**ProGQL Technical Report**

**1. Introduction**

System audit logs record system calls at the kernel level and can be used to construct provenance graphs (PGs) that reveal causal dependencies between processes, files, and network connections. Provenance analysis (PA) leverages these graphs to investigate suspicious activities by tracing dependencies from a point of interest (POI), such as a malware execution or anomalous network connection. PA has been shown effective for investigating **advanced persistent threats (APTs)**, where multi-stage and multi-host behaviors are correlated into an attack chain.

However, existing PA approaches face two key limitations. First, they are prone to **dependency explosion**, where the happen-before relation expands into millions of irrelevant edges, overwhelming analysts. Although heuristics such as edge weighting have been proposed, these are typically **hard-coded and inflexible**, preventing adaptation to diverse attack scenarios. Second, current techniques lack effective means to **incorporate expert knowledge** into the analysis. Analysts often need to experiment with weight functions, filtering conditions, and cross-host correlation, yet current tools offer no explicit language support.

To address these challenges, we developed **ProGQL**, a domain-specific language (DSL) and system for provenance analysis. ProGQL’s has two major contributions:

1. **A novel DSL for provenance analysis.**  
   ProGQL introduces constructs for recursive, constraint-based graph search, edge weight assignment, impact score propagation, and graph merging. These operators allow analysts to declaratively encode domain knowledge into queries, enabling fine-grained control over provenance graph construction.
2. **A well-engineered system for scalable query execution.**  
   The ProGQL engine supports efficient execution of DSL queries through incremental graph traversal, edge merging, iterative score propagation, and optimized graph composition. To ensure portability, we designed the system using the **Factory Pattern**, enabling seamless support for multiple database backends (PostgreSQL, MyRocks, MariaDB, Neo4j, Nebula).

Together, these contributions allow ProGQL to generate focused provenance graphs, reduce false positives, and scale to tens of millions of events. This technical report complements our ICDE 2026 submission by providing (1) a **formal definition of ProGQL’s language constructs**, (2) detailed case studies demonstrating its **expressiveness and effectiveness**, and (3) algorithmic descriptions of key components.

**2. Formal Language Definition**

**2.1 Grammar and Language Design**

The full grammar is available in our repository:

* [appendix/ProGQL.g4](https://github.com/ProGQL/ProGQL/blob/main/appendix/ProGQL.g4)

**Grammar 1** shows the representative [BNF grammar](https://github.com/ProGQL/ProGQL/blob/main/appendix/grammar.png)

ProGQL extends Cypher with novel operators. The grammar specifies both the adapted constructs (e.g., MATCH, WHERE, RETURN) and the **new operators introduced in ProGQL**, which include:

* **Constrained graph traversals**: <Bfs> and <Dfs> operators combined with <Backward> and <Forward> explicitly define traversal strategies and directions. These operators enable recursive graph searches with constraints to filter disqualified edges at each step. This functionality is not supported in Cypher’s path semantics. Cypher (even with APOC) remains **path-oriented** and cannot express iterative traversals that repeatedly expand from an already discovered edge set until all adjacent edges are explored.
* **Aggregation over traversals**: <Max>, <Min>, and <Collect> expressions allow recursive constraints like *“continue backward only if r.starttime < max(outgoing edge endtimes)”*. These constraints cannot be expressed in Cypher, as it treats paths as static results and cannot model edges produced by dynamically constrained traversals.
* **Graph restructuring operators**: UNWIND and YIELD were adapted and extended to materialize edge sets as variables and to iteratively refine the analysis. This gives analysts explicit control over intermediate graph states.
* **Edge weighting and Impact score propagation**: SET e.weight = projection(…) and SET u.rel = reduce(…) introduce language-level constructs for assigning edge weights from feature projections and propagating scores across edges. These are new operators that capture the weight computation and impact propagation steps that provenance analysis implemented only procedurally.
* **Graph merge**: union and intersect are introduced at the query level, enabling analysts to directly combine multiple provenance subgraphs (e.g., backward and forward analyses) within a single query, rather than requiring external post-processing.

UNION(g1, g2) → merges the nodes and edges of both subgraphs into one.

INTERSECT(g1, g2) → keeps only the overlapping nodes/edges.

In Cypher, UNION merges **tabular query results**, not graphs. Cypher doesn’t have INTERSECT.

