. For example, the query

```
MATCH (x:Officer), (y:Intermediary) TRY INSERT (x)-[:officer_of]->(y)
```

which attempts to insert `officer_of`-edges is rewritten into

```
MATCH (x:Officer), (y:Intermediary)
WITH collect({x: x, y: y}) AS var_bindings
CALL pgshield.tryInsert("(x)-[:officer_of]->(y)", var_bindings)
YIELD out_var_bindings
WITH out_var_bindings["x"] AS x, out_var_bindings["y"] AS y
RETURN x, y
```

`TRY INSERT` accepts arbitrary path patterns specifying the insertion and, unlike for constraints, they can contain variables from previous clauses to refer to existing nodes.

Calling the `tryInsert` procedure invokes PGShield's *verifier*, which reads all constraints from the store, and uses the metadata as explained in Section 2 to quickly verify whether the insertions induced by the pattern and variable bindings would violate any of them. If no violation is detected, the insertions are realized, the *metadata updater* updates the metadata, and the variable bindings are returned such that the user can reuse them in a follow-up query.

Advanced Insertions and Deletions. The full syntax of `TRY INSERT` is `TRY INSERT` <mode> <path pattern> `EXCEPT` where <path pattern> describes the path (fragments) to be inserted. By default either all or none of the paths are inserted. The user can change this behaviour by specifying another <mode>: `KEEP` inserts paths sequentially one after the other, and stops once an insertion would lead to a path constraint violation – already inserted paths are kept. `CONTINUE` proceeds similarly, but skips insertions which would result in violations, instead of stopping completely. In both cases the order of insertion can be controlled by a preceding `ORDER BY` clause. When the optional `EXCEPT` keyword is specified `TRY INSERT` will return all variable bindings for which the insertion failed. The user can then inspect them or perform another action to rectify the situation.

GQL's `DELETE` keyword is also intercepted by the *rewrite proxy* to update the metadata after deletions (recall that they cannot fail). The current version of PGShield does not support the addition or the removal of labels on existing nodes. However, this could be implemented by combining the iterative algorithms for deletions and insertions (with updated labels) of nodes.

Constraint Management. Besides the clauses for creating and initializing constraints above, PGShield also provides the clauses `SHOW PATH CONSTRAINTS`, `DROP PATH CONSTRAINT` <name>, as well as `RESET PATH CONSTRAINT` <name> for listing all constraints, deleting constraints, and deleting the metadata of a constraint, respectively.

Advantages of the Architecture. Thanks to the proxy component, PGShield's API can be exposed as a natural extension of GQL, and is transparent for clients: The user may freely choose any Neo4j client. Running as a plug-in allows for implementing the iterative update algorithm for the metadata directly on the property graph, and without back-and-forth communication with Neo4j via queries.

4 DEMONSTRATION SCENARIOS

To demonstrate the novelty, responsiveness, and applicability of PGShield, we provide multiple scenarios for two real-world datasets: A property graph of the International Consortium of Investigative Journalists (ICIJ) [2] with 550600 nodes and 999353 edges, and a Code Property Graph [6] representing the source code of the GNU Coreutils project with 206506 nodes and 387284 edges.

For every scenario we provide a *Jupyter notebook* which describes the property graphs and constraints, and guides users through the workflows discussed in Section 3, while letting the user track how PGShield works internally. Moreover, the user has access to Neo4j's *interactive web interface* to monitor changes, inspect the metadata, and resolve detected violations. Excerpts from both are shown in Figure 2. Each scenario highlights different aspects of the system.

In the following we describe one scenario on the ICIJ graph in detail, and then discuss the others. The nodes of the property graph describe, among others, officers, intermediaries, and entities, e.g., companies or organisation. The edges describe relationships between them, e.g. `officer_of` or `president_of`. We note that entities can also be officers of other entities or even of officers or intermediaries, even though they are not labelled as officer.

Step 1: The user can discover the property graph, it's schema, and possible constraints using the notebook and the web interface.

Step 2: The user can then choose a constraint, optionally modify it, and create, i.e. register, it. For example, the following query creates a constraint stating that officers cannot be presidents of intermediaries, neither directly nor indirectly via `officer_of` edges.

```
CREATE PATH CONSTRAINT no_president_of_intermediary DENY
  (:Officer)-[:president_of]->()-[:officer_of]->*(:Intermediary)
```

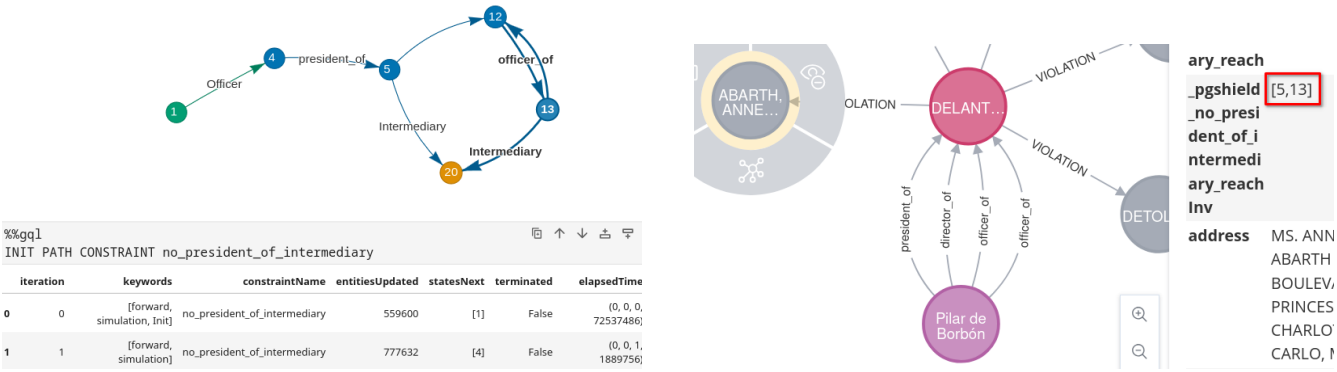

Figure 2: Screenshots of a Jupyter notebook (left) and the interactive web interface (right) users can interact with.

