OpenSPG

Use Case

In a network environment, each router participates in the exchange of routing information through protocols like the Border Gateway Protocol (BGP). Routers receive advertised routes from peers, apply import policies (e.g., based on trust level, prefix filters, or route origin), and selectively propagate accepted routes to other peers. This process enables the dynamic construction of routing tables, which are essential for directing internet traffic efficiently across autonomous systems (ASes).

However, the highly distributed and policy-driven nature of BGP introduces vulnerabilities during topology changes or failures. For example, when a BGP router loses connectivity to an upstream peer, it triggers a **route withdrawal**, ceasing advertisement of network prefixes that were previously reachable via that peer. This event initiates a **routing convergence process**, where adjacent routers must recompute alternative paths. Depending on the network topology and routing policies, this may involve several rounds of updates and recalculations across multiple ASes.

During this convergence period, transient issues such as **route flapping**, **inconsistent path selection**, or **temporary blackholes** may occur. Packets destined for withdrawn prefixes may be **dropped** or **rerouted inefficiently**, leading to **service degradation**, **increased latency**, or **temporary outages**. The impact can be especially severe in latency-sensitive applications like VoIP, video streaming, or financial transactions.

Given the time-sensitive and cascading nature of these disruptions, it is imperative to **rapidly localize the root cause** — the network element (e.g., a misconfigured router, a failing link, or a flapping peer connection) that initially triggered the instability. Traditional monitoring systems may detect the symptoms (e.g., packet loss or reachability issues) but not the origin. Therefore, **automated analysis tools** capable of correlating BGP updates, inferring propagation delays, and understanding AS-level topologies are crucial.

Such tools can:

- Detect anomalous routing behaviors in real time.
- Trace the **origin of route withdrawals or flaps**.
- Predict the scope and duration of impact.
- Assist operators in proactive mitigation, such as policy reconfiguration or route dampening.

In this context, data-driven approaches — such as **graph-based models**, **temporal correlation of BGP events**, and **semantic enrichment of routing metadata** — can significantly enhance visibility and resilience in inter-domain routing operations.

Create a New Project

1. Create a new project configuration inside the folder (OpenSPG/KAG/kag/examples)

Main elements are as follows and they will be discussed in the following sections:

- Project configuration
- Project builder
- Project solver

```yaml
#------------project configuration start----------------#
openie_llm:
  api_key: your-key
  base_url: https://api.deepseek.com
  model: deepseek-chat
  type: maas

chat_llm: &chat_llm
  api_key: your-key
  base_url: https://api.deepseek.com
  model: deepseek-chat
  type: maas

vectorize_model: &vectorize_model
  api_key: your-key
  base_url: https://api.siliconflow.cn/v1/
```

When creating this configuration important things to pay attention:

1.1. API keys for vectorization and LLM models

```yaml
openie_llm:
  api_key: your-key
  base_url: https://api.deepseek.com
  model: deepseek-chat
  type: maas

chat_llm: &chat_llm
  api_key: your-key
  base_url: https://api.deepseek.com
  model: deepseek-chat
  type: maas

vectorize_model: &vectorize_model
  api_key: your-key
  base_url: https://api.siliconflow.cn/v1/
  model: BAAI/bge-m3
```

1.2. Project name

```yaml
namespace: ExampleNetwork
```

1.3. Project Builder will describe the data pipeline and the data format (csv in this case)

```yaml
#------------project builder start----------------#
kag_builder_pipeline:
  chain:
    type: structured_builder_chain # kag.builder.default_chain.DefaultStructuredBuilderChain
    mapping:
      type: spg_mapping # kag.builder.componnet.mapping.SPGTypeMapping
    writer:
      type: kg_writer # kag.builder.component.writer.kg_writer.KGWriter
  num_threads_per_chain: 1
  num_chains: 16
  scanner:
    type: csv_scanner #json_scanner # kag.builder.component.scanner.csv_scanner
#------------project builder end----------------#
```

1.4 Project Solver will describe the pipeline for reasoning and querying

```yaml
#------------kag-solver configuration start----------------#
search_api: &search_api
  type: openspg_search_api #kag.solver.tools.search_api.impl.openspg_search_api.OpenSPGSearchAPI

graph_api: &graph_api
  type: openspg_graph_api #kag.solver.tools.graph_api.impl.openspg_graph_api.OpenSPGGraphApi

chain_vectorizer:
  type: batch
  vectorize_model: *vectorize_model

exact_kg_retriever: &exact_kg_retriever
  type: default_exact_kg_retriever # kag.solver.retriever.impl.default_exact_kg_retriever.DefaultExactKgRetriever
  el_num: 1
  llm_client: *chat_llm
  search_api: *search_api
```

2. Create a new project

```shell
knext project create --config_path ./kag_config_network.yaml
```

After this script you will have your folder with your chosen project name and modules inside as builder, reasoner, schema, solver as well as configuration file.