* **Entry points identify:** ProGQL identifies top-ranked candidate entry points that are selected as seeds for forward exploration by WITH entry = (MATCH n in nodes(r) WHERE count(in(n))=0 ORDER BY n.rel DESC LIMIT 15)

ProGQL adapts and extends:

**WITH entry = (…)** →

* Cypher has WITH, but in Cypher it is only used to pass intermediate variables between clauses in a linear pipeline.
* ProGQL **extends** WITH to allow *naming a subgraph expression* (like g1, entry) so it can be reused as input to further traversals. This is more like *binding a graph object*, not just passing a column.
* This is crucial for graph composition operations (like union, intersect) that Cypher cannot express directly.

**nodes(r)** →

* Cypher has nodes(path), which returns all nodes along a path.
* ProGQL generalizes this: here r is not a path but an edge set (r IN backward(f)), and nodes(r) means *collect all endpoint nodes of the edge set*.
* This adapts Cypher’s path-based function into an **edge-set–aware operator**, specific to provenance analysis.

**count(in(n))** →

* Cypher has inDegree(n), but it only counts all incoming relationships regardless of type or semantics.
* ProGQL redefines this in provenance terms: count(in(n))=0 is a logical filter: pick candidate boundary nodes in the backward graph. The notion of an “entry node” in provenance analysis is not strictly equivalent to a graph-theoretic node with zero in-degree. Rather, it is defined logically as the boundary where external influence first enters the system.

**LIMIT** →

* Cypher’s LIMIT only restricts how many rows are returned in a tabular output.
* ProGQL adapts it to **control forward exploration seeds**: after ranking by n.rel (impact score), pick the top-k entry nodes to propagate forward.
* This is a **semantic extension**, not just syntax reuse.

A white sheet with black and white text

AI-generated content may be incorrect.

**2.2 Novelty Beyond Existing Languages**

* **Recursive, edge-dependent search** cannot be expressed in Cypher or AIQL/SAQL.
* **Weighting and propagation operators** elevate procedural heuristics into first-class language constructs.
* **Graph composition operators** (union, intersect) allow analysts to declaratively combine analyses.

These features make ProGQL a true **provenance-aware DSL**, not syntactic sugar over existing languages.

**2.3 Case Study: Password Crack Attack**

Using a password cracking case study, we next illustrate, step-by-step, how ProGQL queries can be formulated in an interactive query style to perform provenance analysis for attack investigation.

**Attack description**

By exploiting the Shellshock vulnerability, the attacker is able to gain initial access to host1. Upon successful compromise, the attacker establishes a reverse shell connection to remotely control host1. In this stage, the attacker generally takes a series of stealthy reconnaissance maneuvers. Among those, we emulate the password cracking attack. The attacker downloads and executes a malicious script **gather\_password.sh**. This script identifies victim hosts (i.e., host2) and downloads another malicious script **crack\_passwd.sh**, transfers it to host2 and executes it. **crack\_passwd.sh** then downloads a series of files, including a malicious payload **libfoo.so** from the attack server. **libfoo.so** cracks passwords on the victim host. The resulting **password\_crack.txt** file contains plaintext passwords, which is then transferred to host1 and compressed as the **/tmp/passwords.tar.bz2**. This file serves as a consolidated package of sensitive information, ready for exfiltration.

The provenance graph below only shows critical edges:

A diagram of a diagram

AI-generated content may be incorrect.

**Step 1: Backward BFS from the POI**

Starting from the point of interest (POI), /tmp/passwords.tar.bz2 on host1, ProGQL performs a backward BFS with temporal constraints:

MATCH (p:Process)-[st:FileEvent{id:15035}]->(f:File{name:"/tmp/passwords.tar.bz2", hostid:"1"})

        BFS (r IN backward(f) | MATCH v=dst(r) Where r.starttime<max(collect(vout IN out(v) | vout.endtime))) YIELD g1

        RETURN g1

* This query produces a provenance graph with **154 vertices and 3117 edges**.
* After CPR (ProGQL’s built-in critical path reduction), the graph is reduced to **154 vertices and 332 edges**, eliminating redundant paths while retaining attack-relevant ones. - Should I mention cpr is a built-in process?

<https://github.com/ProGQL/ProGQL/blob/main/technical%20report/Password%20Crack/neo4jpwd-back1.svg>

**Step 2: Edge weighting and impact score propagation**

We next assign weights and propagate impact scores across the backward graph:

MATCH (p:Process)-[st:FileEvent{id:15035}]->(f:File{name:"/tmp/passwords.tar.bz2", hostid:"1"})

        BFS (r IN backward(f) | MATCH v=dst(r) Where r.starttime<max(collect(vout IN out(v) | vout.endtime))) YIELD g1

        UNWIND g1 AS e

        SET e.weight=projection(1/(abs(r.amount-st.amount)+0.0001),ln(1+1/abs(r.endtime-st.endtime)),count(out(v))/count(in(v)))

        MATCH u=src(e) SET u.rel=reduce(sum = 0, o IN out(u) | sum+o.weight\*dst(o).rel)

        RETURN g1

* This step highlights critical dependencies by assigning high scores to influential nodes.

