Chapter 3 Semantic Enhancement Programmable Framework (SPG)

To address the issues mentioned in Chapter 2, we have developed the semantic representation framework (SPG) based on property graph, taking into account the characteristics of enterprise-level business scenarios. This framework defines and represents knowledge semantics from three aspects. Firstly, SPG provides a formalized representation and programmable framework for "knowledge" to be defined, programmed, and understood by machines. Secondly, SPG enables compatibility and progression between knowledge hierarchies, supporting the construction and continuous iterative evolution of the knowledge graph in industrial scenarios with incomplete data. Lastly, SPG effectively bridges the gap between big data and AI technology systems, enabling efficient knowledge transformation of massive data to enhance data value and application value. With the SPG framework, we can construct and manage knowledge more efficiently, while better supporting business needs and application scenarios. Due to its scalability and flexibility, the SPG framework allows for quick construction of domain models and solutions for new business scenarios by extending domain knowledge models and developing new operators.

3.1 The semantic model of SPG

The overall semantic model of SPG is illustrated in Figure 5 of the Chapter 1, and briefly described in Chapter 1 as well. Firstly, SPG formally defines knowledge from the following three dimensions:

1) **Domain Type and Structure Constraint:** In the objective world, there are no things without domain types. However, in the digital world, there are numerous text/numeric representations without domain types. SPG DC requires that everything must have a distinct domain type (Class) and the domain type must have its own inherent structural representation, including properties, relations, etc., which are associated with other things through relations. Additionally, based on the principles of dynamic to static, specific to general, and instance to concept in domain knowledge, SPG DC classifies domain types into Event HyperGraph, Entity, and Concept. This facilitates efficient knowledge classification and reuse in business and achieves automatic hierarchical separation of knowledge from dynamic to static. For detailed information, please refer to the descriptions in sections 4.1.1 and 4.2.

2) **Unique Instance within the Domain:** In the objective world, there are no two things that are exactly the same. However, in the digital world, there are numerous instances of the same thing due to data copying, different descriptive perspectives, and multiple heterogeneous sources. To ensure consistent representation between the digital world and the objective world, SPG requires that every instance within a domain type must be unique to guarantee the accuracy and consistency of the knowledge. To achieve this, SPG Evolving provides programmable capabilities for entity linking, property standardization, and entity resolution through SPG-Programming. Users can use built-in or self-developed algorithms (operators) to improve the uniqueness of instances. More

detailed descriptions are expected to be released in the SPG White Paper 2.0, and relevant information can be found in Chapter 7.2: SPG-Programming.

3) **Logical Dependency between Knowledge:** In the objective world, there are no things that are not related to other things. We often understand things through their connections with other things. These connections represent the intrinsic characteristics of things as well as the logical/physical relations with other things. They encompass both the general commonality of inductive significance and the specific uniqueness at the instance level. SPG defines dependencies between knowledge through the SPG Reasoning predicate/logic system, including logical dependencies and inference between properties, relations, types, etc. Additionally, SPG defines basic predicate primitives through the predicate system to support knowledge reasoning and inference. This allows for better handling the associations and dependencies between knowledge, and supports modeling and analysis of the complex business scenarios. Detailed descriptions can be found in sections 4.3 and 4.4.

Furthermore, the SPG framework achieves compatibility and progression between knowledge hierarchies to adapt to industrial-level knowledge graph. In practical applications, businesses often face the objective reality of incomplete datasets, incomplete expert experiences, and incomplete understanding of the knowledge graph. On one hand, businesses expect to quickly realize business value through the knowledge graph. On the other hand, the coverage of business data and the experience of the knowledge graph are also incomplete, requiring continuous business iterations to gradually deepen the understanding and application of the knowledge graph. However, RDF/OWL requires complete knowledge exchange, which is inconsistent with the objective reality of practical application scenarios. To address this issue, SPG requires compatibility and progression from left to right when defining knowledge representation. Users can choose the simplest SPG Compatible mode to directly construct the representation of property graph from the big data system, or they can enhance the semantic clarity of the subject model by adding SPG DC domain model constraints. Additionally, users can continuously improve the uniqueness of subjects and the semantic associations between subjects by adding entity linking and entity resolution operators through SPG Evolving. Finally, a symbolic representation of knowledge is constructed through complex predicate and logic systems. Through the layered compatibility and progression of SPG, the cost of implementing knowledge graph business can be greatly reduced. In the process of knowledge graph application, users can gradually improve and optimize the domain knowledge graph by selecting different modes and operators based on their own needs and data conditions.

In conclusion, the SPG framework effectively bridges the gap between big data architecture and knowledge systems, enabling the automatic construction of knowledge systems from big data systems. Specifically, by employing the ER2SPG approach to transform data from the big data system into the SPG knowledge graph representation, seamless integration of data and knowledge can be achieved. Furthermore, the SPG-Reasoning component enables the construction of a machine-understandable symbolic system, which facilitates the linkage with deep learning models through knowledge constraints, logical symbols, etc.,

thereby providing more possibilities for knowledge graph applications. Additionally, the SPG framework aims to establish a symbolic linkage with large language models (LLMs) through SPG-Reasoning. By mapping the output of LLMs into the symbolic representations and inputting the symbolic representation of the knowledge graph into the LLMs, better integration and collaboration between knowledge and models can be achieved, enabling efficient interaction and co-evolution between knowledge and models. This is of great significance for achieving more intelligent application scenarios.

In summary, the SPG framework enables the automatic transformation and application of data into knowledge by bridging the big data architecture and constructing a machine-understandable symbolic system. In the future, the SPG framework will continue to leverage its advantages, explore more application scenarios, and establish closer linkage with LLMs, bringing more possibilities to knowledge graph applications.

3.2 SPG Layered Architecture

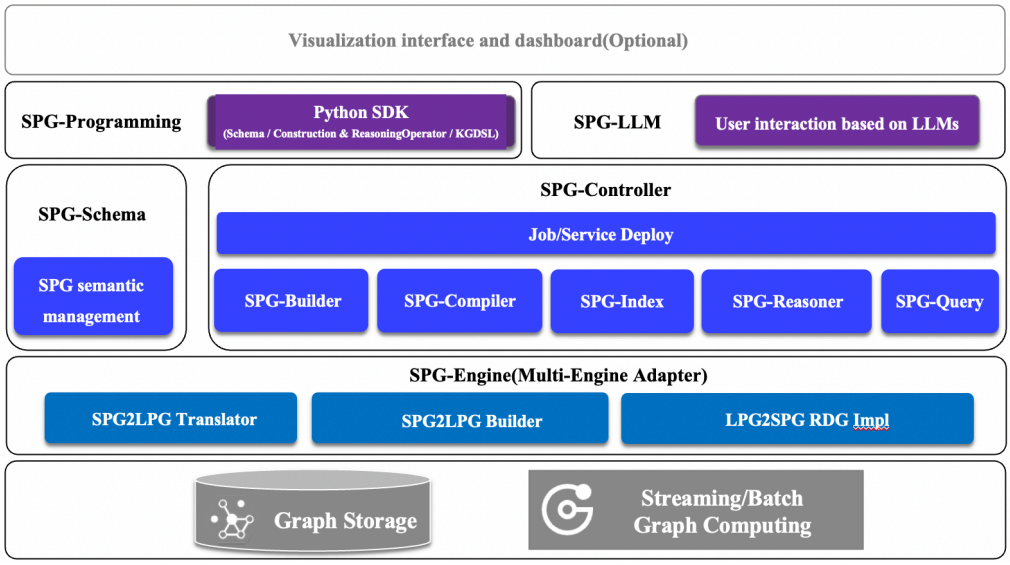

Figure 23: Overall Architecture of the Knowledge Engine based on SPG

The core objective of SPG is to build a standardized knowledge engine architecture based on SPG, providing clear semantic representation, logical rule definition, operator framework (construction, reasoning), etc., for the domain knowledge graph construction. It supports pluggable adaptation of the basic engines, algorithm services, and the solution construction by various vendors. This section provides a brief overview of the overall framework.

- SPG-LLM: Responsible for the interaction subsystem with LLMs (Large Language Models), such as natural language understanding (NL), user instructions, queries, etc. See Chapter 8 for more details.
- SPG-Schema: Responsible for the design of the Schema framework that enhances the semantic understanding of the property graph, including subject models, evolution models, predicate models, etc. See Chapter 4 for more details.

- SPG-Controller: Responsible for the design of the control center subsystem, including control framework, command distribution, plugin integration, etc. See Chapter 6 for more details.
- SPG-Programming: A programmable framework subsystem responsible for the design of the SDK framework and compilation submodules, such as knowledge construction, knowledge evolution, expert experience projection, knowledge graph reasoning, etc. See Chapter 7 for more details.
- SPG-Engine: Knowledge graph engine subsystem responsible for the design of the integration/adaptation layer for multiple engines, such as reasoning engine, query engine, etc. See Chapter 5 for more details.

3.3 The Objectives of SPG

We aim to build a next-generation cognitive engine infrastructure based on SPG, as shown in Figure 24, which represents the overall capabilities. The legend in the figure also indicates the coverage of this whitepaper.

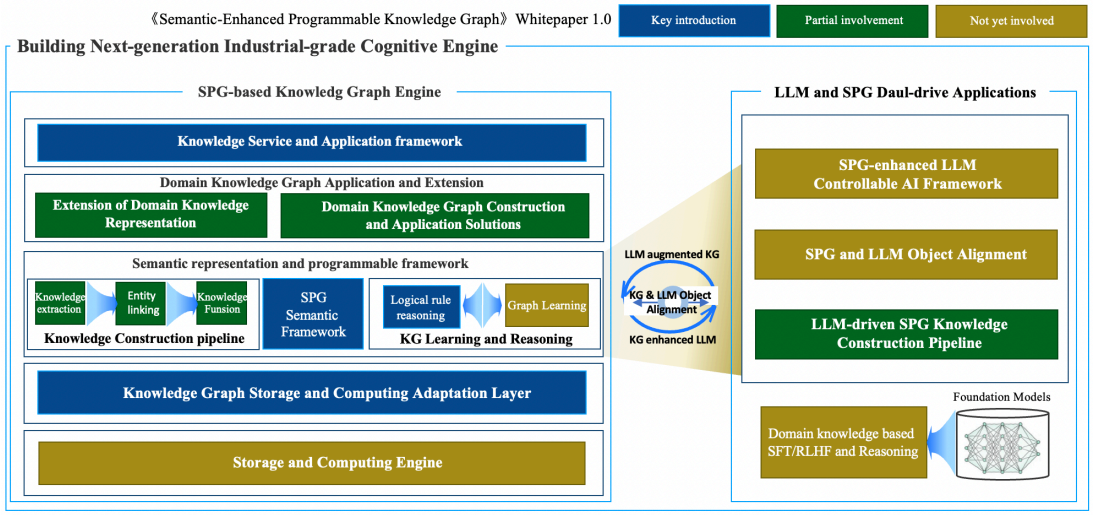

Figure 24: Target Architecture of SPG and LLMs Bidi-Driven (Draft)

This whitepaper, titled “Semantic-Enhanced Programmable Knowledge Graph (SPG) 1.0”, is the initial release. It discusses the current pain points and possible solutions in the development of knowledge graph, as well as the proposed approach, core capabilities, and overall framework of SPG, as described in Chapter 2. In the future, SPG will continue to improve the content of the whitepaper, including domain model extensions, programmable framework, knowledge construction engine, knowledge reasoning engine, and the bidirectional interaction between LLMs and KG. Additionally, SPG will accelerate the open-source development of semantic and basic engine frameworks, promoting the industrial implementation of knowledge graph. The 1.0 version of the whitepaper will focus on the following topics:

- **SPG Semantic Foundation Framework:** Introduces the background of SPG, the core problems it addresses, and presents the semantic framework and schema model of SPG through two business cases.

- **SPG Logical Rule Framework:** Introduces the logical rule system of SPG and how it organically integrates logical rules with factual knowledge based on SPG.
- **SPG Multi-Engine Adaptation Layer:** Provides a detailed introduction to the capabilities of the adaptation layer, incorporating the adaptation abstractions of SPG2LPG and LPG2SPG, to facilitate the efficient integration of the graph storage and the graph computing engines developed by various vendors.

We will continue to update the whitepaper, including versions 2.0 and 3.0. In this release, certain topics, such as the programmable framework and knowledge reasoning, have only been briefly introduced. In the future, we will focus on these topics in separate discussions. We will also continue to explore and make breakthroughs in the bidirectional interaction between SPG and LLMs. The release plan for the future of SPG is outlined in the Chapter 11.

Chapter 4 SPG-Schema Layer

4.1 Overall Architecture of the SPG-Schema

The core objective of SPG is to leverage the advantages of the property graph compatibility with the big data architecture, and address practical problems in industrial practice to achieve semantic enhancement and build a comprehensive semantic system. This chapter provides a detailed explanation of two aspects: the extension of the SPG DC subject classification model and the extension of semantic predicates in SPG Reasoning. First, extending the subject model based on the definition of schemas or fields in the big data tables is the most direct and flexible approach. It involves mapping the columns or fields of the table model to the types, properties, and relations of the SPG subject model. This mapping allows for the integration of data from multiple heterogeneous sources into an incomplete subject structure. Next, the iterative evolution of the incomplete subject structure is carried out to achieve the extension of the logical predicate semantics. In this process, SPG draws inspiration from the minimal usable set of ρdf and the logical predicate capabilities of OWL. It defines the minimal semantic units of the SPG subject model and expands the expression of SPG in terms of the predicate semantics and the logical rules.

4.1.1 Extension of the Subject Classification Model

To enhance the semantic expression of the node types in LPG, SPG extends and introduces additional subject classification models on the node types and edge types in LPG. This expansion aims to accommodate a more diverse representation of knowledge. The expanded subject types include standard types, concept types, entity types, event types, and more. The domain classification model of SPG is shown in Figure 25.