Create a Schema

```shell
cd network/schema
```

```scheme
namespace NetworkInitial

TaxonomyNetworkElement(TaxonomyNetworkElement): ConceptType
    hypernymPredicate: isA

NetworkElement(NetworkElement): EntityType
    properties:
        neID(neID): Text
        neName(neName): Text
        hasNetworkElement(hasNetworkElement):NetworkElement
        IND#belongTo(belongTo): TaxonomyNetworkElement

VirtualRoutingFunction(VirtualRoutingFunction): EntityType
    properties:
        vrfName(vrfName): Text
        IND#belongTo(belongTo): TaxonomyNetworkElement
```

When creating a schema important the basic elements/things to pay attention:

- Defining nodes: Concept type, Entity Type and Event Type
- Defining properties : Properties can be of basic, standard types and relationships
  - Basic Type：Text、Integer、Float
  - Standard Type：STD.ChinaMobile、STD.Email、STD.IdCardNo、STD.MacAddress、STD.Date、STD.ChinaTelCode、STD.Timestamp

        - **Properties** can exist on both nodes and relationships, where:
            - **Node properties** describe the characteristics of an entity (e.g., `age` for `Person`).
            - **Relationship properties** describe specific attributes of the connection between two entities (e.g., the amount of a transaction, the date of a transaction, etc.).
        - The property id, name, and description are built-in and do not need to be explicitly declared
        - The English name of the property must start with a lowercase letter and can only contain letters and numbers (no hyphens etc)
        - Constraints can be defined for properties and rules:
            - Properties: notNull, MultiValue
        - Rules
    - Event types can point to any type, entity types cannot point to event types, and concept types can only point to other concept types, while the reverse is prohibited.
    - Concept types can only have the parent class "Thing" and cannot inherit other types. This is because concept types inherently have a hierarchical relation, implying inheritance semantics. If concept types were to inherit, it would result in conflicting semantics.
    - Shema should be as close as possible to the data otherwise there should be some mapping between non-matching attribute name/schema element (SPGTypeMapping parses the attribute name from the CSV file and map it to the properties defined in the EntityType.
    - If the relation is defined as relation and not as property then there should be a specific table for it. Otherwise it should be defined in the property. Risk mining example: The example doesn't work because there is not a table in the dataset pointing at the relation. Either the it should be defined in the property and riectly add it to the table or it should be defined in the relation and create a new table.
    - Sometimes leaving leadTo in the properties can cause an error of duplicate key entry. (In the example it is on the relations.

scss Full Screen

```scss
RouteWithdrawEvent(RouteWithdrawEvent): EventType
    properties:her): Text
        peerAddress(peerAddress): Text
        isAdjRIBin(isAdjRIBin): Text
        isAdjRIBout(isAdjRIBout): Text
        IND#belongTo(belongTo): TaxonomyControlPlane
    relations:
        CAU#lead
        subject(subject): NetworkElement
        index(index): Index
        trend(trend): Trend
        time(time): STD.Date
        neID(neID): Text
        routeDistinguisher(routeDistinguisTo(leadTo): DroppedTrafficEvent
```

Details can be found in the following links:

openspg.yuque.com

openspg.yuque.com

openspg.yuque.com (Example schema customisation)

The relationships can be described in 2 ways: Phsically in the schema and using DSL rules. "The relationships expressed using DSL rules in the SPG schema are generated through real-time computation during N-degree inference, which effectively meets this requirement."

Knowledge Graph Construction

KGBuilder Pipeline:
openspg.yuque.com

- Structured Mapping: The original data and the schema-defined fields are not completely consistent, so a data field mapping process needs to be defined.
- Entity Linking: In relationship building, entity linking is a very important construction method. This example demonstrates a simple case of implementing entity linking capability for companies.
- RiskMining application it takes around 15 minutes to build the data (40KB) with vectorizer

Inference

graph inference-based question answering can be done in 2 ways(openspg.yuque.com):

- **Inference with Existing Data Modeling:** This type of inference is for structured data that has a clear data schema. Challenges are:
  - Data Scale Limitation: Large models cannot directly handle massive amounts of structured data.
  - Insufficient Knowledge Dependency: Large models lack sufficient knowledge about the underlying data.
- **Inference without Data Modeling:** This type of inference is for unstructured data that lacks a clear data schema.
  - In such scenarios, the system cannot rely on a predefined schema to optimize the planner (Planner) and instead uses a weak schema constraint mechanism to express any type of data through entity types (Entity).