<https://github.com/ProGQL/ProGQL/blob/main/technical%20report/Password%20Crack/neo4jpwd-back-weight1.svg>

**Step 3: Backward + forward analysis with entry selection**

ProGQL then selects the top-ranking candidate entry nodes and performs forward BFS from them. Finally, it intersects the backward and forward graphs to isolate the most relevant provenance subgraph:

MATCH (p:Process)-[st:FileEvent{id:15035}]->(f:File{name:"/tmp/passwords.tar.bz2", hostid:"1"})

        BFS (r IN backward(f) | MATCH v=dst(r) Where r.starttime<max(collect(vout IN out(v) | vout.endtime))) YIELD g1

        UNWIND g1 AS e

        SET e.weight=projection(1/(abs(r.amount-st.amount)+0.0001),ln(1+1/abs(r.endtime-st.endtime)),count(out(v))/count(in(v)))

        MATCH u=src(e) SET u.rel=reduce(sum = 0, o IN out(u) | sum+o.weight\*dst(o).rel)

        RETURN g1

intersect

WITH entry = (MATCH n in nodes(r) WHERE count(in(n))=0 ORDER BY n.rel DESC LIMIT 15)

        BFS (re IN forward(entry) | MATCH u=src(re) Where re.endtime>min(collect(uin IN in(u) | uin.starttime)) and re.starttime<1724731846719889370) yield g2

        RETURN g2

* The resulting subgraph has **39 vertices and 86 edges**.

A diagram of a flowchart

AI-generated content may be incorrect.

It also can be found at <https://github.com/ProGQL/ProGQL/blob/main/technical%20report/Password%20Crack/neo4jpwd-vm1.svg>

**Step 4: Multi-host correlation**

Repeating the same analysis on host2 and unioning the graphs from host1 and host2 yields a concise multi-host attack provenance graph:

MATCH (p:Process)-[st:FileEvent{id:15035}]->(f:File{name:"/tmp/passwords.tar.bz2", hostid:"1"})

        BFS (r IN backward(f) | MATCH v=dst(r) Where r.starttime<max(collect(vout IN out(v) | vout.endtime))) YIELD g1

        UNWIND g1 AS e

        SET e.weight=projection(1/(abs(r.amount-st.amount)+0.0001),ln(1+1/abs(r.endtime-st.endtime)),count(out(v))/count(in(v)))

        MATCH u=src(e) SET u.rel=reduce(sum = 0, o IN out(u) | sum+o.weight\*dst(o).rel)

        RETURN g1

intersect

WITH entry = (MATCH n in nodes(r) WHERE count(in(n))=0 ORDER BY n.rel DESC LIMIT 15)

        BFS (re IN forward(entry) | MATCH u=src(re) Where re.endtime>min(collect(uin IN in(u) | uin.starttime)) and re.starttime<1724731846719889370) yield g2

        RETURN g2

UNION

(MATCH (p:Process)-[st:NetworkEvent{id:100005}]->(f:Network{srcip:"192.168.1.128/32",dstip:"192.168.1.131/32",hostid:"2"})

        BFS (r IN backward(f) | MATCH v=dst(r) Where r.starttime<max(collect(vout IN out(v) | vout.endtime))) YIELD g1

        UNWIND g1 AS e

        SET e.weight=projection(1/(abs(r.amount-st.amount)+0.0001),ln(1+1/abs(r.endtime-st.endtime)),count(out(v))/count(in(v)))

        MATCH u=src(e) SET u.rel=reduce(sum = 0, o IN out(u) | sum+o.weight\*dst(o).rel)

        RETURN g1

intersect

WITH entry = (MATCH n in nodes(r) WHERE count(in(n))=0 ORDER BY n.rel DESC LIMIT 15)

        BFS (re IN forward(entry) | MATCH u=src(re) Where re.endtime>min(collect(uin IN in(u) | uin.starttime)) and re.starttime<1724731846712161377) yield g2

        RETURN g2)

* The union graph has **124 vertices and 282 edges**.

A drawing of a pipe

AI-generated content may be incorrect.