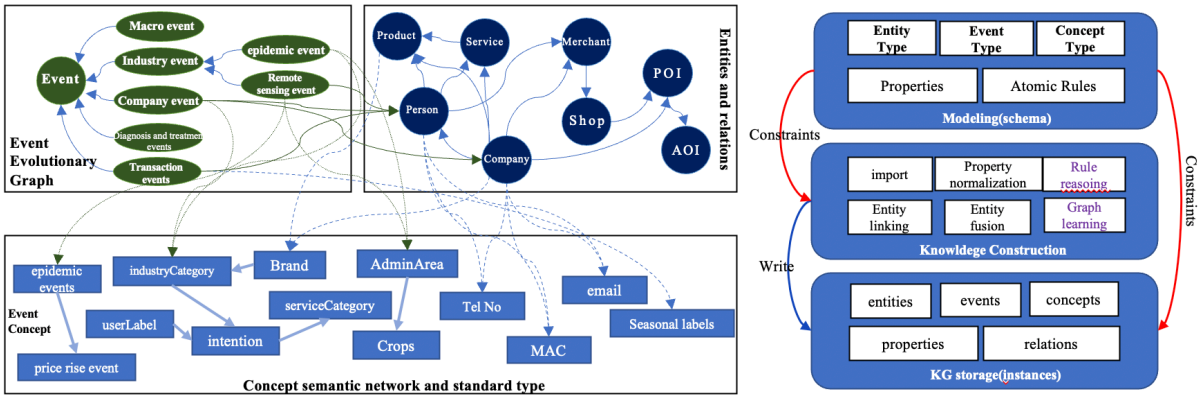

Figure 25: SPG Domain Classification Model

A brief explanation of the SPG subject classification model is as follows:

- **Entity:** Objective objects with strong business relevance, represented by a complex structure characterized by multiple properties and relations, such as users, companies, merchants, etc.
- **Concept:** Abstractions from concrete entities to general ones, representing a group of entity instances or event instances, forming a classification system. Concepts are relatively static and

represent common knowledge with strong reusability, such as audience labels, event classifications, administrative divisions, etc. To simplify the enterprise applications, standard types are also included in the concept category.

- **Event:** Temporal and spatial multi-dimensional types with constraints (such as time and space). For example, the industry events, company events, medical events extracted through NLP, CV, or the user behavior events generated from actions like purchasing, redeeming, registering, etc.
- **Property:** Properties are the components of the entities, events, concepts, etc., used to describe the individual elements of a complex structure. Each property element can be associated with a specific simple or complex structure, such as base types, standard types, concept types, etc.
- **Relation:** Relations are defined similarly to the properties, and express the association between a complex object and the other objects. The difference between the relations and the properties is that relations involve entity types as the associated objects.

1. Entity Types

Entity types are the basic unit of types in SPG. They are composite data types composed of multiple properties and relations, and are directly extended from the Node types in LPG. In the Chapter 2, we conducted an in-depth analysis of the challenges currently faced in knowledge management with LPG. To address the high data preparation costs and the lack of semantic capabilities in property type in LPG, SPG-Schema extends the expression of property value types on the LPG Node types. The type of the value can be standard type, concept type, entity type, etc. In order to achieve inheritance and reuse of the entity types, we draw inspiration from and extend the semantic of the "subClassOf" predicate to enable subclass inheritance of properties and relations from the parent classes. To address the issue of inconsistent naming of the same entity type due to heterogeneity, we support the fusion of the entity types and align the logical alignment of the entity types in different knowledge graphs through entity linking and entity resolution operators. In the future, we will focus on releasing the operator binding section in the SPG Whitepaper 2.0.

2. Concept types and Event types

In the schema of the domain knowledge graph, there is a subjective design issue due to different internal demands in different business domains. This leads to the existence of multiple similar types for the same entity, as different businesses may have different naming and granularity requirements for these types. However, the data for these similar types all come from the same source, which severely affects and hinders knowledge dissemination and causes inconsistencies between knowledge and data. To avoid such inconsistencies, SPG-Schema introduces concept types to classify the similar types and resolves knowledge heterogeneity by linking concepts with basic entity types.

In addition, the event model in the enterprise causal knowledge graph involves multiple dimensions of time and space. When modeling the causality layer, it is necessary to associate simple or complex logics such as causality, sequence, co-occurrence, and structure. To address the inability of LPG to perfectly

express these requirements, SPG-Schema introduces the concept of events to extend the classification model and better express the horizontal and vertical associations between the events, entities, and concepts.

The Concept-Event Quadrant Diagram, as shown in Figure 26, describes the associations between the entity types, concepts, and events based on the principles of domain knowledge transitioning from dynamic to static, from specific to general, and from instances to concepts. At a more specific level of definition and instantiation, the Concept-Event Quadrant Diagram can be divided into four components: abstract entities, concrete entities, abstract events, and concrete events. Physically, they are interconnected and form a unified graph.

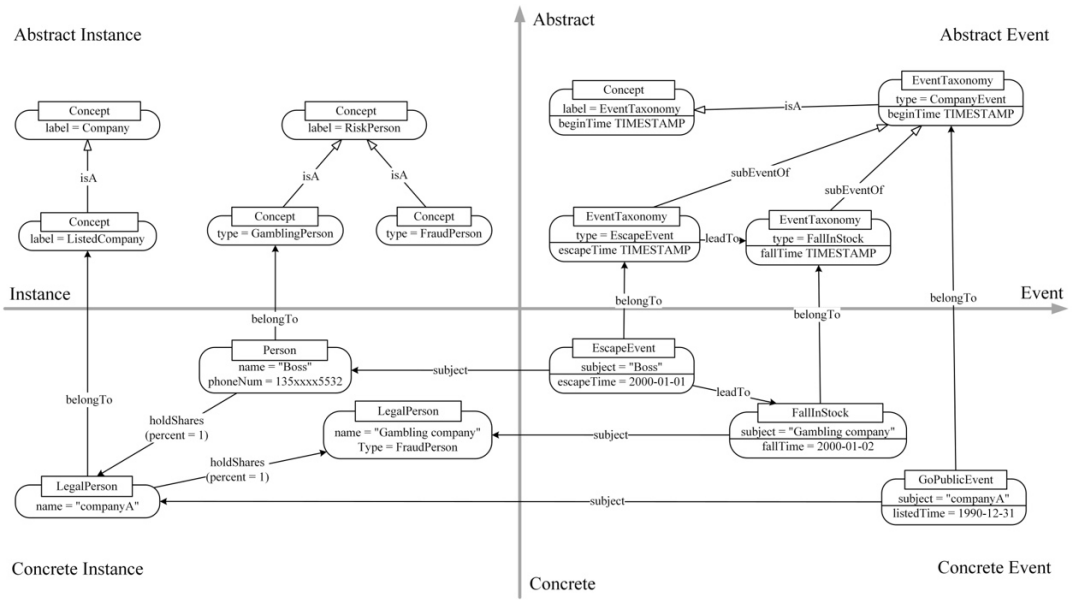

Figure 26: The Concept-Event Quadrant Diagram

Specifically, the quadrants are divided horizontally into the entity domain and the event domain, and vertically into the abstract domain and the concrete domain. The vertical division can also be referred to as the concept domain and the instance domain. The abstract entity type represents the abstract concept of the concrete entity types, while the concrete entity refers to the specific instantiation of the entity types. The abstract event represents the abstract concept of the events to express causality and sequence between the events, while the concrete event corresponds to the specific instantiation of the event types.

3. Standard Properties

In the RDF/OWL model, each entity, relation, and property needs to be modeled independently. While the syntax elements of the property graph are simpler compared to RDF, they are merely declarations of data structures and cannot effectively utilize property knowledge for knowledge dissemination. In the actual business implementation process, it is often necessary to use the properties for knowledge dissemination and analysis. However, in LPG, the properties without explicit domain type constraints are simply literal texts or numbers. This not only fails to ensure the integrity and correctness of the property, but also makes it difficult to implement effective queries and propagation based on the structure. Leads to the additional modeling for the properties and significant data preparation costs due to the variations in property scales. As

the demand for knowledge graph-based association analysis increases, these costs are expected to rise even further.

To balance the tradeoff between RDF and LPG regarding properties and effectively reduce data preparation costs, SPG-Schema introduces standard property types to simplify data dependencies. The application of standard properties can automatically materialize textual properties into relations, increasing the ability to propagate knowledge and implicit associations. Since the standard properties are used instead of the relation modeling, explicit definition of the relations is not required. The relation propagation between entity types is achieved through the semantic propagation of the standard properties.

4.1.2 Expansion of the Semantic and Rule Reasoning Capability

In the general modeling process, LPG only consists of two elements: Node and Edge. However, the properties of these elements are often in the form of text/strings, which can lead to various issues in practical business scenarios. In order to achieve more efficient knowledge propagation and inference on top of the expanded 5-category classification model, SPG-Schema introduces a series of semantic predicates to constrain LPG, enabling more syntax and semantics. The specific semantic syntax hierarchy diagram can be seen in Figure 27.

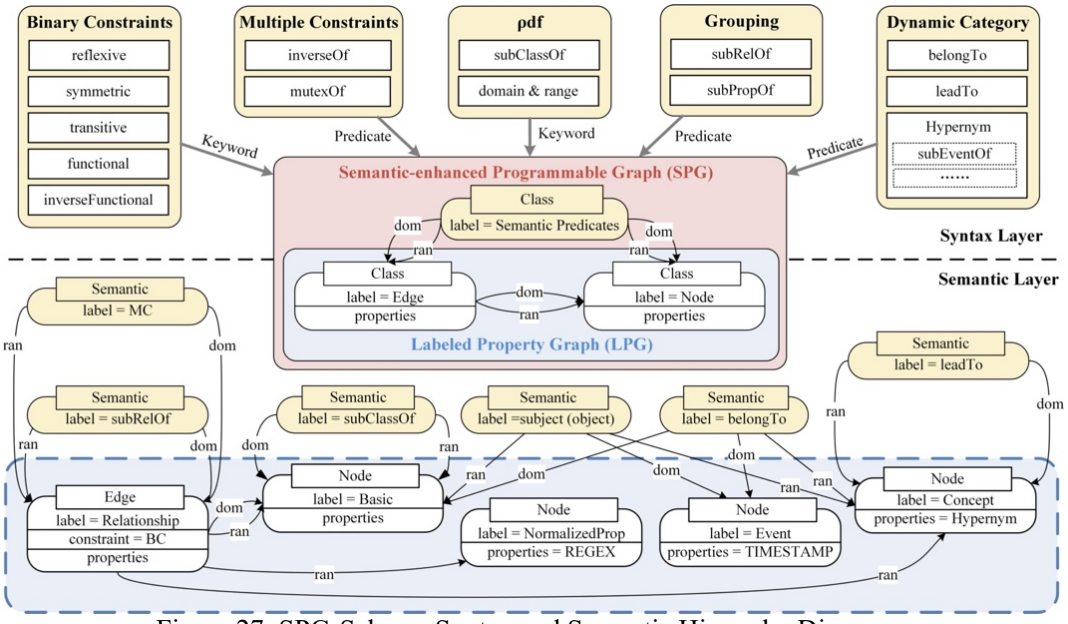

Figure 27: SPG-Schema Syntax and Semantic Hierarchy Diagram

- Syntax Layer: In this layer, the syntax content of SPG-Schema is defined. The related syntax used for semantic reasoning in SPG-Schema can be divided into five categories: minimal constraint set (ρdf), binary constraints (BC), multiple constraints (MC), relation group constraints, and dynamic types. These can be applied to SPG-Schema in the form of Keywords and built-in Predicates from the standard namespace std, depending on the usage scenario.

- Semantic Layer: In this layer, the specific domain (dom) and range (ran) of the semantic reasoning capabilities in SPG-Schema are defined. This reflects the association between the built-in predicates of the reasoning rules and the refined entity classification model.

4.1.3 The Four-Layer Architecture of SPG-Schema

For the overall framework of SPG-Schema, starting from four basic contents, the requirements are gradually expanded and decomposed to determine the content included in SPG-Schema Core based on the semantic completeness and the practical industrial needs. Lightweight syntax is used as much as possible to avoid high complexity, ensuring that the complexity of SPG-Schema does not exceed PTIME and guaranteeing efficiency in industrial-level implementation. The balance between semantic complexity and business application cost is achieved.

The MOF architecture is a layered metadata architecture, and based on this structure, we can also divide the overall modeling hierarchy of SPG-Schema into four layers, as shown in Figure 28.

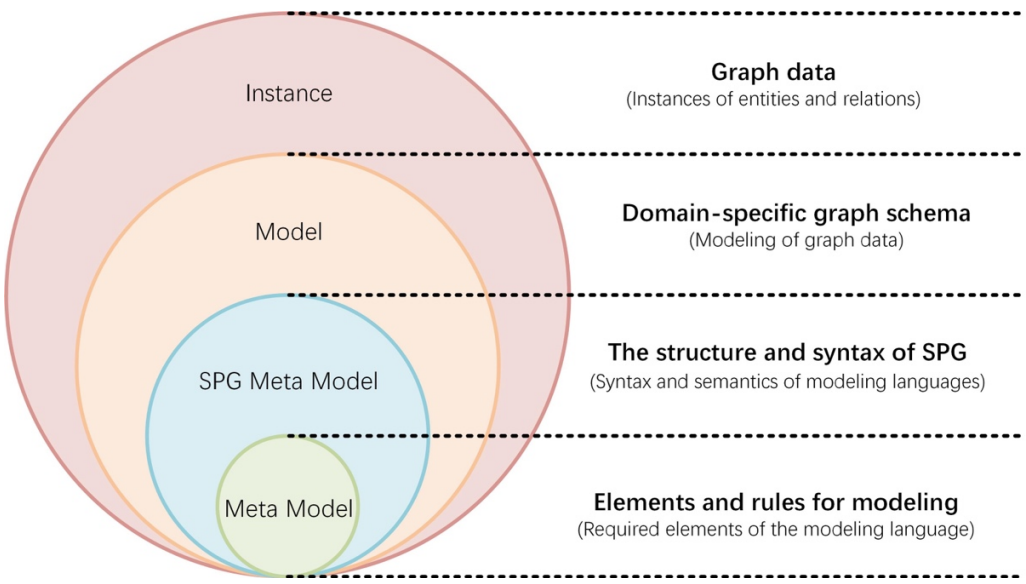

Figure 28: The Four-Layer Architecture of SPG-Schema

Based on the above summary, combined with the risk mining knowledge graph and the enterprise causal knowledge graph as the basic scenarios, the overall four-layer architecture of SPG-Schema is shown in Figure 29.