The Schema Rules

This is the data for RouteWithdrawEvent:

```python
id,name,subject,index,trend
1,顺丁橡胶成本上涨,商品化工-橡胶-合成橡胶-顺丁橡胶,route,withdraw
2,RouteWithdrawEvent1,P1,route,withdraw
3,RouteWithdrawEvent2,P2,route,withdraw
4,RouteDropEvent1,P3,route,drop
```

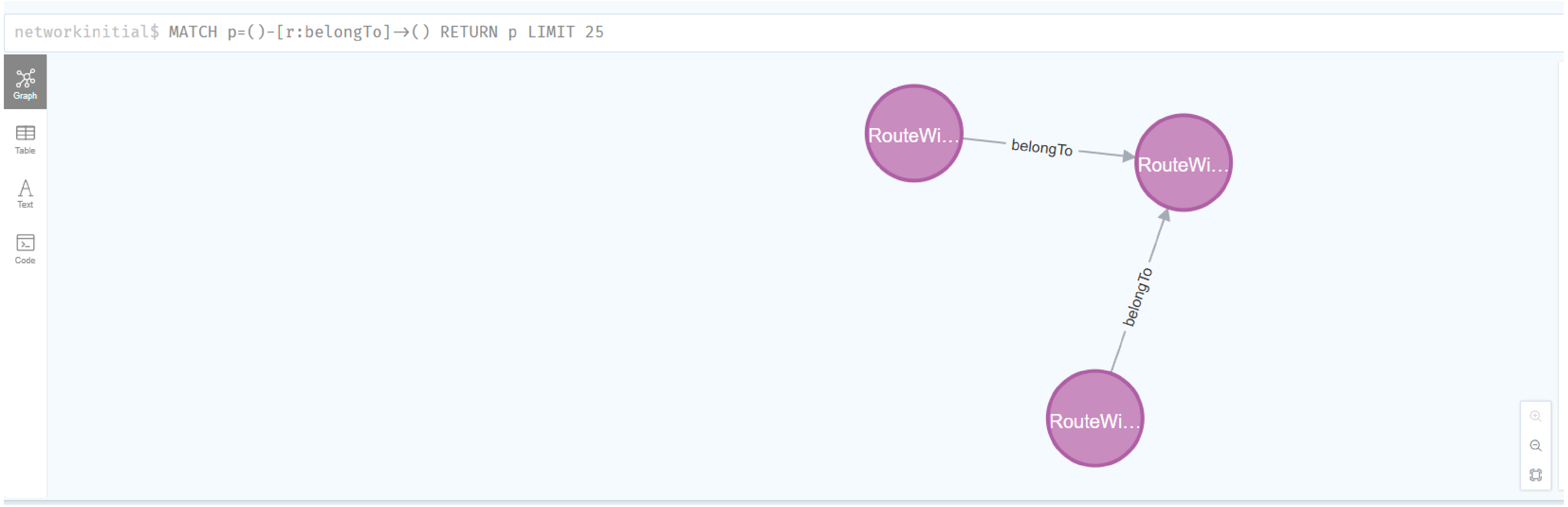
BelongTo Relationship

The concept rules are added for a **belongTo** relationship for the condition(constrant) **index=route** and **trend=withdraw** as follows:

```javascript
`TaxonomyControlPlane`/`RouteWithdraw`:
    rule: [[
        Define (e:RouteWithdrawEvent)-[p:belongTo]->(o:`TaxonomyControlPlane`/`RouteWithdraw`) {
            Structure {
            }
            Constraint {
                R1: e.index == 'route'
                R2: e.trend == 'withdraw'
            }
        }
    ]]


```

Then the data can be queried from Neo4J:

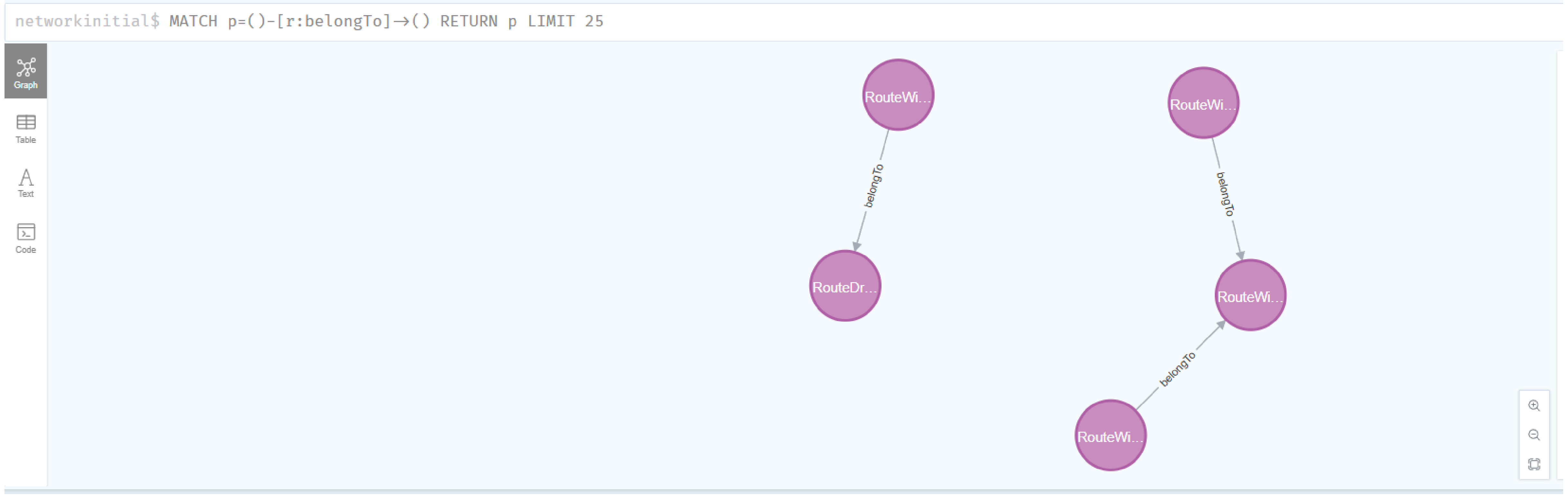


```
networkinitial$ MATCH p=()-[r:belongTo]→() RETURN p LIMIT 25
```

When I added the concept rules for a belongTo relationship for RouteDrop I see the following data from Neo4J

```javascript
`TaxonomyControlPlane`/`RouteWithdraw`:
    rule: [[
        Define (e:RouteWithdrawEvent)-[p:belongTo]->(o:`TaxonomyControlPlane`/`RouteWithdraw`) {
            Structure {
            }
            Constraint {
                R1: e.index == 'route'
                R2: e.trend == 'withdraw'
            }
        }
    ]]

`TaxonomyControlPlane`/`RouteDrop`:
    rule: [[
        Define (e:RouteWithdrawEvent)-[p:belongTo]->(o:`TaxonomyControlPlane`/`RouteDrop`) {
            Structure {
```



```
networkinitial$ MATCH p=()-[r:belongTo]→() RETURN p LIMIT 25
```

LeadTo Relationship

This is supposed to be a logical rule which should be created on the fly. The idea is that one event should lead to another event and for that case based on the given constraints a new node and a property should be created.

```groovy
```

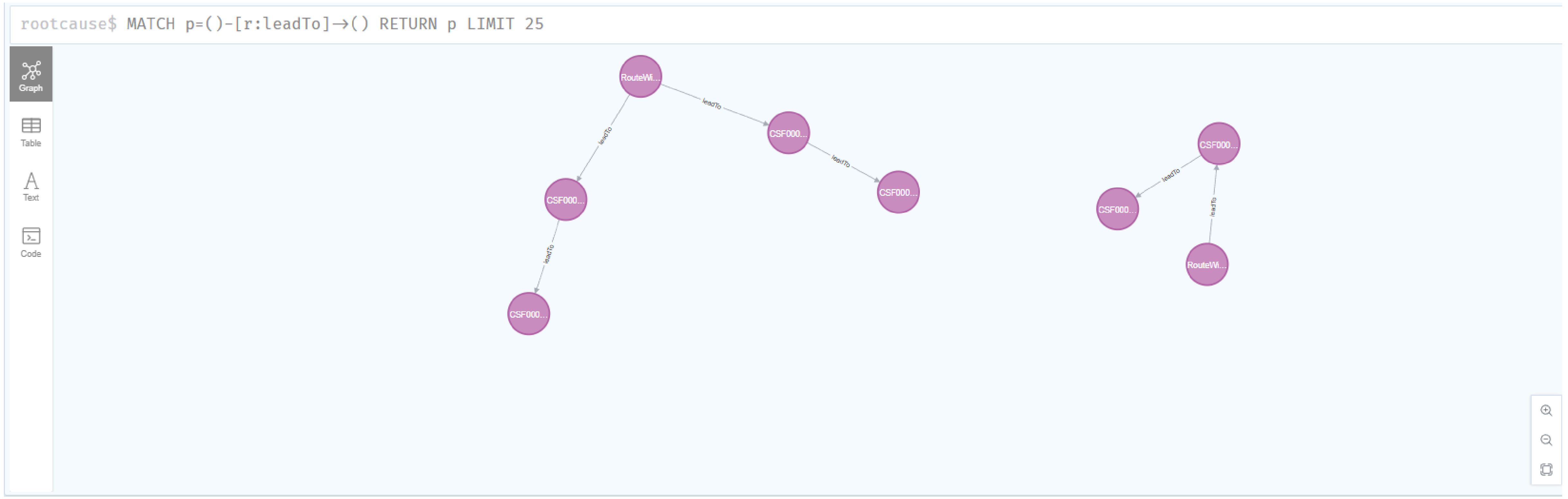
```
`TaxonomyControlPlane`/`routewithdraw`:TaxonomyForwardingPlane/`trafficwithdraw`
    rule: [[
        Define (s:`TaxonomyControlPlane`/`routewithdraw`)-[p:leadTo]->(o:`TaxonomyForwardingPlane`/`trafficwithdraw`) {
            Structure {
                (s)-[:subject]->(prod:NetworkElement)-[:hasNetworkElement]->(down:NetworkElement)<-[:relation]-(c:VirtualRoutingFunction)
            }
            Constraint {
            eventName = concat(c.name, "trafficwithdrawevent")
            }
            Action {
                downEvent = createNodeInstance(
                    type=DroppedTrafficEvent,
                    value = {
                        subject=c.id
                        name=eventName
                        trend="withdraw"
```