<https://github.com/ProGQL/ProGQL/blob/main/technical%20report/Password%20Crack/neo4jpwd-vmunion.svg>

**Controlled depth comparison with Cypher**

To further highlight ProGQL’s precision, we conducted a controlled comparison by running backward BFS from the POI on host1 with a maximum depth of **nine**. We then compared the provenance graphs produced by ProGQL and by a best-effort Cypher approximation of the same query.

Below we show the Cypher query we constructed as a best-effort approximation of the ProGQL backward BFS; this highlights the structural similarities but also makes evident where Cypher fails to enforce recursive, edge-dependent constraints.

MATCH (p:Process)-[st:FileEvent{optype:"write"}]->(start:File{name:"/tmp/passwords.tar.bz2", hostid:"1"})

// Carry the POI and initialize a max end-time constant.This matches the same starting condition as in ProGQL.

WITH start, 1724731846719889370 AS initialMaxEndTime

// APOC expands all backward paths up to maxLevel. Here, YIELD path always means the **entire path from start to the current node**, not just the most recent hop.

This whole-path semantics is where Cypher diverges from ProGQL. ProGQL expands *step by step* with constraints applied at each hop; APOC dumps a superset of all traversed edges into **path**.

CALL apoc.path.expandConfig(start, {

    relationshipFilter: "<",

    minLevel: 0,

    maxLevel: 9,

    bfs: true,

    uniqueness: "NODE\_GLOBAL",

    filterStartNode: false

}) YIELD path

// Label the path (visitedPath) and extract its current endpoint (currentNode).

WITH start, path AS visitedPath, last(nodes(path)) AS currentNode, initialMaxEndTime

// Intended to prune traversal depth.

WHERE length(visitedPath) < 9

// Tries to restrict expansion to edges that are in the current path. But because **visitedPath** is the *whole path*, this condition still admits edges from earlier steps - including ones that would have been pruned if filtering happened inline (like in ProGQL). This is the main FP source.

OPTIONAL MATCH (currentNode)-[outgoing]->()

WHERE outgoing IN relationships(visitedPath) OR currentNode = start

// Aggregate outgoing edges to compute the temporal bound. But since outgoing edges include superset ones, the max can be inflated.

WITH start, currentNode, outgoing, max(outgoing.endtime) AS maxEndtime, initialMaxEndTime, visitedPath

// Set the temporal constraint. At the POI, use the initial cutoff; otherwise, inherit the max endtime.

WITH start, currentNode, outgoing, initialMaxEndTime, visitedPath,

     CASE

WHEN currentNode = start THEN initialMaxEndTime

ELSE COALESCE(maxEndtime, initialMaxEndTime)

     END AS updatedMaxEndTime

// Apply causal pruning (only accept edges that end before the cutoff). But pruning happens **after expansion**, so invalid edges were already collected into visitedPath.

MATCH (currentNode)<-[incoming]-()

WHERE incoming.starttime < updatedMaxEndTime

// Gather pruned incoming edges and project source/sink sets. This simulates causal filtering, but again false positives that slipped into visitedPath remain in play.

WITH start, currentNode, COLLECT(incoming) AS updatedVisitedRels, updatedMaxEndTime, visitedPath

WITH start, currentNode, updatedVisitedRels, updatedMaxEndTime,

     [r IN updatedVisitedRels | startNode(r)] AS sourceNodes,

     [r IN updatedVisitedRels | endNode(r)] AS sinkNodes, visitedPath

// Output the traversal frontier.

RETURN DISTINCT currentNode, updatedVisitedRels AS visitedRels, updatedMaxEndTime, sourceNodes, sinkNodes;

At depths (1–8), the difference is not immediately visible. At depth 9, the provenance graph returned by ProGQL has 119 vertices and 2998 edges (<https://github.com/ProGQL/ProGQL/blob/main/technical%20report/Password%20Crack/neo4jpwd-back1-depth-9.svg> ), while the PG returned by Cypher has 138 vertices and 6600 edges(<https://github.com/ProGQL/ProGQL/blob/main/technical%20report/Password%20Crack/CypherPwdByDepth-9.svg> ).

Cypher (even with APOC) is fundamentally **path-oriented**: it enumerates complete paths from the start node. Filters like WHERE incoming.starttime < updatedMaxEndTime are evaluated **after a path has been generated**, based only on the nodes and relationships in that path. This means Cypher cannot enforce **recursive, edge-dependent constraints during the traversal itself**. Instead, it allows traversal to continue along edges that should already have been pruned, and only later removes them from the returned results. As a consequence, Cypher produces additional edges (false positives) because it lacks ProGQL’s ability to prune dynamically at each recursion step. In contrast, **ProGQL is subgraph-oriented**: its BFS operator applies recursive causal constraints (e.g., r.starttime < max(out(v).endtime)) at every step, ensuring that only causally consistent edges are included.