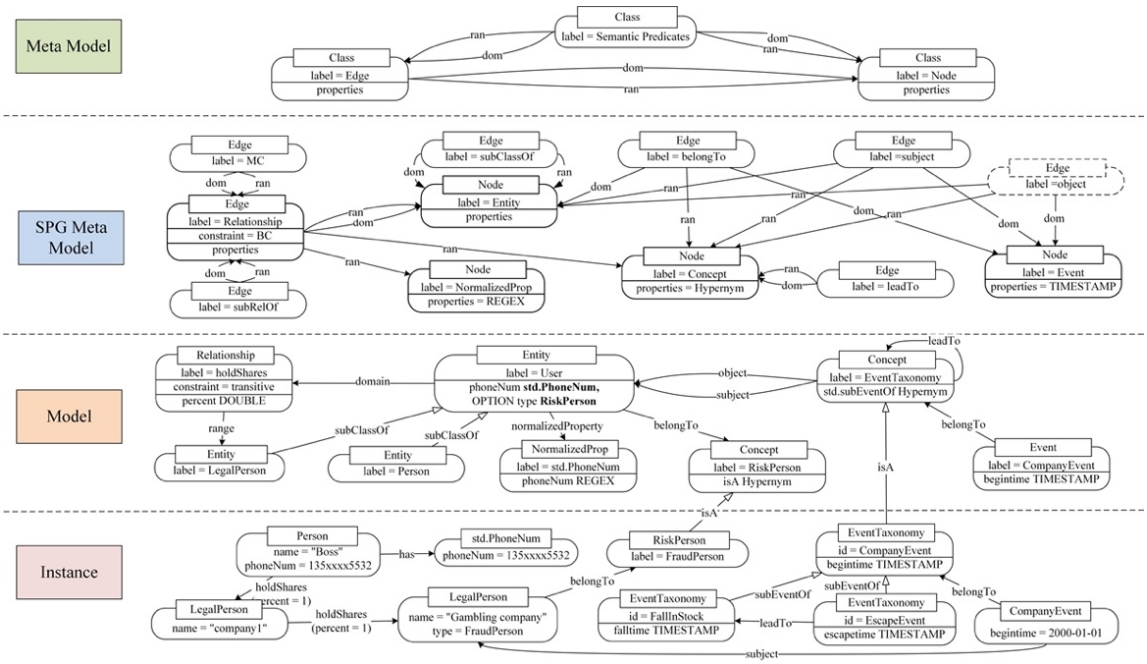

Figure 29: SPG-Schema Four-Layer Architecture Diagram

The explanation of the four-layer architecture is as follows:

- Meta Model: This layer provides an overall definition of the SPG Meta Model paradigm. It defines the elements needed for modeling with SPG-Schema, but it is transparent to general users.
- SPG Meta Model: This layer instantiates the Meta Model layer and provides a more detailed classification of types and predicates. It defines the structure and syntax of the modeling language, adds more semantics to the modeling in the Model layer.
- Model: This layer defines a specific system model. Users can define the classes, properties, relations, and their semantics that can be perceived by the Instance layer using the syntax. The data stored in the Model layer represents the modeling in the Instance layer, where the data is the instance of the model in the Model layer.
- Instance: SPG-Schema adopts the Instance-Class separation paradigm, where each Instance must strictly adhere to the constraints defined in the Schema of the Model layer. This layer is the largest and most concrete layer. The contents of the Instance layer are specific instances.

4.2 Semantic Enhancement of Nodes and Edges

The main syntax design of the SPG-Schema extension is achieved by introducing additional keywords based on the PG-TYPES [16]. Therefore, the main semantics of SPG-Schema are also divided into three categories: NodeType, EdgeType, and GraphType, which correspond to the definition of nodes (Node), edges (Edge), and views in LPG. The basic definition of the semantics is located in the SPG Meta Model layer, and it adds more related semantics to the Meta Model layer by classifying the LPG node types more finely.

4.2.1 Syntax and Semantics for the NodeType

Based on the definition in section 4.1.1, we have classified the commonly used node types in the knowledge graph into entity type, standard type, concept type, and event type. we will introduce the basic syntax and semantics for these four types.

1. Entity Type

Using the syntax "CREATE ENTITY TYPE", you can define a node type (Class) and the labels and property types that appear in the node type. For example, you can create a "User" class using the following syntax:

```
// Definition of the User entity
CREATE ENTITY TYPE ( User {
  phoneNum std.PhoneNum,
  OPTIONAL taxonomy RiskPerson,
  OPEN
});
```

In the above example, the User type is defined with two properties: "phoneNum" and "taxonomy". The type of "phoneNum" is "std.PhoneNum", and the type of "taxonomy" is "RiskPerson". The OPTIONAL keyword indicates that the "taxonomy" property is optional. The OPEN keyword indicates that when defining the model and populating instance data, additional property fields can be added.

In some cases, it may also be necessary to declare a type as an abstract type, which means that it cannot be directly instantiated. In the example above, the "User" type can be annotated as an abstract entity type using the ABSTRACT keyword. Therefore, when populating instance data, it cannot be directly populated under this type, but rather in the subtypes that inherit from this abstract parent class. This type may not be very useful, but it can be helpful for reusing shared properties among multiple subclasses.

In addition to the above three keywords, we can add more semantic information and constraints to the node definitions by referring to the keyword constraints in PG-Keys [17]. The keywords EXCLUSIVE, MANDATORY, and SINGLETON are used to represent unique, at least, and at most constraints, respectively. This further enhances the semantic constraints on entity properties in SPG-Schema. In the following example, we will modify the "User" type to be an abstract type and apply the three property constraint keywords.

```
// Extend the definition of the User type as an abstract type
CREATE ENTITY TYPE ABSTRACT ( User {
  EXCLUSIVE idcard STRING,                    // Each instance should have a unique ID card number
  MANDATORY name STRING,                      // Each instance should have at least one name
  SINGLETON birthday DATE,                    // Each instance can have at most one birthday
  OPTIONAL phoneNum std.PhoneNum,             // Optional field to add phone number with standard property
  OPTIONAL SINGLETON taxonomy RiskPerson,     // Optional concept classification property
  OPEN
});
```

When the three keywords (EXCLUSIVE, MANDATORY, and SINGLETON) are used together with the OPTIONAL keyword in type constraints, the former is used to constrain the data filling in the instance layer, while the latter corresponds to whether the instance can use that property. They are not conflict with each other. For example, “OPTIONAL SINGLETON type RiskPerson” can be used to indicate that the data may not include the concept classification property “taxonomy”, but once this property is added, the instance can have only one classification property.

2. Standard Types

To distinguish the standard types from the user-defined types, the concept of namespaces is introduced, with the “std” namespace representing the standard types. All the standard types are defined within the “std” namespace and created using regular expressions. Since standard types are more associated with property standardization and are derived from property, they typically only have one property. After defining a standard property using the syntax “CREATE NORMALIZED TYPE”, you can use it directly or use the “SET PROP” statement and “NORMALIZED” to normalize the properties. It is important to note that standard properties should be used in conjunction with regular expressions guided by the “REGEX” keyword to constrain the pattern/format of the property values.

```
// Definition of the standard type Email
CREATE NOMALIZED TYPE (std.Email {
  value STRING REGEX '[a-zA-Z0-9_\-\.]+@[a-zA-Z0-9_\-\.]+\.[a-zA-Z]{2,3}'
});
// Definition of standard type phoneNum
CREATE NOMALIZED TYPE (std.PhoneNum {
  value STRING REGEX '/^(13[0-9]|14[01456879]|15[0-35-9]|16[2567]|17[0-8]|18[0-9]|19[0-35-9])\d{8}$/'
});
// Modify the phoneNum property in the User class to be a standard property
SET PROP (User.phoneNum) NORMALIZED std.PhoneNum;
```

Standard types are also one of the most important aspects of property standardization in SPG-Schema. Unlike regular LPG properties, when the property in SPG is defined as a standard type, and the semantic of that standard type is propagable, the property will be automatically converted into a relation, and the value of the property will be treated as an instance of the standard type. This generates more meaningful information for semantic reasoning. In this example, we modify the “phoneNum” property of the “User” type to be a pre-defined standard property “std.PhoneNum”. Additionally, users can directly specify properties as standard types when creating node types.

3. Concept Types

When creating a concept type, you can use the “OPTIONAL” keyword to mark certain properties as optional, while the remaining properties are considered mandatory by default. By using this type, you can associate entitys, properties, and other elements with relevant business conceptual domains, enabling further business inference.

```
// Definition of the concept of company classification
CREATE CONCEPT TYPE (CompanyTaxonomy {
```

```
  isA std.Hypernym,
  OPTIONAL beginTime TIMESTAMP,
  OPEN
});
```

In the example above, a concept type called “CompanyTaxonomy” is created as a conceptual domain. When populating concept instances, they will categorized to this concept type. The property “isA”, of type “std.Hypernym”, is a mandatory property within this conceptual domain, representing the hierarchical relation between concept instances in the domain. The field “begintime” is an optional property. More details about the “std.Hypernym” type will be discussed in section 4.3.5 with further elaboration.

4. Event Types

When creating an event type, there are mandatory and optional requirements for time, subject, and object. The “REQUIRED” keyword is used to indicate mandatory requirements for the fields, while the remaining fields are considered optional. It is important to note that event types must have a timestamp property and a specified subject type by default. Therefore, we can define an event using the following syntax.

```
// Definition of the company operation event
CREATE EVENT TYPE (CompanyEvent {
  {REQUIRED occurrenceTime TIMESTAMP, OPEN}
  REQUIRED SUBJECT (Company | Person),
  OBJECT (Company | Person)
});
```

In the above example, we define a company operation event. It includes a mandatory timestamp property called “occurrenceTime”, and a mandatory subject type of either “Company” or “Person”. An event should be treated as a graph structure. For example, in the defined company operation event called “CompanyEvent”, it automatically uses the built-in predicate “std.subject”, indicated by the “SUBJECT” keyword, to point to the Company and Person types as the possible subjects of the event. This is a mandatory property, while the “OBJECT” can be empty.

4.2.2 Syntax and Semantics of the EdgeType

The semantics of EdgeType specify the labels, properties, and types of the property values that appear in an edge type. It also specifies the allowed source and target entity types. When using the “CREATE EDGE TYPE” statement to create a relation, it is required that both the source and target entity types have been defined. Otherwise, there will be a dangling mount situation, resulting in a relation error, and the creation will not be allowed.

```
// Definition of the holdShares relation
CREATE EDGE TYPE
(Person)-[holdShares {percent DOUBLE}]->(LegalPerson);
```

In many cases, users may not want to use the triple representation when using the relation. Therefore, we can set relation aliases to directly refer to a triple relation. There are two ways to set aliases: direct

definition and later modification. For the newly defined “holdShares” relation, the former sets an alias directly in the definition using the AS “< >” statement, while the latter uses the “ALTER” keyword to add an alias to the relation that has been forgotten to be set.

```
// Specify the alias "holdSharesType" when defining the relation
CREATE EDGE TYPE
(Person)-[holdShares {percent DOUBLE}]->(LegalPerson)
AS <holdSharesType>;

// Use ALTER to set an alias for the above holdShares relation
ALTER EDGE TYPE
(Person)-[holdShares {percent DOUBLE}]->(LegalPerson)
AS <holdSharesType>;
```

In the definition of an edge type, the “< >” notation is used for the first time to represent the alias of the edge. In SPG-Schema, we use “< >” to quickly refer to a graph. For relation, it essentially represents a graph type composed of node-edge-node relation. In addition to the basic definition, similar to the property constraints mentioned above, the “EXCLUSIVE” keyword can also be applied to relation constraints, which we refer to as cross-type constraints. Since there has a unique source node and target node for each relation instance by default, the “MANDATORY” and “SINGLETON” keywords are not used to constrain relations.

```
// Applied to the source node => an entity cannot simultaneously have multiple outgoing edges of the same relation type.
CREATE EDGE TYPE (EXCLUSIVE Class1)-[ Type { propClause }]->(Class2);

// Applied to the target node => an entity cannot simultaneously have multiple incoming edges of the same relation type.
CREATE EDGE TYPE (Class1)-[ Type { propClause }]->(EXCLUSIVE Class2);
```

By constraining the source and target nodes of an edge, we can easily achieve relation constraints similar to those in relational databases.

```
// One-to-one
CREATE EDGE TYPE (EXCLUSIVE Class1)-[ Type { propClause }]->(EXCLUSIVE Class2);

// one-to-many,
CREATE EDGE TYPE (EXCLUSIVE Class1)-[ Type { propClause }]->(Class2);

// many-to-many
CREATE EDGE TYPE (Class1)-[ Type { propClause }]->(Class2);
```

In addition, we have also added a specific type of constraint called Binary Constraints (BC) to the relation type. These constraints specify the characteristics shared by all individual instances of a specific relation type. We will focus on introducing these constraints in Chapter 4.3.2.

4.3 Semantic Enhancement of the Predicates and Constraints

In order to manage the minimal predicate set and expand the set of built-in predicates in SPG-Schema, a namespace mechanism is used, similar to RDF. This mechanism has already been used when using standard types. By introducing the “std” namespace, built-in predicates can be categorized, ensuring that the existing

basic predicates form the minimal set. When building schemas for different domain knowledge graphs in the future, specific namespaces can be added for different domains. In addition, by defining namespaces at different levels, the semantic and capabilities of the Schema Core can be enriched and improved. This will provide greater flexibility and utility for applications in different industrial environments.