Given the structure of the data (Structure) and constraint, at the end, the instances are assigned to the specific classes. In this case, rule defines **a causal relationship** between two types of events—**`routewithdraw`** and **`trafficwithdraw`**—within different taxonomies (`ControlPlane` and `ForwardingPlane`). It does this by specifying a structured, rule-based inference that dynamically creates a new event (`DroppedTrafficEvent`) and links it to the existing one using the `leadTo` relationship.

It encodes causal inference logic: *"If a route withdrawal happens in the control plane and it's linked to a specific network structure, then infer that a traffic withdrawal event should happen in the forwarding plane."*

This is part of OpenSPG's event-centric knowledge modeling, where such rules help automatically infer downstream effects of events in enterprise systems like telecom networks.

The second rule in the OpenSPG definition establishes a **causal inference** from a `trafficwithdraw` event to a `networkdrop` event, both within the `TaxonomyForwardingPlane`. Specifically, it states that if a `trafficwithdraw` event occurs and is linked via the `subject` relationship to a `VirtualRoutingFunction`, then a new event—`networkdropevent`—should be generated automatically. This new event is modeled as a `DroppedTrafficEvent` with properties indicating a "drop" trend and "network" index. The rule constructs this new node and creates a `leadTo` edge from the original `trafficwithdraw` event to the newly inferred `networkdrop` event. This supports automated propagation of cascading network anomalies in the knowledge graph by making implicit cause-effect relationships between events explicit through structured rules.



```
rootcause$ MATCH p=()-[r:leadTo]→() RETURN p LIMIT 25
```

MySQL and Neo4j Database Imports

The schema is stored in the MySQL database and the instance data is stored in the Neo4j database. When I make any changes to the data or schema or the data, before reuploading them I delete everything so that there would be no problem between different rule/schema versions.

MySQL

Script for entering the docker mysql environment

```sql
docker exec -it release-openspg-mysql mysql -uroot -p
```

Default password:openspg

After entering the password you can see the tables created for openspg database:

```sql
show databases;
USE openspg;
show tables;
| Tables_in_openspg                   |
+-------------------------------------+
| kg_biz_domain                      |
| kg_builder_job                     |
| kg_config                          |
| kg_data_source                     |
| kg_ontology_entity                 |
| kg_ontology_entity_parent          |
| kg_ontology_entity_property_range |
| kg_ontology_ext                    |
| kg_ontology_property_constraint    |
| kg_ontology_release                |
| kg_ontology_semantic               |
```

I am deleting the rules created to be sure that no two rules that I create would cause any problem.

```sql
select * from kg_ontology_semantic; --> This table show the rules created for the schema elements (just showing which elements)
delete from kg_ontology_semantic;
select * from kg_semantic_rule; --> This table show the rules executed for the schema elements (full rule as in the concept.rule)
delete from kg_semantic_rule;
select * from kg_ontology_property_constraint;
delete from kg_ontology_property_constraint;
select * from kg_ontology_entity_parent;
delete from kg_ontology_entity_parent;
select * from kg_ontology_entity;
delete from kg_ontology_entity;
```

Neo4j

Other thing to delete is the Neo4j database that you're worki6ng on

```sql
DROP DATABASE network;
```

Finally it is necessary to delete the folder where the metadata is stored.

```plaintext
/home/iomuser1/github/openspg/KAG/kag/examples/network/builder/extract-runner-ckpt
```

OWL-OpenSPG Schema Comparison

High Level Comparison

OpenSPG

Advantages

- Already integrated with AI models, and cloud platforms.
- Built-in support for integrating AI models with graph reasoning
- Can connect with ML models for predictive analytics
- Enables neural-symbolic learning → using deep learning to enhance knowledge graph reasoning
- Supports faster graph traversals than triplestores (hypothetically)

Disadvantages

- More complex due to its hybrid model (Property Graph + RDF)
- Newercompared to RDF triplestores and Property Graph databases
- Fewercommunity resources, documentation, and best practices.
- Limitedavailability of third-party integrations, connectors, and tools
- Updates and feature improvements may not be as fast or stable as in triplestores
- As an open-source project (OpenSPG), long-term support depends on community engagement and developer contributions.
- Semantic reasoning (SPG-Reasoner) can be computationally expensive, especially forlarge-scale graphs.
- Combining Property Graph traversal with reasoning-based inference could lead to latency issues in real-time applications.