To assert that the constraint is registered, the user can also list all constraints via `SHOW PATH CONSTRAINTS`. This query will also provide some information on the automaton PGShield constructed for the constraint, and the automaton can be visualized within the notebook using a Python helper function, as shown in Figure 2.

Step 3: Next, the constraint has to be initialized by running `INIT PATH CONSTRAINT no_president_of_intermediary`. Afterwards, we show plots in the notebook which visualize the runtime and number of metadata updates for each iteration of the metadata computation. Moreover, the user can inspect the metadata in the web interface, and trace how it was computed with the help of the automaton. For example, Figure 2 shows the metadata $S_v^{\text{out}} = \{5, 13\}$ for an intermediary node involved in a violation on the right-hand side.

Step 4: We demonstrate PGShield's ability to prevent insertions that would result in a constraint violation with the following query, which intentionally provokes the insertion of violations.

```
MATCH (:Officer)-[:president_of]->(s:Entity),
      (t:Intermediary) WHERE s <> t
TRY INSERT (s)-[:officer_of]->(t)
RETURN COUNT(*) AS num_successful_insertions
```

Running the query returns a count of 0 successful insertions, which the user can verify in the web interface. Users can also inspect which and *why* insertions were prevented by inserting `VIOLATION` edges instead, using PGShield's `EXCEPT` clause.

```
MATCH (:Officer)-[:president_of]->(s:Entity),
      (t:Intermediary) WHERE s <> t
TRY INSERT (s)-[:officer_of]->(t)
EXCEPT TRY INSERT (s)-[:VIOLATION]->(t)
RETURN COUNT(*) AS num_failed_insertions
```

The user can now visualize `VIOLATION` edges and their neighbourhood to see that inserting `officer_of` edges would indeed have resulted in a path violating the constraint. Furthermore, they can reconstruct with the help of the metadata and the automaton how PGShield derived this quickly (cf. the right-hand side of Figure 2).

Step 5: We illustrate that PGShield realizes, by default, either all insertions or none by providing a query that matches pairs of nodes for which inserting an edge between them leads to a violation, and pairs for which inserting an edge is unproblematic. We then introduce PGShield's `KEEP` mode, which verifies and realizes insertions sequentially until a violation is detected, keeping insertions performed before encountering the violation.

```
CALL () {
  MATCH (:Officer)-[:president_of]->(s:Entity),
        (t:Intermediary) WHERE s <> t LIMIT 25 RETURN s, t
  UNION MATCH (s:Entity), (t:Officer) LIMIT 50 RETURN s, t
} ORDER BY t.name DESC
TRY INSERT KEEP (s)-[:officer_of]->(t)
RETURN COUNT(*) AS num_successful_insertions
```

As before, the user can inspect and trace PGShield with the help of the notebook, and the web interface. Furthermore, they can change the `ORDER BY` clause in the query above and observe how this changes the number of realized insertions. The user can also replace the `KEEP` with the `CONTINUE` mode (cf. Section 3), and compare them.

Other Scenarios. We also provide scenarios with multiple constraints, and with an increasing number of (attempted) insertions. The user can view the runtime and updates per iteration for insertions as well, and can compare them to the corresponding numbers for the initialisation. The Code Property Graph, whose nodes represent source code fragments, admits more involved constraints with multiple recursions within the same pattern, and more complex insertions, e.g. the addition of new code. Of course, users are free to adapt any constraints and queries, or even create their own.

ACKNOWLEDGMENTS

The authors were supported by ANR-21-CE48-0015 VeriGraph. A. Bonifati is also funded by IUF Endowed Chair.

REFERENCES

[1] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. 2016. Data Cleaning: Overview and Emerging Challenges. In *Proceedings of the 2016 International Conference on Management of Data* (2016-06) *(SIGMOD/PODS'16)*. Association for Computing Machinery (ACM), San Francisco California USA, 2201–2206. https://doi.org/10.1145/2882903.2912574

[2] International Consortium of Investigative Journalists. 2026. ICIJ Offshore Leaks Database. https://offshoreleaks.icij.org/pages/database. Accessed: 2026-01-26.

[3] ISO Central Secretary. 2024. *Information Technology — Database Languages — GQL* (1 ed.). Standard ISO/IEC 39075:2024. International Organization for Standardization, Geneva, CH.

[4] Christopher Spinrath, Angela Bonifati, and Rachid Echahed. 2026. *PGShield: Maintaining Path Constraints in Property Graphs [Artifacts]*. Zenodo. https://doi.org/10.5281/zenodo.20830104

[5] Christopher Spinrath, Angela Bonifati, and Rachid Echahed. 2026. Repairing Property Graphs under PG-Constraints. *Proceedings of the VLDB Endowment* 19, 6 (May 2026), 1212–1225. https://doi.org/10.14778/3797919.3797929

[6] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy* (2014-05). IEEE, San Jose, CA, 590–604. https://doi.org/10.1109/SP.2014.44