The Model layer defines the perceptible class, property, and relation and their semantics for users. As the abstraction layer of the Model layer, the SPG Meta Model layer needs to define schema structures through a series of constraints. In order to express the semantics of entities, starting from the minimal constraint set (ρdf), in section 4.2, we introduced the core semantics of SPG-Schema. In this section, we will focus on the additional built-in predicates and constraint semantics that we have added to the Meta Model level of SPG Meta Model. The constraints that appear at the Model level are categorized into single binary relation constraints, multiple binary relation constraints, grouping rules, and dynamic types. Currently, the predicates and constraints we are discussing are all stored under the standard namespace “std”.

4.3.1 The Minimal Constraint Set ---- ρdf

The concept of ρdf [18] is derived from the minimal set of predicates in RDF. In order to better meet the needs of SPG, adjustments have been made to retain subClassOf, domain, and range as the minimal constraint set of the SPG-Schema.

1. Entity Type Hierarchy (subClassOf)

The “subClassOf” predicate is used to supplement the semantics of type inheritance by defining hierarchical relations between node types. When defining a node type, inheritance can be achieved by specifying the name of the type preceded by the “SUBCLASSOF” keyword.

```
// Definition of two subclasses of the User
CREATE ENTITY TYPE (Person {age INT, OPTIONAL father Person}) SUBCLASSOF (User);
CREATE ENTITY TYPE (LegalPerson {amount INT, legalId STRING}) SUBCLASSOF (User);
```

By inheriting from the abstract node type (User), both of these subclasses will automatically include the properties and constraints defined in the “User” class. In addition, the “Person” type will include an additional “age” property and an optional “father” property. The “LegalPerson” type will include an additional “amount” property indicating the number of shares held, and a “legalId” property indicating the legal identification number.

It is important to note that when using the “subClassOf” keyword, it is necessary to ensure that the subclass node does not include properties with the same name as those in the superclass node. Otherwise, there will be issues with overwriting and overriding. This applies to both cases: when the name and type are the same, and when the name is the same but the type is different. We consider both of these cases to be errors and they will not be processed.

2. Domain and Range of the Relations

When defining relations, both the source node type and target node type need to be specified. In order to reduce the redundancy and computational costs associated with entity conversion, modifications to the DOMAIN and RANGE should follow the principle of "addition without modification". For example, in the previous example, a relation called "holdShares" was created, which included a property called "percent" representing the percentage of shares held. However, due to an initial improper definition, it was discovered that there is an additional requirement for the relations of the form "(LegalPerson) -[holdShares] -> (LegalPerson)". To meet this requirement, the value domain of the relation needs to be modified to include "LegalPerson".

```
// The defined holding relation mentioned above
(Person)-[holdShares {percent DOUBLE}]->(LegalPerson)
// Adding the domain LegalPerson for it
ALTER <hold_share> DOMAIN (LegalPerson);
// The actual form of the relation after the operation should be
(Person | LegalPerson)-[holdShares {percent DOUBLE}]->(LegalPerson)
```

4.3.2 Single Binary Relation Constraints (Binary Constraints, BC)

In SPG-Schema, the BC features are defined as follows: "Binary constraints (BC), i.e., defined to be (ir)reflexive, (in)transitive, (a)cyclic, (a/anti)symmetric, etc".

These constraints focus more on the properties of a binary relation and are usually defined together when defining the relation. Therefore, we define these constraints as keywords. After defining BC constraints for a relation at the Model layer, all instances of that relation at the Instance layer should comply with these constraints. According to the above definition, BC constraints in SPG-Schema should have the basic properties of reflexivity, symmetry, and transitivity.

Let's assume that several node types, Class1 and Class2, have already been defined. We can create reflexive, symmetric, and transitive relations using the keywords "REFLEXIVE", "SYMMETRIC", and "TRANSITIVE" respectively. The semantic deductions of these definitions are also provided.

```
// Definition of reflexive relation edgeA.
CREATE EDGE TYPE REFLEXIVE (Class1)-[ edgeA { prop STRING }]->(Class1);
// This means that
(a:Class1)-[p:edgeA]->(a:Class1)
--------------------------------
// Definition of symmetric relation edgeB
CREATE EDGE TYPE SYMMETRIC (Class1)-[ edgeB { prop STRING }]->(Class2);
// This means that
(a:Class1)-[p:edgeB]->(b:Class2),
--------------------------------
(b:Class2)-[p:edgeB]->(a:Class1)
// Definition of transitive relation edgeC
CREATE EDGE TYPE TRANSITIVE (Class1)-[ edgeC { prop STRING }]->(Class1);
// This means that
(a:Class1)-[p1:edgeC]->(b:Class1),
(b:Class1)-[p2:edgeC]->(c:Class1)
```

```
--------------------------------
(a:Class1)-[p3:edgeC]->(c:Class1)
```

In addition to the three basic constraints mentioned above (reflexivity, symmetry, and transitivity), in order to achieve semantic completeness, we have also added functional and inverse functional relations based on the OWL syntax. These relations are defined using the keywords "FUNCTIONAL" and "INVERSE_FUNCTIONAL" respectively. The definitions are as follows:

$$FunctionalProperty\colon\ \top \sqsubseteq (\leqslant 1\ r.\top)\ \ (e.g., \top \sqsubseteq (\leqslant 1\ hasMother))$$
$$InverseFuntionalProperty : \top \sqsubseteq (\leqslant 1\ Inv(r).\top)\ \ (e.g., \top \sqsubseteq (\leqslant 1\ isMotherOf^{-}))$$

```
// Definition of functional relation edgeD
CREATE EDGE TYPE FUNCTIONAL (Class1)-[ edgeD { prop STRING }]->(Class2);
// This means that
(a:Class1)-[p:edgeD]->(b:Class2),
(a:Class1)-[p:edgeD]->(c:Class2)
--------------------------------
  (b:Class2) = (c:Class2)

// Definition of inverse functional relation edgeE.
CREATE EDGE TYPE INVERSE_FUNCTIONAL (Class1)-[ edgeE { prop STRING }]->(Class2);
//  This means that
(b:Class2)-[p:edgeE]->(a:Class1),
(c:Class2)-[p:edgeE]->(a:Class1)
--------------------------------
  (b:Class2) = (c:Class2)
```

4.3.3 Multiple Binary Relation Constraints (Multiple Constraints, MC)

To enhance the semantic reasoning capabilities of SPG-Schema, additional support for MC predicates has been added. These predicates are more focused on the relation inference between two binary relations. They are set using the "SET REL" syntax, where the predicate section uses built-in predicates under the std namespace.

Let's assume that two relations have already been defined with the aliases "<Pred1>" and "<Pred2>". We can define relation inversions and mutual exclusions using the built-in predicates "std.inverseOf" and "std.mutexOf" respectively. The semantic deductions of these definitions are also provided.

```
// The two relations are mutually inverse
SET REL <Pred1>-[std.inverseOf]-<Pred2>;
```

The "inverseOf" predicate is used to define an inverse relation. If two relations are defined as inverseOf each other, it essentially means that they are a pair of equivalent inverse relations. For example, the "superior" relation can be defined as the inverseOf the "subordinate" relation. During reasoning, it is possible to utilize the "superior" relation to automatically infer the "subordinate" relation, thereby solving industrial problems arising from complex semantic reasoning. In the example mentioned, "(Class1)-[Pred1]->(Class2)" and "(Class2)-[Pred2]->(Class1)" form a pair of inverse relations.

```
//The two relations are mutually exclusive
SET REL <Pred1>-[std.mutexOf]-<Pred2>;
```

The “mutexOf” predicate is used to define a mutual exclusion relation, where an instance relation can only be chosen from the two defined relation types. In the example mentioned, for the same relation instance ‘s’, it is not possible to have both the “Pred1” and “Pred2” relations simultaneously. In other words, “(Class1)-[Pred1]->(Class2)” and “(Class1)-[Pred2]->(Class3)” can only be mutually exclusive, allowing for a binary choice.

In the MC predicates, the appearance of angle brackets “< > -[ ]- < >” is used for a more user-friendly representation, which aligns with the original usage habits of Cypher users. In this context, the angle brackets “< >” are used as aliases when defining relations. For example, “<Pred1>” can essentially be seen as the triplet relation “(Class1)-[Pred1]->(Class2)”. This greatly simplifies the complexity of expressing relation between multiple binary relations, making the overall syntax more concise and easy to understand.

4.3.4 Relation Grouping

In real-world scenarios, it is often necessary to query a class of similar relations through an abstracted relation. Therefore, it is important to introduce relation grouping predicates to assist users in constructing groups and reduce query costs. The relation grouping can be divided into two parts: relation grouping and property grouping. Since properties can be considered as a kind of relation to some extent, the “SET REL” syntax is uniformly adopted to define relation grouping.

1. Relation Grouping

The first application scenario of classification is relation grouping, which is primarily defined using the built-in predicate “std.subRelOf”. When constructing a group, it is important to ensure the existence of a top-level relation. This top-level relation should be marked with the “ABSTRACT” keyword when creating it using the “CREATE EDGE TYPE” statement. This indicates that no instances can be associated with the abstract grouping relation. Any loaded entity relations should belong to specific relations within this group.

Relation grouping can also be considered as a relation between two binary relations, but it is different from the MC predicates mentioned above. Relation grouping predicates have a distinction in syntax from relation type definitions. While the MC predicates define an equivalence relation between two binary relations, in relation grouping, there is a clear hierarchy between the two relations. Therefore, the syntax for defining relation grouping is in the form of a triplet with arrows: “< >-[ ]->< >”.

```
// Creating a grouping of family relations using the ABSTRACT keyword for annotation
CREATE EDGE TYPE ABSTRACT (Person)-[kinship]->(Person) AS <kinship>;

// Definition of three family relations: isFatherOf, isMotherOf, and conjugality
CREATE EDGE TYPE (Person)-[isFatherOf]->(Person) AS <father>;
CREATE EDGE TYPE (Person)-[isMatherOf]->(Person) AS <mother>;
CREATE EDGE TYPE (Person)-[conjugality]->(Person) AS <conjugality>;

// Definition of relation grouping
```

```
SET REL <father>-[std.subRelOf]-><kinship>;
SET REL <mother>-[std.subRelOf]-><kinship>;
SET REL <conjugality>-[std.subRelOf]-><kinship>;
```

Based on the above definitions, we consider the parent-child relation, mother-child relation, and marital relation as specific relations within the family relation grouping. These three relations can be directly obtained through the "kinship" relation. However, when loading instances, there will not be any triplets belonging to the family relation group. Instead, they should belong to one of the specific relations: parent-child, mother-child, or marital relation.

2. Property Grouping

After property normalization, properties can also be considered as specific relations and can be grouped using the "subPropOf" predicate. However, this requirement does not align with a complete and comprehensive semantic definition. There may still be a need for property grouping even for non-standardized properties.

Therefore, the syntax for property grouping is similar to "subRelOf", as it involves grouping properties. The difference lies in the fact that the top-level grouped property may have its own instance data, and properties are loaded into entities. Therefore, there is no need to define an abstract top-level property in advance using the "ABSTRACT" keyword. Instead, the (type.attribute) pattern is used to specify the desired property under a specific type.

```
//To group transaction aggregated values within the Person type
SET REL (Person.1_day_complaint_rate)-[std.subPropOf]->(Person.day_complaint_rate);
SET REL (Person.7_day_complaint_rate)-[std.subPropOf]->(Person.day_complaint_rate)
```

4.3.5 Dynamic Types

1. The Hypernym Predicate for Concept Hierarchy

Due to the diversity of conceptual domains, different domains may use different hypernyms to express hierarchical relations. Therefore, we support the use of the "Hypernym" predicate to express a class of hypernyms while defining events. This allows for the hierarchical classification of the events in different conceptual domains. This predicate is defined simultaneously when defining the concept type. For example, in the concept of risk personnel classification, we can set the hypernym as "isA", while in the concept of city classification (CityTaxonomy), we can set the hypernym as "locateAt".

```
// Definition of Risk Personnel Classification Concept:
CREATE CONCEPT TYPE (RiskPerson {
  isA std.Hypernym,
  OPEN
});

// Definition of City Classification Concept
CREATE CONCEPT TYPE (CityTaxonomy {
```

```
  locateAt std.Hypernym
});
```

With this definition, in the risk personnel classification concept, there can be instances such as “gambler” isA “risk personnel”. In the city classification concept, there can be instances such as “Chengdu” locateAt “Sichuan” and “Sichuan” locateAt “China”.

In addition, event types are a special type in the schema. They are essentially a graph and are often associated with the event classification concepts. Therefore, it is necessary to use special predicates to constrain the event types. The “std.subEventOf” is the specific predicate used for the event concept hierarchy within the Hypernym predicate. When defining the event concept types, “std.subEventOf” must be used as the hypernym predicate.

```
// Define the concept of company operation events.
CREATE CONCEPT TYPE (CompanyOperationTaxonomy {
  std.subEventOf std.Hypernym,
  OPTIONAL beginTime TIMESTAMP,
OPEN
 });

// At the instance layer: The concept of executive escape event belongs to the concept of company operation events
<EscacapeEvent:CompanyOperationTaxonomy>-[std.subEventOf]-><CompanyEvent:CompanyOperationTaxonomy>;

// At the instance layer: The instance of stock price fall event belongs to the concept of company operation events
<FallInStock:CompanyOperationTaxonomy>-[std.subEventOf]-><CompanyEvent:CompanyOperationTaxonomy>;
```

The “std.subEventOf” predicate applies to the event concept instances at the instance layer. After the event concept hierarchy is defined, the subject and object of a child event must be subclasses of the subject and object types of the parent event, respectively. The child event can also have additional properties apart from the parent event. For example, in the given example, the “EscapeEvent” concept and the “FallInStock” concept both belong to the “CompanyEvent” concept.