RDF

Advantages

- Strong reasoning (including multi-hop reasoning), enabling automatic inference of newrelationships.
- Uses standards, making it interoperable with other semantic web technologies.
- Easy integration with external datasets
- Allows for schema evolution without breaking existing data.

•Supports FAIR (Findable, Accessible, Interoperable, Reusable) principles for long term knowledge management and maintenance

Disadvantages

•Querying large datasets can be slower than native PG

•Joins-heavy queries (due to triple structure) can become computationally expensive

•Large-scale RDF datasets require efficient indexing strategies to avoid performancebottlenecks.

•Slower for graph traversal

•RDF-based ontologies require strict schema management, which can be challenging when data models change

Feature	SPG-Schema (Semantic-enhanced Property Graphs)	Ontology-based Schema (RDFS/OWL - RDF Triplestore)	Schema-less (Property Graphs - Neo4j, JanusGraph)
Schema Definition	**Hybrid model**: Combines elements of **RDF schemas and Property Graphs** with controlled semantics.	**Strict ontology-driven**: Uses RDF Schema (RDFS) and OWL for defining classes, properties, and constraints.	**Flexible, No enforced structure**: Nodes and edges can have any properties dynamically.
Data Model	Property Graph with **semantic enhancements** (supports labels, relationships, and typed attributes).	**Triple-based** (subject-predicate-object) with strict ontological constraints.	**Node-Edge model** (like SPG but without schema constraints).
Reasoning & Inference	Supports **programmable reasoning (SPG-Reasoner)** using **KGDSL**, enabling custom rules and hybrid AI-symbolic reasoning.	Strong reasoning using **RDFS entailment, OWL DL**, and SPARQL inferencing. Supports **logical inference** but can be computationally expensive.	**No built-in reasoning** (unless using external tools like Graph Data Science in Neo4j).
Query Language	**KGDSL (Knowledge Graph DSL) + Property Graph queries (Cypher, Gremlin)**	**SPARQL (standard for RDF queries)**	**Cypher (Neo4j), Gremlin (TinkerPop)**
Performance & Scalability	Optimized for **both transactional and analytical workloads**. Can leverage **big data architectures**.	RDF triplestores may struggle with **large-scale queries** due to the triple indexing overhead.	**High-performance queries**, but lacks semantic search capabilities.
Best Use Cases	- Hybrid graphs where **semantic reasoning and graph traversal** are both important. - **Enterprise knowledge graphs** with AI-driven reasoning. - **Scalable, big data-driven graphs**.	- **Semantic Web, Ontologies, Open Data** (e.g., Wikidata, DBpedia). - **Regulatory & Compliance data** where reasoning is crucial. - **Healthcare and scientific domains**.	- **Operational Graph Databases** (recommendation engines, fraud detection, social networks). - **Fast, flexible applications** without strict schema constraints.

Challenges	- **More complex** than traditional Property Graphs. - **Learning curve** due to hybrid schema.	- **Performance bottlenecks** at scale due to reasoning overhead. - **Difficult to manage for dynamic/fast-changing data**.	- **Lack of constraints** can lead to inconsistent data. - **No built-in reasoning** for inferencing.
Flexibility vs. Control	**Balanced:** Provides schema control but allows extensions.	**Highly controlled:** Strict ontology-based constraints.	**Highly flexible:** No constraints but no schema validation.
Expressiveness	Supports **semantic constraints** but retains flexibility for graph traversal.	High expressiveness due to **OWL-based reasoning**, supporting complex **class hierarchies, rules, and inference**.	Less expressive; relationships exist but **lack semantic reasoning** capabilities.

OpenSPG Schema - OWL comparison

HYP: Hypernym relation

OpenSPG Relation Type	OWL Equivalent	Explanation
isA Is a type of...	**rdfs:subClassOf**	Describes subclass relationships (e.g., "Dog is a type of Mammal").
locateAt Is located at...	**Object Property** (`hasLocation`)	Defines relationships between entities (e.g., "Office is located at New York").
mannerOf A is a specific implementation or way of B. Similar to "isA," but used for verbs. For example, "auction" → "sale"	**Object Property** (possibly with `rdfs:subClassOf` or `rdf:Property`)	Describes specific methods or implementations (e.g., "Auction is a specific form of Sale").