At shallow depths (1–8), the difference is not immediately visible, because most paths near the POI happen to overlap with causally valid edges, so Cypher’s outputs look superficially similar. However, starting at depth 9, the divergence becomes pronounced. Cypher continues expanding along many irrelevant paths - such as PAM libraries, NSS lookups, and background socket activity - because it evaluates constraints only after paths are generated, rather than pruning edges recursively during traversal. ProGQL, by contrast, consistently excludes these false positives, maintaining a concise subgraph that reflects only attack-relevant dependencies.

**3. Expressiveness and Case Studies**

To further address reviewer concerns about the **usability and practical benefits of ProGQL**, we performed additional experiments across three multi-host attack scenarios: Password Crack, Data Leakage, and Vpn Filter. The password crack case has already been described in the previous section; below, we introduce the other two attacks.

* **Attack Case – Data Leakage**: After Shellshock Penetration, the attacker attempts to steal all the valuable assets from the host. This stage mainly involves the behaviors of local and remote file system scanning activities, copying and compressing of important files. The attacker initiated the second stage of the attack by downloading the script **leak\_data.sh** from their server. The script was then transferred to a compromised host (host2) using scp and executed. The **leak\_data.sh** script bundled specific files, including hidden files and sensitive system files, into a tarball. This tarball was then compressed as **leaked.tar.bz2** and exfiltrated to the attacker’s designated server for further use.
* **Attack Case – VPN Filter**: At this stage, the attacker targeted to establish a persistent connection between the victim hosts and the C2 server. The attacker utilized VPNFilter malware, which infected millions of IoT devices by exploiting a number of known or zero-day vulnerabilities. The attacker first downloaded **vpn\_filter.sh** script to host1, the script then was transferred to host2 and executed. **vpn\_filter.sh** changed to the /tmp directory on host2, downloaded **vpnfilter** from the attacker’s server, made it executable, and then ran it with C2 server IP address. This established a persistent connection between host2 and the C2 server.

**3.1 Feature Ablation Experiments**

**ProGQL** can naturally express both **entry node candidate selection** and **feature-based edge weighting**. For example, top-K entry node ranking can be directly expressed using LIMIT, making it straightforward to select the most likely attack entry points. Similarly, feature selection can be declaratively controlled via projection functions in the query. For instance:

SET e.weight = projection(1/(abs(r.amount - st.amount) + 0.0001))

This example uses only the **data flow feature** (amount) for weight computation, thereby simulating the **1-feature ablation**. By extending the projection to include temporal and structural features, we obtain the **3-feature setting**.

A graph with lines and numbers

AI-generated content may be incorrect.A graph with lines and text

AI-generated content may be incorrect.A graph of a graph with numbers and symbols

AI-generated content may be incorrect.A graph of a function

AI-generated content may be incorrect.A graph showing a number of data

AI-generated content may be incorrect.A graph with a line and a chart with text

AI-generated content may be incorrect.

**Figure 1: Feature ablation analysis across three attack cases (Password Crack, Data Leakage, and Vpn Filter)**

* **Orange line** = 1 Feature, with **red X** marking failures to identify ground-truth entry nodes.
* **Blue line** = 3 Features, with **purple triangles** marking failures to identify ground-truth entry nodes.
* **Dashed line at TopK** = max entry nodes for Host1 or Host2.

**Trade-off Between Recall and False Positives**

The comparison between **recall** and **precision** across the three case studies shows the natural trade-off:

* Increasing **Top-K candidate entry nodes** consistently improves recall, as more candidates are included.
* However, this comes at the cost of higher false positive rates (lower precision), because additional non-critical edges are inevitably introduced.
* This behavior matches the intuition in cyber attack investigations: critical edges form only a small fraction of the overall graph, so enlarging the candidate pool helps capture them but risks decreasing precision.