2.The belongTo Predicate for Dynamic Types

Dynamic types refers to the practice of associating a concept instance with an entity instance or an event instance, effectively using the concept instance's name as the type of that instance. We primarily use the “SET REL” syntax and the “belongTo” keyword to specify the specific instances (including the entity instances and event instances) and their belonging to specific concept instances.

```
// Instances of basic entity types belong to the concept of risk personnel classification
SET REL <User>-[std.belongTo]-><RiskPerson>;
```

Firstly, we can associate the entity types with the concepts. In the previous context, we defined the concept of “RiskPerson” and the entity type “User”. We can use the “std.belongTo” keyword to create an association indicating that the “User” entity type belongs to the “RiskPerson” classification concept.

```
// Definition of Company Operation Event
CREATE EVENT TYPE (CompanyEvent {
```

```
  {REQUIRED begintime TIMESTAMP}
  SUBJECT (Company | Person),
  OBJECT (Company | Person)
});
// Instances of company operation events belong to the concept of company operation events.
SET REL <CompanyEvent>-[std.belongTo]-><CompanyOperationTaxonomy>;
```

According to the previously defined concept taxonomy for company operations, “CompanyOperationTaxonomy”, we can associate the newly defined company event type, “CompanyEvent” with it, using the “std.belongTo” keyword. This association allows us to define instances of company events that belong to specific instances of the company operation concept within the company operation event classification concept.

3. The leadTo Predicate for Concept Inference

The “leadTo” predicate, unlike “belongTo”, represents a causal relation at the event concept level. Utilizing this rule allows for the automatic inference of related relations within event types. However, both “leadTo” and “belongTo” use the same syntax. We can further describe how SPG-Schema uses “leadTo” for semantic inference by redefining the event types of executive escape and stock price drop.

```
// Define that the operation event of one company can lead to the operation event of another company
SET REL <CompanyOperationTaxonomy>-[std.leadTo]-><CompanyOperationTaxonomy>;
// If an event instance "a" belongs to the concept of executive escape event, and there is an instance of executive escape
event concept leading to an instance of stock price fall event
<a:EscapeEvent>-[std.belongTo]-><EscapeEvent:CompanyOperationTaxonomy>;
<EscapeEvent:CompanyOperationTaxonomy>-[std.leadTo]-><FallInStock:CompanyOperationTaxonomy>
---------------------------------------------------------------------
// Automatically generate an instance "b" of FallInStock event, associated with the stock price fall event of company A
<b:FallInStock>-[std.belongTo]-><FallInStock:CompanyOperationTaxonomy>;
```

The relation defined by “std.leadTo” represents an association at the modeling level but applies to event concept instances at the instance layer. In the example mentioned above, the instance "ExecutiveEscapeEvent leads to FallInStock" represents a causal relation. When an event ‘A’ of an executive escape occurs and is associated with the executive escape event concept in the domain, a new event instance, “FallInStock of Company A”, can be directly generated.

4.4 Semantic Enhancement through Rule Definitions

To form a more comprehensive semantic framework, in addition to the basic predicates and constraints, it is also necessary to introduce rule definitions for supplementation. However, rule definitions are mostly implemented through programming rather than syntax definition. Therefore, in this section, we will introduce the syntax for rule definitions and provide relevant examples for reference.

4.4.1 The Self-defined Relation/Property Rules

The Self-defined relation/property rules are common requirements in business scenarios, where a specific relation is generated only when certain conditions are met. To accommodate rule definitions, we

can further expand the EdgeType syntax using the "RULE" keyword. Since the rule definitions often apply to the specific instances, we can use the format (instance:Class) to represent the relation triplets. However, creating a relation without a rule block guided by "RULE" will not have any additional effects.

```
// To define the hold share rate on a single chain link (recursively defined)
CREATE EDGE TYPE
(s:Person)-[p:hold_share_rate]->(o:LegalPerson)
RULE {
  STRUCTURE {
    (s)-[p1:hold_share_rate]->(c:LegalPerson),
    (c)-[p2:hold_share_rate]->(o)
  },
  CONSTRAINT {
    real_rate("The actual holding proportion") = p1.real_hold_share_rate*p2.real_hold_share_rate
    p.real_hold_share_rate = real_rate
  }
};
```

The rules can be divided into structural rules and constraint rules. These two kinds of rules are distinguished within rule blocks, guided by the keywords "Structure" and "Constraint" respectively. The former is composed of the triple relations that form the structure of an instance graph, while the latter is a piece of rule code.

4.4.2 Contradictory Rules

```
// Define relation --- Certificate is validly owned by LegalPerson
CREATE EDGE TYPE (Cert)-[effect_owned_by]->(LegalPerson) AS <effect_owned_by>;
// when the certificate is valid, there exists a inverse relation - LegalPerson owns the certificate.
CREATE EDGE TYPE (o:LegalPerson)-[p2:has]->(s:Cert) CONDINVERSEROF <effect_owned_by>
RULE {
  STRUCTURE {
  },
  CONSTRAINT {
    s.is_effect == true
  }
} AS <has>;
```

The conditional inverse relation is introduced using the "COMDINVERSEOF" keyword combined with the "RULE" keyword. It is guided by curly brackets, which enclose several conditional rule statements. Only when the conditional rules are satisfied can the two relations be considered as inverse relations. When the conditional rule is empty, it defaults to being equivalent to the inverse relation. In the example above, a basic relation, "<effect_owned_by>", is defined. Through inverse relation, it is defined that the "legal person owns certificate" relation will only be established when the condition of the certificate still being valid is met, and it is annotated with the alias "<has>".

4.5 The Relationship between SPG-Schemas and PG-Schemas

The PG-Schemas [16] aims to address the shortcomings in property graph database management and the lack of schema support in existing systems. It enhances type definitions and improves data integrity constraints to provide more flexible type management, supporting a certain degree of type inheritance and reuse. By formally defining entities, relations, and properties through PG-Keys [17], they can be expressed formally on the property graph. By establishing a flexible and powerful framework for defining key constraints, it enhances the logical connections and consistency among different elements of the schema. However, the complexity of knowledge graph construction and usage remains high, and users still need to prepare a large amount of data work. Additionally, there are still some issues when directly representing SPG using PG-Schemas.

- Lack of business semantic of types: PG-Schemas allow users to dynamically combine node types using the operators "&" (and) and "|" (or), providing richer type expression capabilities. However, it does not effectively capture the business semantics of types, including the internal semantic structure within types and the semantic representation between types. The fuzzy hierarchy relations between class labels also makes it difficult to control the structure in practical business implementation.
- Lack of support for logical dependencies: PG-Keys provide a framework for defining key constraints as global constraints to enhance the data integrity of the property graphs. However, further exploration is needed for the complexity of validation and maintenance in specific query languages, as well as for the problems of implication and reasoning. SPG knowledge management aims to achieve organic integration between logical rules and factual knowledge, depicting logical dependencies between knowledge, and building a hierarchical derivation mechanism for knowledge to reduce ineffective duplicate construction and ensure logical consistency.
- Partial support for feature completeness: The PG-Schemas article mentions that in the current version, binary constraints (BC) and introspection (IS) features are not fully supported. The authors also found through comparative experiments that BC features are lacking in other works such as RDFS, SHACL, and ShEx.

Overall, PG-Schemas primarily focus on enhancing database management with features such as improved type definitions and strengthened key constraints, as described in Chapters 1, 2, and 3. On the other hand, SPG is designed to provide knowledge management capabilities in terms of logical dependencies, knowledge hierarchy, knowledge construction, and programmability. These are two different perspectives, where PG-Schemas enhance capabilities for databases, while SPG aims to lower the barrier of entry for users to use knowledge graph and reduce their involvement in knowledge construction. PG-Schemas can provide global consistency validation, which can be applied to the knowledge graph produced by SPG. Combining the official description of PG-Schemas, the relationship between SPG and PG-Schemas can be illustrated as shown in Figure 30.

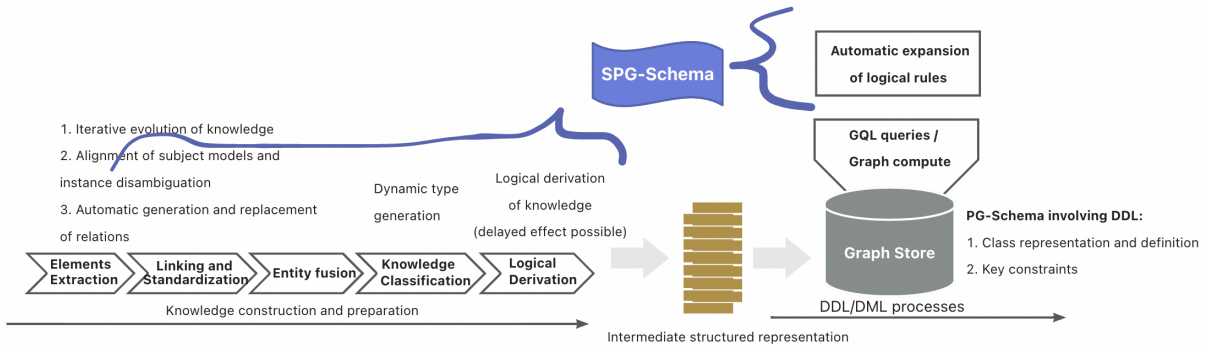

Figure 30: The Relationship between SPG and PG-Schemas

4.6 Summary of SPG-Schema

This chapter provides a detailed description of the overall architecture and the design of syntax and semantics for SPG-Schema. It also explains the extension details of the main model based on the general property graph model. The current version mainly focuses on three main categories: concepts, events, and standard types, which are derived from business requirements in SPG DC. The design and formation of logical syntax and related predicates for SPG Reasoning, which enhances semantic reasoning capabilities, are also described for the expanded classification model. In future versions, there will be a further rigorous proof of the semantic completeness of the current syntax, in order to form a user-friendly and complete syntax and semantic system. More detailed introductions to predicate logic semantics and syntax will be published in the form of serialized articles on the SPG technical community and public platform.

Chapter 5 SPG-Engine Layer

This chapter focuses on the implementation of the actual execution process of SPG syntax, which we refer to as the SPG-Engine layer. The SPG-Engine layer is a module that converts the inference and computation of SPG to be executed in an actual LPG system. The underlying dependencies of SPG typically include basic capabilities such as graph storage, graph querying, and graph computation, which are usually provided by the graph service provider of LPG. This chapter describes the overall architecture of the SPG-Engine layer, dividing it into functional modules such as graph model definition, graph data import, graph querying, and computation, and provides ways to integrate with the underlying LPG processing system.

5.1 The Architecture of SPG-Engine

The SPG-Engine is a module that converts the inference and computation of SPG to be executed in an actual LPG system. The underlying dependencies of SPG typically include basic capabilities such as graph storage, graph querying, and graph computation, which are provided by the graph service provider of LPG. In order to meet the requirements of knowledge graph inference and service capabilities based on SPG, we divide the requirements for engine capabilities into basic capabilities and advanced capabilities.

GQL [19] is the ISO international standard for the property graph query language, which is scheduled to be released in 2024. It defines the specification for querying based on property graphs and is compatible with both weakly-typed labels and strongly-typed types. The SPG solution does not impose any restrictions on whether the underlying graph service uses labels or types. As long as it can implement the interface of the SPG-Engine LPG Adapter, it can be integrated with the SPG engine. The SPG-Engine LPG Adapter provides a way for the third-party property graph systems to connect to the SPG system. It can be implemented using the GQL language or self-defined functions/procedures, and can also be implemented using a single HTAP [20] graph database system or a combination of an OLTP graph database system and an OLAP graph computing system.

In conjunction with the architecture diagram shown in Figure 24, the detailed functions of each module in the SPG Engine are shown in Figure 31.

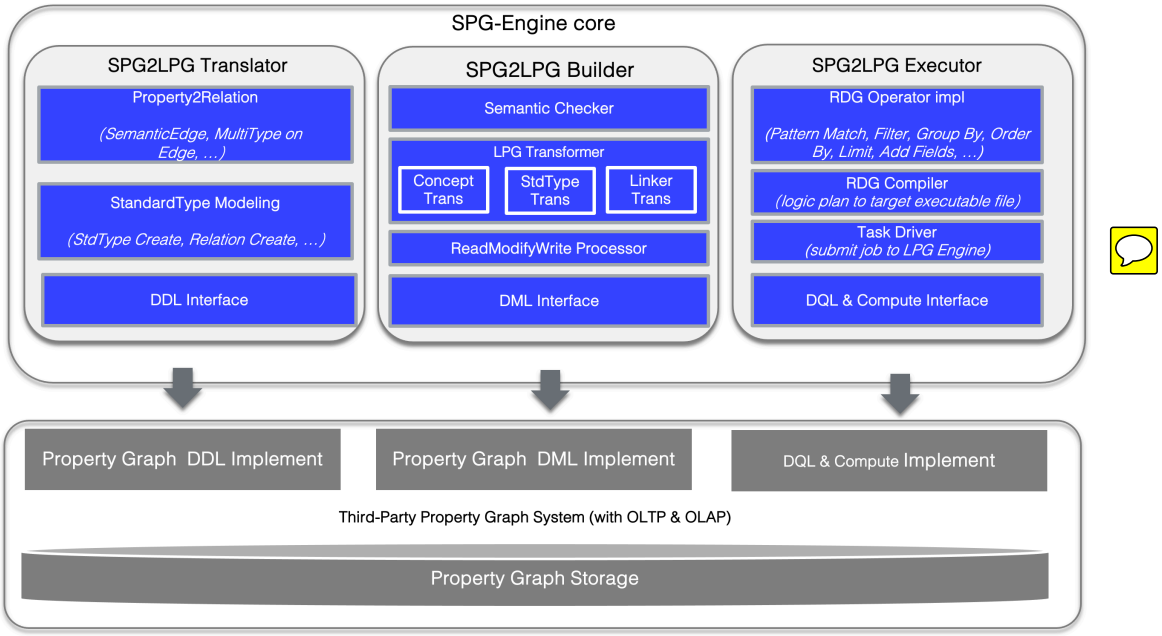

Figure 31: Overall Architecture of SPG-Engine

The SPG-Engine Core layer is a functional module that implements the conversion between SPG and LPG, running as a dependency package within the SPG-Controller process. The third-party property graph system serves as an independent service process, responsible for actual SPG data storage, querying, and computation tasks. It interfaces with the SPG2LPG Translator, SPG2LPG Builder, and SPG2LPG Executor in the SPG-Controller through DDL interfaces, DML interfaces, query interfaces, and computation interfaces. The third-party property graph system needs to meet the basic requirements specified in the SPG-Engine specification and implement the integration interface of the SPG-Engine LPG Adapter. For advanced requirements, they can be described using configuration files, and if not possible to implement, empty interface implementations should be provided.