SYNANT: Synonymy/Antonymy relation

Synonymy and **Antonymy** typically aren't directly supported in OWL but can be expressed through **labels** or **comments** that describe relationships between terms. RDF and OWL focus more on the **semantic meaning** of concepts, rather than direct synonym/antonym distinctions.

OpenSPG Relationship Type (Descriptions from OpenSPG)	Possible OWL Equivalent / Concept	Explanation

synonym Expresses synonyms.	**rdfs:label / equivalentClass**	Use `rdfs:label` or `rdfs:comment` to associate labels or synonyms. Alternatively, `equivalentClass` can be used to declare two classes as equivalent.
antonym Expresses antonyms.	**rdfs:comment / negative relations via axioms**	**Antonyms** can be indicated with annotations or by defining **inverse relationships** or negative axioms.
symbolOf A symbolically represents B. For example, "red" → "passion".	**ObjectProperty** or **AnnotationProperty**	A symbol (like "red" for "passion") might be represented using an **ObjectProperty** or **AnnotationProperty** in OWL to link concepts symbolically.
distinctFrom A and B are different members of a set, and something that belongs to A cannot belong to B. For example, "August" → "September".	**DisjointClasses**	In OWL, two classes can be defined as **disjoint** using `rdfs:disjointWith` or `owl:disjointWith`, meaning they cannot have any instance in common.
definedAs A and B have significant overlap in meaning, but B is a more explanatory version of A. For example, "peace" → "absence of war".	**EquivalentClass / rdfs:comment**	A more explanatory or detailed definition can be represented using `rdfs:comment` or by specifying an `EquivalentClass` axiom.
locatedNear A and B are usually found near each other. For example, "chair" → "table".	**ObjectProperty** or **proximity-based reasoning**	OWL does not have a direct way to specify proximity, but **object properties** can be used to link concepts, and reasoning can be used to infer "closeness" based on relationships.

similarTo A and B are similar. For example, "blender" → "food processor".	**EquivalentClass / ObjectProperty**	Similar concepts can be modeled using `EquivalentClass`, or using `ObjectProperties` if the similarity reflects a **relationship** between entities.
etymologicallyRelatedTo A and B have a common origin. For example, "folkmusiikki" → "folk music".	**AnnotationProperty**	**Etymology** can be represented as an **annotation property** or an **ObjectProperty** linking related terms based on linguistic origin.

CAU: Causal relation

Causal relationships are usually expressed through **ObjectProperties**. OWL can define these using properties like `causes` and `leadsTo`. Reasoning can infer causal relationships.

OpenSPG Relationship Type (Descriptions from OpenSPG)	**Possible OWL Equivalent / Concept**	**Explanation**
leadTo Expresses the logical rule through which an event is generated, such as an instance of event A generating an instance of event B under specified conditions. This predicate is recognized by the system as an intention for instance generation, used for implementing the instance propagation of events.	**ObjectProperty** (causal)	A **cause-effect** relationship (like "hunger leads to the need to eat") is expressed using **ObjectProperties** and can be inferred through reasoning.
causes Expresses a constant causal relation without any conditional constraints.	**ObjectProperty / Class-level axioms**	Similar to `leadTo`, **causes** can be modeled using an **ObjectProperty** that connects an event to its effect.
obstructedBy A is a goal that can potentially be hindered by B, where B acts as an obstacle to hinder the realization of A. For example, "sleep" → "noise".	**Negative ObjectProperties**	Causal obstructions can be modeled through **inverse relationships** or by introducing **negative object properties** (e.g., `obstructs`).

causesDesire A triggers a desire or need for B in a person, where the state or event of A stimulates a desire or need for B. For example, "hunger" → "go to the store".	**ObjectProperty** (desire-related)	This could be modeled using **ObjectProperties** to connect events or states with desires or needs (e.g., "hunger causes the desire for food").
createdBy B is a process or motive that creates A. For example, "cake" → "baking".	**ObjectProperty** (creator)	`createdBy` can be represented as an **ObjectProperty**, linking an event to the process or action that created it.

SEQ: Sequential relation

Sequential relationships (like "happened before") are modeled using **transitive object properties** in OWL.