**1 Feature vs 3 Features**

Across all three benchmarks, the **3-feature weighting** scheme consistently outperforms the 1-feature variant:

* **Recall:** Both 1- and 3-feature settings eventually achieve high recall at larger Top-K, but 3 features often reach comparable recall earlier (e.g., Password Crack case), indicating more efficient prioritization.
* **Precision:** The benefit of 3 features is most evident in precision. For both Data Leakage and Vpn Filter, precision with 3 features is higher, reflecting reduced false positives. This shows that combining temporal, structural, and amount features helps suppress spurious edges that a single feature cannot disambiguate.
* **Robustness:** 1-feature ranking misses entry nodes (red X), while 3-feature ranking avoids or delays such failures (purple triangles). This highlights that **multi-feature weighting reduces the risk of overlooking critical entry nodes**.

The ablation analysis demonstrates that **ProGQL not only allows flexible control over entry node selection and feature weighting but also validates the necessity of combining multiple features**. Using three features produces higher precision and more robust recall than using a single feature. This reinforces the claim that ProGQL provides a principled and effective framework for **investigating cyber attacks on provenance graphs**.

**3.2 Conciseness Evaluation**

We compared query conciseness across languages using three metrics: **# of constraints, # of words, and # of characters**. Figure 2 plots these metrics for three multi-host attack cases.  Across three cases,ProGQL consistently outperforms Cypher in terms of brevity: on average, ProGQL requires **9× fewer constraints, 15× fewer words, and 17× fewer characters**.

We define a **constraint** as an atomic restriction that filters, bounds, or enforces semantics in a query. This includes:

* **WHERE clauses** (temporal, structural, equality/inequality conditions)
* **typed edge/node restrictions** (e.g., FileEvent{optype:”write”})
* **aggregations used for pruning** (e.g., max(...), min(...))
* **bounds** such as LIMIT clauses or depth cutoffs.

A **word** is defined as a token separated by whitespace.

A **character** is defined as any symbol in the query string **excluding whitespace**, such as letters, digits, punctuation, and operators.

A graph of words and a number of words

AI-generated content may be incorrect.A graph of blue and orange bars

AI-generated content may be incorrect.A graph of different colored bars

AI-generated content may be incorrect.

**Figure 2: Conciseness evaluation of queries written in ProGQL and Cypher**

Figure 2 shows that ProGQL substantially reduces syntactic overhead compared to Cypher. This reduction highlights the expressiveness and practicality of ProGQL in real-world attack investigation tasks.

**3.3 Evaluation Across 14 Attacks**

We benchmarked ProGQL on DARPA TC datasets and constructed multi-host scenarios (Password Crack, Data Leakage, VPN Filter).

* **Average input size:** ~19M events
* **Average output PG:** ~176 edges (≈0.1% of raw events)
* **Memory:** ProGQL 5GB vs. DEPIMPACT >100GB
* **Runtime:** ProGQL queries 20× faster than Cypher (20s vs. 414s)
* **Backend portability:** Relational (PostgreSQL, MyRocks, MariaDB) and Graph (Neo4j, Nebula) supported via factory pattern

**4. System Design**

ProGQL’s system is a well-engineered system. It is designed to be **scalable and extensible**, with three main modules:

1. **Data Importer**

* Parses system audit logs (Sysdig, DARPA TC).
* Performs batch insertion into multiple backends.

1. **Language Parser**

* Implements grammar rules (ProGQL.g4).
* Extracts query context for execution.

1. **Query Execution Engine (Section IV.C)**

* **Graph traversal**: incremental BFS/DFS queries to database.
* **Edge merge**: collapse parallel edges by temporal threshold.
* **Weight computation**: feature projections + normalization.
* **Score propagation**: iterative weighted reduce until convergence.
* **Output processing**: efficient union/intersection with edge signatures.

**Design pattern:** The engine employs a **Factory Pattern** to abstract backend-specific query generation. This allows a single ProGQL query to be executed on PostgreSQL, MyRocks, MariaDB, Neo4j, or Nebula transparently. This architectural choice strengthens.

**5. Conclusion**

ProGQL combines a provenance-aware DSL with a scalable, extensible system design. By introducing novel language constructs and a factory-driven query engine, ProGQL enables analysts to declaratively encode domain knowledge, scale to millions of events, and reconstruct multi-stage, multi-host attack chains with high precision.

This technical report complements our ICDE 2026 submission by detailing the language’s formal definition, case studies, system design, and algorithmic foundations.

**Appendix**

**Algorithm 1: Backward BFS in PA**

[GitHub link](https://github.com/ProGQL/ProGQL/tree/main/appendix/backward%20bfs%20in%20PA#backward-bfs-in-pa)

**Algorithm 2: Language Parser**

[GitHub link](https://github.com/ProGQL/ProGQL/tree/main/appendix/language%20parser)