As a module supporting core functionality, the performance and elastic deployment capability of SPG-Engine are crucial. A high-performance third-party property graph system can not only handle large amounts of data but also ensure system stability and responsiveness. The elastic deployment capability makes the system more flexible in adapting to various application scenarios and changing requirements, improving user satisfaction and business adaptability. We can evaluate performance and elastic deployment capability through methods such as load testing, benchmark testing, and simulation of real-world scenarios. In the current version of SPG 1.0, our focus is on functionality integration and implementation to ensure the complete realization of core functions. In future versions, we will enhance and strengthen the description and implementation of performance and elastic deployment capabilities and conduct more rigorous evaluations to meet user expectations.

5.2 SPG2LPG Translator

The SPG-Schema chapter describes the relationship between SPG-Schema and LPG-Schema, as shown in Figure 29 in Chapter 4. SPG adds semantic predicates, type extensions, logical rules, etc., on top of LPG.

In this chapter, the Semantic Layer's representation of the schema needs to be transformed to the corresponding schema format in the LPG engine. The SPG Meta Model can be transformed back and forth with the Meta Model, following the schema model hierarchy. The SPG2LPG Translator is primarily responsible for converting the SPG schema to the LPG schema format. One of the biggest differences between the SPG Schema and LPG Schema is the properties types. The translation framework needs to convert the semantic property types in SPG to text/numeric data types in LPG and generate the corresponding relations. Additionally, semantic constraints, such as inheritance, dynamic types, and sub-properties, also need to be translated. Furthermore, the built-in standard property types in the SPG Schema need to be converted into separate entity types with constraints and generate the corresponding relations. The SPG2LPG Translator, based on the mapping relationship from the SPG Meta Model to the Meta Model, can be divided into three layers:

- Propert(21)elation layer: converts properties into edges in the property graph, including standard properties, concepts, and events.
- StandardType Modeling layer: transforms standard properties, concepts, and events into corresponding vertex models.
- DDL interface layer: maps the conversion content from layers 1 and 2 to the DDL interface, requiring underlying property graph support for this capability.

Table 7 shows the type/relation mappings required to complete the translation from SPG Meta Model to LPG Meta Model.

Table 7: Translation Mapping from SPG Meta Model to LPG Meta Model

SPG Meta Model	Function	LPG Meta Model
Class	Entity Type	Node
Concept	Concept	Node
NormalizedProp	Standard Properties	Node
Event	Event	Node
leadTo	Causal Predicates between Concepts	Edge
hypernym	Hierarchical Predicates between Concepts	Edge
object	Object of an Event	Edge
subject	Subject of an Event	Edge
subClassOf	Hierarchy Relation of Entity	Edge
RelationShip	Relation Type	Edge
subRelOf	Hierarchical Relationship between Relations	Edge
MC	Multivariate Constraints of Relations	Edge

In SPG Schema, entity types/concept types support inheritance, and relations support reverse edges. These definitions need to be translated. Examples are provided in Figure 33 and Figure 34.

1. **Semantic Translation of Entity Types**

For entity types defined using the subClassOf (inheritance) predicate, the properties of the parent class need to be read first and then merged with the properties of the subclass to form the property set of the subclass. It is important to note that the property names of the parent class and subclass should not overlap, as this is a constraint of subClassOf. Additionally, all top-level parent classes of entity types inherit from the root type "Thing". The "Thing" type includes three basic properties: primary key ID, entity name, and description.

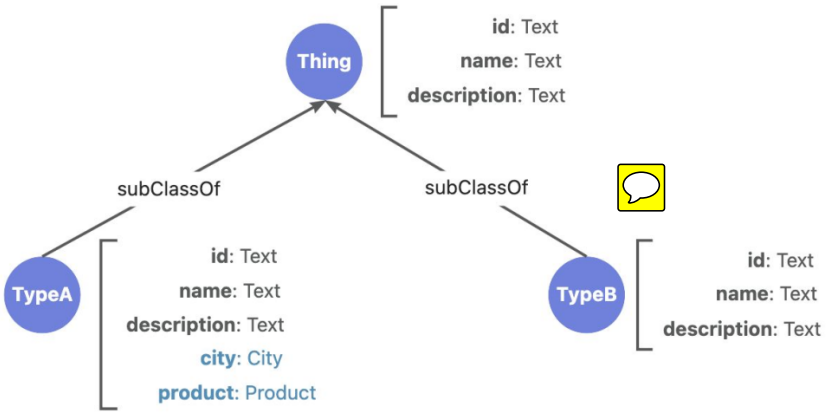

Figure 32: Illustration of the subClassOf Semantic

2. Conversion of Property Types to Entity Types/Concept Types

When the property in the SPG Schema is of type entity or concept, the following conversion actions need to be taken:

- Translate the property type to a text type.
- Add a text type property named "rawOf + propertyName" to store the original value of the property.
- Add a relation from the current entity type to the target entity type or concept type, with the relation name matching the property name.
- If there are sub-properties on the property, these need to be synchronized and created as properties of the relation.

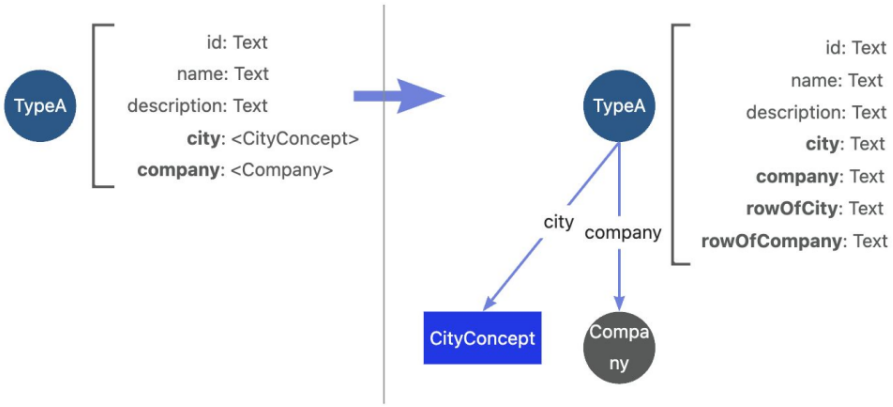

Figure 33: Lossless Redundant Adaptation Process from SPG Semantic Properties to LPG

5.3 SPG2LPG Builder

This module primarily addresses the handling of one or more entity/relation additions/deletions that may occur during the conversion of SPG-formatted data to LPG-formatted data. Data changes can include importing entities, deleting entities, importing relations, deleting relations, importing concepts, and deleting concepts. The submodules for transforming the SPG Meta Model into the LPG Model are shown in Figure 34, and they consist of three parts:

- Semantic Checker: This layer is responsible for semantic checks, ensuring that the input content complies with the constraints defined in SPG.
- LPG Transformer: This layer maps the SPG data to the actual property graph storage model. Detailed transformations will be described below.
- ReadModifyWriter Processor: This layer ensures consistency in read and write operations.

The conversion and adaptation process from SPG to LPG involves entity conversion, relation conversion, concept conversion, etc. For entity conversion, the following DML operations are required: UpsertNode/UpsertVertex for adding or updating nodes, DeleteNode/DeleteVertex for deleting nodes, AddEdge for adding relations, DeleteEdge for deleting relations, GetEdge for querying relations, etc. Firstly, the process checks if the entity property values comply with the corresponding type definitions. Then, queries the relations corresponding to the property names of the current entity and deletes the retrieved relations.

- When the property type is an entity type, the original value is written to the raw property. If the strategy is ID equality, a new relation is directly generated from the current entity to the entity with the ID indicated by the property value. If the strategy is operator, the operator logic is executed, and then a new relation is generated from the current entity to the entity with the result ID indicated by the linking operator.
- When the property type is a concept type, the original value is written to the raw property, and a new relation is generated from the current entity to the concept type specified by the property, with the ID of the property value.
- When the property type is a standard type, the original value is written to the raw property. A new standard type entity with the ID equal to the property value is created, and a new relation is generated from the current entity to the entity generated in the previous step.

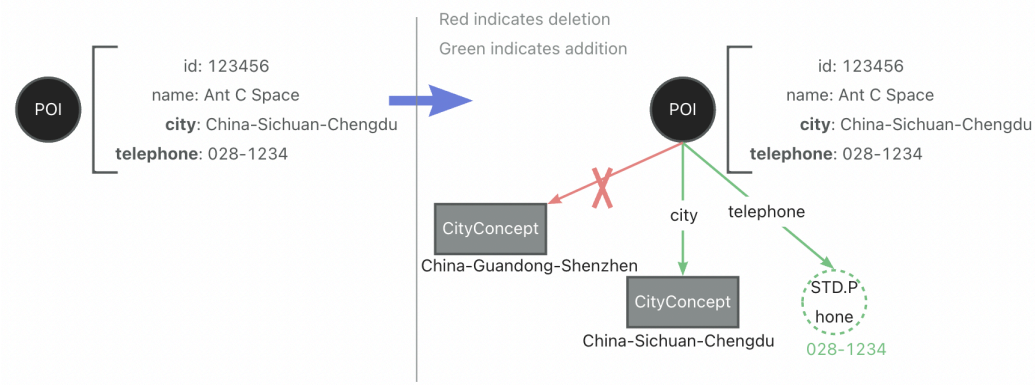

Figure 34: Subgraph Transaction Update Process of SPG Entity Instance converts to LPG

For relation conversion, the following LPG DML operations are required: UpsertNode/UpsertVertex for adding or updating nodes, AddEdge for adding relations, DeleteEdge for deleting relations, GetNode for querying nodes, GetEdge for querying relations, etc. The conversion logic is shown in Figure 37. The process checks if the relation changes comply with the semantic constraints. After adding or deleting a relation, the values on the equivalent properties of the relation need to be updated synchronously.

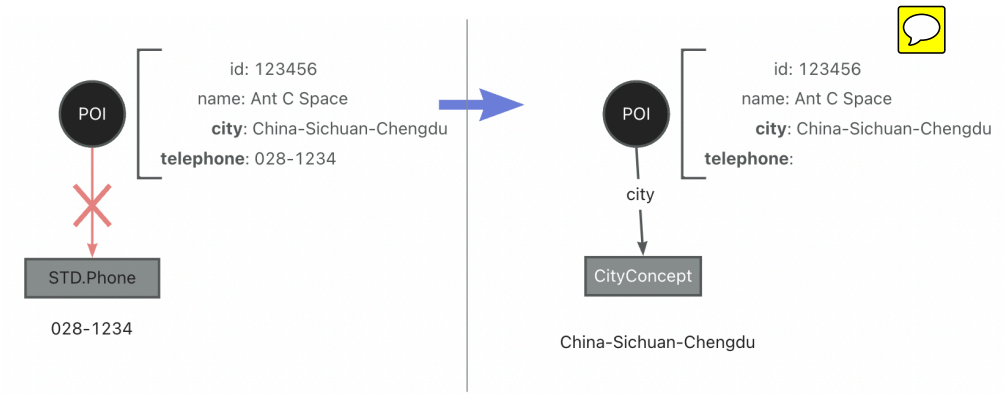

Figure 35: SPG semantic constraint check when updating entity instances

5.4 SPG2LPG Executor

The SPG2LPG Executor is mainly responsible for executing the execution plan composed of RDG operators (Resilient Distributed Graph, RDG) issued by the SPG-Reasoner. The design idea of the RDG model is derived from RDD [21] in Spark. Similar to the RDD approach, RDD simplifies the expression complexity of original MapReduce data operations by abstracted operators such as Map, Filter, and ReduceByKey. The data operation problems also exist in the knowledge graph, so the RDG model is abstracted to transform the required graph operations into operator operations in order to express complex computing processes. The execution plan is organized in a tree structure and the operators on the tree are executed in post-order traversal. To achieve the above goals, the entire SPG2LPG Executor is divided into three parts, each with the following functions:

- RDG Operator Impl: This layer implements the RDG operators based on the underlying LPG engine. It includes the functionality defined by each operator, such as Pattern Match, Filter, etc.

- RDG Compiler: The RDG compiler converts the execution plan issued by SPG-Reasoner into executable binary files that can be executed by the underlying LPG engine.
- Task Driver: This module submits the binary files generated by the RDG Compiler to the LPG Engine for execution. It needs to interface with the specific engine.

1. Execution Plan Generation

The execution plan expresses the process of data processing. Taking the determination of whether a user is a multi-device user as an example, the KGDSL rules are expressed as follows.

```
Define (s:Person)-[p:belongTo]->(o:UserClass/ManyDeviceUser) {
   Structure {
      (s)-[t:has]->(u:Device)
   }
   Constraint {
      has_device_num("Number of devices owned") = group(s).count(u.id)
      R1("owned more than 100 devices"): has_device_num > 100
      R2("Age greater than 18 years old"): s.age > 18
   }
}
```

After being transformed by the SPG-Reasoner, the following operator tree is formed.

```
└──DDL(ddlOp=Set(AddPredicate(PredicateElement(belongTo,p,(s:Person),EntityElement(ManyDeviceUser,UserClass))
   └──Filter(rule=LogicRule(R2,"Age greater than 18 years old",BinaryOpExpr(name=BGreaterThan)))
      └──Filter(rule=LogicRule(R1,"owned more than 100 devices",BinaryOpExpr(name=BGreaterThan)))
         └──GroupByAndAgg(group=Set(NodeVar(s,null))
            └──PatternMatch(pattern=PartialGraphPattern(s,Map(s -> (s:Person), u -> (u:Device)),Map(s -> Set((s)->[t:has]-
```

The operator tree starts with the PatternMatch node and ends with the DDL node. Each node in the tree represents an RDG operator.