OpenSPG Relationship Type (Descriptions from OpenSPG)	**Possible OWL Equivalent / Concept**	**Explanation**
happenedBefore A occurs before B.	**Transitive ObjectProperty**	Sequential dependencies are often modeled using **transitive object properties** (e.g., `happenedBefore`).
hasSubevent A and B are events, where B is a sub-event that occurs as part of A. For example, "eating" → "chewing".	**SubClassOf** / **ObjectProperty**	Sub-events can be modeled using `hasSubevent` as an **ObjectProperty**, or events can be subclassed as part of a larger event using `rdfs:subClassOf`.
hasFirstSubevent A is an event that begins with sub-event B. For example, "sleeping" → "closing eyes".	**SubClassOf** / **ObjectProperty**	Similar to **hasSubevent**, but specifically marking the **first** subevent in a sequence.
hasLastSubevent A is an event that ends with sub-event B. For example, "cooking" → "cleaning the kitchen".	**SubClassOf** / **ObjectProperty**	Similar to **hasSubevent**, but specifying the **last** subevent.

hasPrerequisite In order for A to occur, B needs to occur; B is a prerequisite for A. For example, "dreaming" → "sleeping".	**ObjectProperty** (precondition)	Prerequisites are modeled as **ObjectProperties** where an event depends on the occurrence of another event.

IND: Induction relation

`belongTo` in OpenSPG is similar to `rdf:type` in OWL, where entities are classified under broader categories.

OpenSPG Relationship Type (Descriptions from OpenSPG)	**Possible OWL Equivalent / Concept**	**Explanation**
belongTo This relation is commonly used in SPG to describe the classification relation from entity types to concept types. For example, "company event" → "company event category".	**rdf:type** / **rdfs:subClassOf**	This is used for categorizing entities into classes. OWL uses `rdf:type` to indicate class membership.

INC: Inclusion relation

Part-whole relationships are modeled using `isPartOf` (e.g., a "wing" is part of a "bird"). **OWL** supports this through **ObjectProperties**.

OpenSPG Relationship Type (Descriptions from OpenSPG)	**Possible OWL Equivalent / Concept**	**Explanation**
isPartOf A is a part of B.	**ObjectProperty**	This is equivalent to `isPartOf` in OWL, represented as an **ObjectProperty** (e.g., "wing is part of bird").

hasA B belongs to A as an inherent part or due to societal constructs. HasA is often the reverse relation of PartOf. For example, "bird" → "wing".	**ObjectProperty**	This is often the reverse of **isPartOf**, so `hasA` is typically modeled as an **ObjectProperty**.
madeOf A is made up of B. For example, "bottle" → "plastic".	**ObjectProperty / DataProperty**	`madeOf` is represented as an **ObjectProperty**, typically linking an object to its material or component.
derivedFrom A is derived from or originated from B, used to express composite concepts.	**ObjectProperty**	`derivedFrom` expresses a **part-whole** or **origin relationship** and is represented as an **ObjectProperty**.
hasContext A is a word used in the context of B, where B can be a subject area, technical field, or regional dialect. For example, "astern" → "ship".	**AnnotationProperty**	**Contextual relationships** (e.g., "astern" related to ships) are often captured using **AnnotationProperties** in OWL.

USE: Usage relation

Usage relations like `usedFor` and `capableOf` can be modeled using **ObjectProperties** in OWL.

OpenSPG Relationship Type (Descriptions from OpenSPG)	**Possible OWL Equivalent / Concept**	**Explanation**
usedFor A is used for B, where the purpose of A is B. For example, "bridge" → "crossing over water".	**ObjectProperty**	This is equivalent to an **ObjectProperty** linking entities to their intended use (e.g., a "knife" used for "cutting").

capableOf A is capable of doing B. For example, "knife" → "cutting".	**Functional ObjectProperty**	Represents a **capability**, usually modeled as an **ObjectProperty** (e.g., "knife" capable of "cutting").
receivesAction B is an action that can be performed on A. For example, "button" → "press".	**ObjectProperty**	This can be modeled as an **ObjectProperty** linking entities with actions they can receive (e.g., "button" → "press").
motivatedByGoal Someone does A because they desire outcome B; A is a step towards achieving goal B. For example, "competition" → "winning".	**ObjectProperty**	This is similar to expressing goal-directed actions, typically using an **ObjectProperty** in OWL.

Logs

```
docker logs -f release-openspg-server

docker logs -f release-openspg-mysql

docker logs -f release-openspg-neo4j

docker logs -f release-openspg-minio
```

Validation of the schema: AssertionError: Line# 235: TaxonomyNetworkElement is illegal, please ensure that it appears in this schema

TaxOfProduct doesnt exist in the phsical schema

Network Element addition didnt work

Change the table

Change the schema

change the data

change indexer

What is not/working between previous and current versions OpenSPG?

OpenSPG 0.5	OpenSPG 0.0.3
Possible to create Index in the schema	Index doesn't work
There are scanners for data import (e.g. jsonscanner, csvscanner)	No scanners
Only supports for Neo4j	Supports TUgraph and Neo4j
Project can be deleted curl http://127.0.0.1:8887/project/api/delete?projectId=1	No possibility to delete project