2. RDG Operators

In the explanation of the execution plan in section (1), we discussed the order of execution for KGDSL operators. This section primarily introduces the definition of operators in RDG. The table below provides a list of operators.

Table 8: RDG Operator List

ID	Operator	Description
1	patternMatch	Given a subgraph pattern and a starting instance, obtain the subgraph instance (RDG) starting from that instance, returning null if not satisfied.
2	expandInto	Select an entity instance as the starting instance from the already obtained subgraph instances (RDG).
3	filter	Given a condition expression, check if the currently obtained subgraph instance satisfies it, discard the subgraph instance if not.
4	groupByAndAgg	Perform aggregate computation operations on the existing subgraph instances, such as calculating the number of neighbors for a specific point.
5	orderByAndLimit	Sort and truncate operations.
6	limit	Truncate operations without sorting.
7	shuffleAndFilter	Split and aggregate subgraph data based on different perspectives of the entity types.
8	addFields	Store temporary variables computed in the Rule.
9	dropFields	Remove unused temporary variables.
10	join	Merge different subgraph instances (RDG) together.
11	linkedExpand	Invoke a third-party service for link prediction.
12	ddl	Implement the definition of entities and relations.
13	select	Output specified subgraph results.
14	cache	Cache the current RDG for internal computation using the operator.

3. Generating Executable Code

RDG operators represent atomic operations. To convert the RDG operator tree into executable code for the underlying engine, it is necessary to combine it with the execution plan tree generated by the Execution Plan Generator. This process is illustrated in Figure 36.

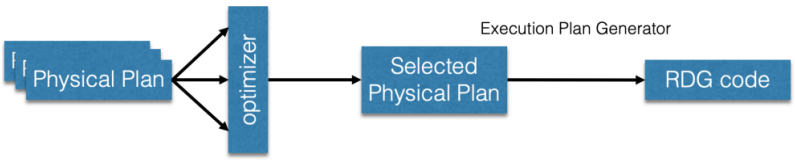

Figure 36: Executable Code Generation Process Flow

The Compiler generates executable code from the Physical Plan and RDG operators. The pseudocode for an Operator is as follows:

```
 abstract class PhysicalOperator[T <: RDG[T]: TypeTag]
extends AbstractTreeNode[PhysicalOperator[T]] {
 /**
  * The context during physical planner executing
  * @return
  */
 implicit def context: PhysicalPlannerContext[T] = children.head.context
 /**
  * The output of the current operator
  * @return
  */
 def RDG: T = children.head.RDG
```

```
  /**
   * The meta of the output of the current output
   * @return
   */
  def meta: List[Var]
}
```

Using the RDG operators as nodes to form a tree-like structure, execute in postorder traversal, for example, the PatternMatch operator.

```
final case class PatternMatch[T <: RDG[T]: TypeTag](
    in: PhysicalOperator[T],
    pattern: Pattern,
    meta: List[Var])
  extends PhysicalOperator[T] {
  override def rdg: T = in.rdg.patternMatch(pattern)
}
```

The input is the RDG of the child nodes. After invoking patternMatch, a new RDG data is returned. Using the example from section 1, the following code (using Neo4j client as an example) would be generated:

```
(new Neo4jRDG(driver)).patternMatch(pattern)
                      .GroupByAndAgg(s, COUNT)
                      .filter(expr("owned more than 100 devices"))
                      .filter("Age greater than 18 years old")
                      .ddl(new AddPredicate("s", "o", "belongTo"))
```

4. Task Execution

Based on different types of LPG engine interfaces, there are two different scenarios:

- Scenario 1: LPG provides query languages such as Cypher [22].
- Scenario 2: LPG provides computational programming frameworks like Spark RDD, allowing users to implement calculations by embedding self-defined operators.

For Scenario 1, the generated executable file is DQL, which communicates with the LPG query engine to perform queries and modifications on the target.

For Scenario 2, the generated code from section 4.3 can be packaged as a plugin and submitted to the LPG engine. This process is illustrated in Figure 37.

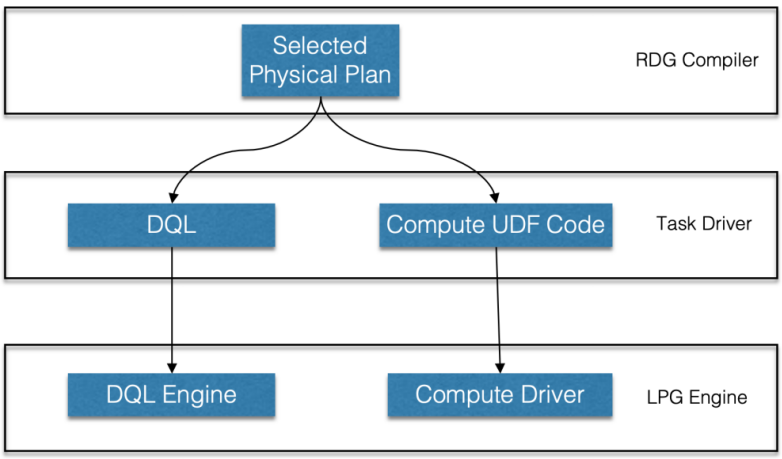

Figure 37: Task Driver Process Diagram

5.5 Basic Requirements for SPG-Engine on the Property Graph Systems

The third-party property graph systems are responsible for storing, querying, and performing calculations on SPG data. These systems use property graphs as the data model, representing and storing data using nodes, edges, and properties. They provide query and computation capabilities based on graph structure semantics. When classifying nodes and edges in the graph, the system can use weakly-typed labels or strongly-typed types, collectively referred to as Labelled Property Graph (LPG) in this context.

The third-party property graph system is an independent service process that should support distributed deployment. It should have independent cluster installation, deployment, management, monitoring, and operation methods, preferably with a web-based UI interface. The graph system interacts with SPG-Controller process through a set of adapter interfaces, which is the SPG-Engine LPG Adapter.

The third-party property graph system needs to include both data storage and querying, as well as analysis and computation on the graph data. Generally, there are two implementation approaches: using a single underlying system with HTAP capabilities, or using different TP and AP systems together to meet the requirements. Regardless of the implementation approach, the system should fulfill the basic requirements for SPG integration with third-party property graph systems, and ideally, fulfill the advanced requirements as well. The implementation should conform to the interface specifications described in SPG-Engine Core.

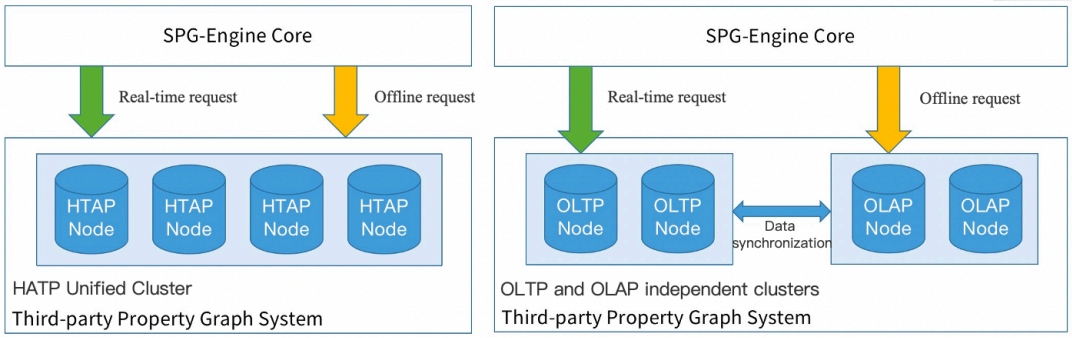

Figure 38: Architecture Diagram for a Single HTAP System and Different TP/AP Systems Integration

The basic requirements capabilities are summarized in Table 9.

Table 9: Basic Functional Requirements for Third-Party Property Graph Systems

Capability	Subcapability	Detailed Description
Graph Model	Supports Property Graph Model	Provides storage and querying of vertices, edges, and properties. Vertices and edges have types or labels.
	Supports Edge Properties	Supports directed and undirected edges. Allows multiple edges of the same type between two vertices.
	Supports Property Types	Supports basic property types: integer, float, string, etc. Supports complex property types: BigDecimal, DateTime, GeographicPoint, etc. Supports container property types: list, set, map, property group, etc.
	Supports Graph Isolation Requirements	Supports storage, querying, and computation of multiple graphs within the same graph service.
Graph Data Import	Data Import Methods	Supports batch import of full or incremental data. Supports streaming import.
	Data Source Support	Supports importing data from existing relational databases using JDBC connections with custom SELECT statements. Supports common file data sources like CSV and JSON formats. Supports popular big data storage file formats like Parquet files on HDFS.
	Mapping Methods	Supports mapping multiple entities and relations from a single source table.
	Import Constraints	Batch import can be performed without stopping the graph engine service.
Property Indexing	Supports indexing for common data types	Indexes can be created on integer, float, string, timestamp, etc.
	Supports composite indexes	Supports composite indexes on multiple property fields.
	Supports fuzzy indexing	Supports fuzzy indexing for full-text search on string properties.
Transaction	Supports ACID properties of transactions.	Provides at least Read Committed isolation level, and allows application-level explicit locking to achieve Serializable isolation level.
Graph Query Language	Supports DQL	Supports standardized query language like GQL or provides DQL query capabilities.
Deployment	Supports flexible deployment modes.	For smaller data volumes, supports master-slave deployment. For larger data volumes, supports distributed sharding deployment.

5.6 Advanced Requirements for SPG-Engine on the Property Graph Systems

5.6.1 System Capabilities

- Support for triggers: Triggers are a special type of stored procedure that sets up an automated event response mechanism in the database. When the specific database operations occur (such as insertions, updates, or deletions), the triggers automatically execute the predefined code or GQL / Cypher statements.
- Support for user-defined functions/procedures/algorithms: User-defined functions / procedures / algorithms provide a mechanism to extend graph queries, allowing users to customize the functionality of the database and access the internal API directly.
- Support for storage and querying of time-series data: Determine the validity of nodes or edges based on a timestamp field, as well as determine the validity of property values based on the timestamps.
- Capability for multi-level graphs: Mapping multiple logical graphs to one or more physical graphs based on business logic. Support mapping from entity to entity, from property to entity, from property to property, and allow querying directly on logical graphs using GQL/Cypher.
- Mapping and conversion of entity/relation types across graphs: Support querying across multiple graphs (including physical and logical graphs).

5.6.2 Semantic Functionality

- Conversion of specific properties to edges: Automatically insert and update edges through triggers and specify the storage method (in-memory cache or physical disk).

Table 10: Capability for Conversion of Properties to Edges

Requirement	SPG-Engine Capability	Description
Standard Property	Property convert to Relation	When creating specific properties, they are converted to relations
Conceptual Property	Property convert to Multi-hop Relations	When converting specific properties, they are converted to paths of multi-hop relations

```
// Entity type with properties support
create (n:Device {id:0, name:"devid****001"})

// When creating a standard property, the Trigger will execute the statement and convert it into an relation
match (m:wifi {id:"TP_LIN*****0011"}), (n: Device {name:"A"}) create (n)-[:use_wifi]->(m)

// When creating a conceptual property, the Trigger will execute the statement and convert it into multiple hop relations
match (m1:Country {name:"China"})<--(m2: Province {name:"SiChuan"})<--(m3:City {name:"ChengDu"}), (n:Person {name:"Zhang San"}) create (n)-[:livi_in]->(m3)
```

- Support for knowledge hierarchy: Automatically insert and update based on self-defined logical rules through triggers.

Table 11: Capability for SPG Type Classification

Requirement	SPG-Engine Capability	Description
Entity Type	Type constraint	When creating a entity and relation, a entity that meets the constraint rules will be created with an relation connected to the labeled entity.
Relation Type	Type constraint	When creating a entity and relation, a entity that meets the constraint rules will be created with an relation connected to the labeled entity.
Property Type	Property constraint	When modifying properties, the properties of entities that meet the constraint rules will be modified accordingly.

```
// Define entity type and add type constraint, check if the constraints are met when creating Person type, create belongTo relation if they are met
// Create Person entity
create (n:Person {id:"2088****0001"})
// Trigger server-side constraint checking, create corresponding relation if constraints are met
match (n:Person {id:" 2088****0001"})-[:has]->(D:Device)-[:has_wifi]->(W:WIFI)<-[:has_wifi]-(D2:Device)<-[:has]-(s)-[p:belongTo]->(o:Fraudster) create (n:Person {id:" 2088****0001"})-[p:belongTo]->(o:Fraudster)

// Define relation type and add type constraint, check if the constraints are met when creating Device entity, create same_wifi relation if they are met
create (n:Device {id:" devid****001"})
// Trigger server-side constraint checking, create corresponding relation if constraints are met
match (n:Device {id:" devid****001"})--[:has_wifi]->(W:WIFI)<-[:has_wifi]-(o:Device) create (n)-[:same_wifi]->(o)


// Define property type and add property constraint, when modify property of App, modify corresponding property of instances that meet the constraint rules
match (n:App {id:"appid*****0012"}) set n.mark = "black"
// Trigger server-side constraint checking, modify corresponding property of instances if constraints are met
match (n:App {id:" appid*****0012"})<-[:release]-(m:Company) set m.mark = "black"
```

- Support for built-in predicates: The built-in predicates require multiple advanced capabilities of LPG, as shown in Table 12.

Table 12: Advanced Capability Requirements for LPG

Requirement Content	Advanced LPG Capability	Introduction
Transitivite	Condition Matching for Paths	Input points, edges, and path conditions to perform path matching.
SameAs	Graph Mapping (see 1.4)	Match subgraphs based on conditions, perform data computation and aggregation, and use the results to create new subgraphs in the current graph.
ConditionInverseOf	Predicate Matching	The predicate expresses a condition of inverse relation.
InverseOf	Predicate Matching	The predicate expresses an inverse relation.
SubPropertyOf	Support for Property Group Value Types	Property groups can have multiple sub-properties, but sub-properties cannot be of Property Group type.
NormalizedProperty	See "Self-defined Logical Rules"	
EquivalentProperty	Property Mapping (see 1.4)	Equality constraints on multiple properties on a entity.
MutexOf	Primary Key Constraint	Unique primary key constraint for the entity of a certain type.

```
// Transitivity, requires support for path condition matching
// User profile aggregation, gathering people with the same taste as Zhang San
match (n:Person {name:"Zhang San"})-[:Hobbies|isA*3]->(m:Taste)<-[:Hobbies |isA*3]-(m2:Person) return m as Taste, collect(m2) as Crowd
// Ultimate controller exploration
match (n:Company {id:" 4201151234****ABC" ) <-[r:Contorl *1..5]-(m:Person)
return m as Controller, sum(reduce(total = 1, h IN r | total * h.holdingRatio / 100.0 ) ) as holdingRatio order by  holdingRatio

// Contradictory conditions, predicate matching
// Find fully-owned parent companies of Company A
match (A:Company {id:" 4201151234****ABC"})-[: SubsidiaryCompany {holding_rate:1}]->(B) return B
// Find fully-owned subsidiaries of Company B
match (B:Company {id:" 4201151234****ABC"})<-[: SubsidiaryCompany {holding_rate:1}]-(A) return A
```

5.7 Summary

This chapter introduced the architecture and implementation of the SPG-Engine layer. The SPG-Engine layer consists of the SPG-Engine Core and the third-party property graph systems. The SPG-Engine Core provides modules for graph modeling, data import, querying, and computation under the SPG semantics, and invokes the interfaces provided by the third-party property graph systems for execution. The third-party property graph systems are provided by LPG graph service vendors, and can be a unified cluster with HTAP capabilities or a combination of separate clusters for OLTP graph databases and OLAP graph processing. This chapter also described the integration with underlying LPG processing systems and the required functionalities of the third-party property graph systems, enabling SPG to run on various commonly used property graph systems. It also provided optimization directions for the property graph vendors to enhance the performance of SPG. More details and implementation examples of RDG will be continuously published in articles on the SPG official account.

Chapter 6 SPG-Controller Layer

The SPG-Controller is the control layer of the SPG framework, responsible for analyzing, invoking, and managing the execution of services and tasks. As the core hub of the SPG framework, it is closely associated with other modules to collectively complete the entire task flow from user input to result. The SPG-Controller receives requests from the SPG-LLM or SPG-Programming, performs parsing and compilation, and generates task planning. It distributes and invokes tasks, selects corresponding capabilities to complete the specific execution process, including choosing the corresponding runtime from registered and deployed SPG-Engine, SPG-Index, or external capabilities. Finally, it returns the task execution results to the caller. This chapter provides an overview of the architecture and workflow of the SPG-Controller, and provides a general description of task compilation planning, task distribution and invoking, as well as knowledge querying, construction, reasoning, and searching services. The detailed description will be gradually unfolded in conjunction with relevant subsystems in the 2.0 and 3.0 white papers.

6.1 The Architecture and Workflow of SPG-Controller

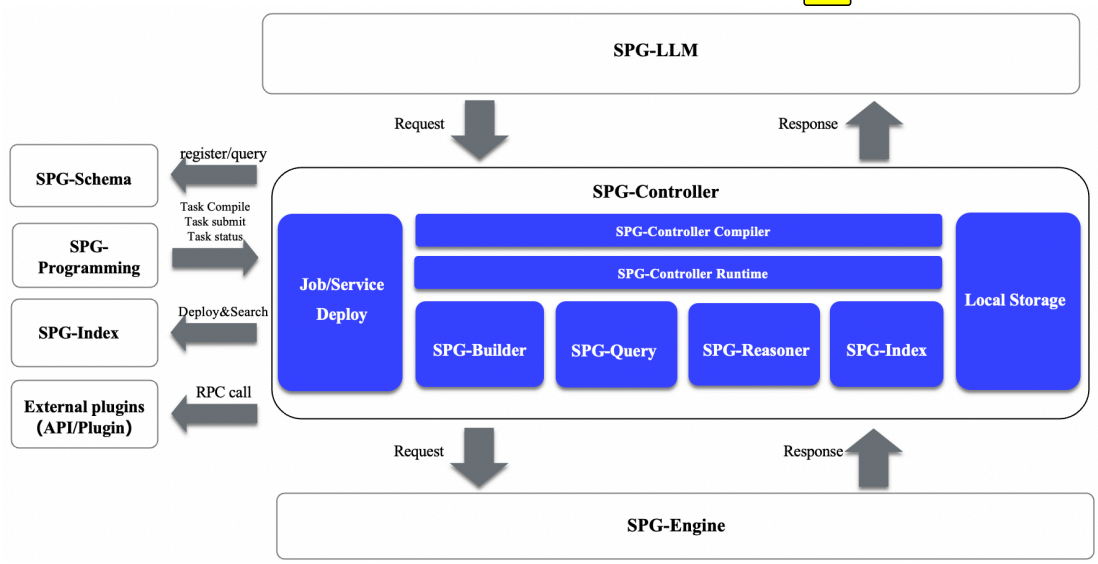

Figure 39: The architecture and workflow of SPG-Controller

The SPG-Controller is the core control hub in the SPG framework, with the following main responsibilities: (1) Parsing, compilation, and task planning: It performs parsing and compilation based on the input from the upper layer, mapping it to the capabilities, commands, and relevant configurations and parameters of each subsystem. Then, based on the results of parsing and compilation, it plans the tasks to achieve the arrangement of task execution mode, process, and cycle. (2) Task distribution and invoking: Based on the results of the task planning, it executes and invokes services and tasks in the corresponding runtime. (3) SPG-Builder: It provides a runtime engine for knowledge construction to transform data into knowledge in SPG. (4) SPG-Query: It provides a runtime engine for knowledge querying, implementing interfaces related to querying and graph-based data mining and analysis in SPG. (5) SPG-Reasoner: It provides a runtime engine for knowledge inference calculation, implementing various inference capabilities

in SPG, such as logical rule inference and neural network inference. (6) SPG-Index: It provides a runtime engine for SPG indexing and searching, implementing the capabilities such as vector search and full-text search in SPG. (7) Job/Service Deploy: It provides the registration and deployment of services or tasks in the corresponding runtime for each SPG Controller, enabling their capabilities to be effectively called.

In the overall system architecture, firstly, the SPG-Controller serves as the runtime engine for the SPG-Schema and provides CRUD Python/Java interfaces. Secondly, it provides task compilation and distribution capabilities for the SPG-Programming. The task compilation includes syntax checking, algorithm/task validity checking and reusing, execution planning, and bytecode generation. Additionally, it accepts pluggable sub-engine modules to be added to the registration information module. Finally, it accepts task requests as input, compiles and distributes tasks, dispatches tasks to sub-engines for computation, receives the results, and assembles them to return to the user.

6.2 Parsing, Compilation, and Task Planning

The input parsing module receives input from SPG-LLM or SPG-Programming and parses it to understand the instructions of the user. This helps identify the task type, execution logic, and process that need to be performed. The input consists of two main parts: task type and task body. The task type determines the type of the task to be executed, while the task body includes the interfaces and their parameters to be invoked. Here are the main task types and their definitions: (1) Schema tasks: Corresponds to basic CRUD interfaces for the knowledge graph schema. The task body includes the interface to be called and its parameters. The interface is defined and implemented in the SPG-Schema module. (2) Query tasks: Corresponds to basic knowledge graph queries, utilizing the graph query capabilities in SPG-Engine. The task body consists of a query language like GQL. (3) Reasoning tasks: Corresponds to reasoning tasks in the knowledge graph, such as rule-based reasoning using KGDSL or neural network-based reasoning tasks. (4) Search tasks: Corresponds to search tasks, such as vector indexing and full-text indexing. This includes tasks such as index creation, tokenization indexing/vector data writing, and text search/vector search. The task body consists of query tasks with conditions. (5) Construction tasks: Corresponds to SPG knowledge graph construction tasks, involving mapping structured data to knowledge, extracting unstructured and multimodal knowledge, and knowledge fusion.

Task planning is based on the results of parsing and compilation to arrange the execution mode, process, and cycle of the tasks.

6.3 Task Distribution and Invocation

Based on the results of task planning, the tasks are distributed and executed in the corresponding runtime, including the registered runtimes for various services and tasks, as well as the pipeline of the task execution. During the task execution process, SPG provides a microservice invocation framework and task scheduling engine to enable service invocation and task execution. Additionally, the SPG-Controller provides storage capabilities to store crucial information during task execution, including log data,

intermediate and final results, execution status, and scheduling data. Storing and tracing this information aids in monitoring and managing the task execution process, ensuring reliability and stability.

6.4 Knowledge Graph Construction

SPG-Builder facilitates the conversion from data into knowledge, including the extraction of knowledge from structured data, semi-structured data, and multi-modal unstructured data. The main capabilities include: (1) ER2SPG: This functionality involves converting data to SPG knowledge, where knowledge is obtained through mapping and transformation of data from a database or big data platform. Based on the conversion of original data to ER (Data to ER, D2R), the input is modified to comply with the SPG specification. (2) Semi-structured knowledge extraction: This functionality is used for extracting knowledge from semi-structured data, and obtain structured elements. This functionality will be added in the second phase. (3) Text knowledge extraction: This functionality will involve utilizing Large Language Models (LLMs) to extract knowledge from text data. This functionality will be added in the second phase. (4) Multi-modal knowledge extraction: This functionality is designed for extracting knowledge from multi-modal unstructured data, such as images, audio, video, and so on. This functionality will be added in the second phase.

With the support of these capabilities, SPG-Builder can extract valuable knowledge from various forms of data, fully harnessing the vast amount of data accumulated in the big data ecosystem. This knowledge is then stored in the SPG repository for future application, analysis, and inference purposes.

6.5 Knowledge Query

SPG-Query provides knowledge graph query and analysis services, including the basic CRUD operations on knowledge graph: graph searching, and graph analysis and mining. Specific query functions include: (2) Basic Query: Supports precise queries on entities, concepts, and properties. For example, querying for the account with the ID “2088****0001” in the risk mining knowledge graph. (2) Advanced Query: Supports fuzzy queries and full-text search on entities, concepts, and properties. For example, performing a fuzzy search on event names or company names in the enterprise causal knowledge graph. (3) Graph Traversal Query: Supports breadth-first and depth-first algorithms for graph traversal queries. For example, querying for accounts that have transaction records in the risk mining knowledge graph. (4) Pattern Matching Query: Supports subgraph queries that satisfy specified patterns. For example, querying for subgraphs in the risk mining knowledge graph that follow the pattern A-B-C-A (transfer of funds).

In addition, SPG-Query provides various graph analysis algorithms, including: (1) Community Detection Algorithms: such as LPA, WCC, SCC, Louvain, etc., which can be used to identify frequent transaction groups in the risk mining knowledge graph. (2) Authority Ranking Algorithms: such as PageRank, HITS, degree centrality, betweenness centrality, closeness centrality, etc., which can be used to calculate the weight of each node in the risk mining knowledge graph. (3) Other Algorithms: such as triangle counting, which can be used to calculate the frequency of a specific transaction subgraph in the risk mining knowledge

graph. These features assist users in gaining a deeper understanding of the knowledge in SPG, enabling them to analyze it more effectively and extract valuable insights.

6.6 Knowledge Graph Reasoning

SPG-Reasoner provides the capability to invoke knowledge graph reasoning, including commonly used knowledge graph reasoning methods. Specific reasoning methods includes: (1) Rule-based Reasoning Algorithms: Used for inferring risk propagation rules in enterprise causal knowledge graph and defining expert rules related to risky activities in the risk mining knowledge graph. (2) Graph Embedding Learning Algorithms: Includes graph neural networks, random walks, translation distance, etc., used for embedding learning and representation learning of the knowledge graph in SPG. (3) Prompt Learning Algorithms: Used for constructing learning algorithms for prompts in SPG-LLM. This functionality will be specifically implemented in future releases. 

To support these reasoning capabilities, SPG-Controller provides storage management for registering information on reasoning rules, reasoning algorithms, and configuring algorithm parameters. This aids in monitoring and managing the reasoning process.

6.7 Full-Text Search and Vector Search

SPG-Index provides search services, including conventional search methods such as vector search and full-text search. SPG-Controller interacts with the external SPG-Index plugin, allowing users to easily search and query data. Specific functionalities include: (1) Index Creation and Management: Supports the creation and management of indexes. (2) Index Data Writing and Updating: Supports writing data to the index and performing updates. (3) Vector-based Search: Supports vector-based search, enabling data search and query based on similarity.

6.8 Deployment of the Services and Tasks

"Job/Service Deploy" handles the registration and deployment of services or tasks corresponding to the core modules. Specific functionalities provided include: (1) Registration: Supports the registration of services and tasks, allowing them to be discovered and invoked in SPG-Controller. (2) Management: Supports the management of microservices or tasks, including the oline/offline operations and the configuration management. (3) Execution: Supports the execution of services or tasks in the runtime environment. (4) Monitoring: Supports monitoring the running status, including resource monitoring, to promptly detect and resolve service failures.

In the implementation, SPG-Controller stores the registered and deployed services and tasks, and schedules them during the task execution. To achieve the task scheduling, a task scheduling engine can be used to manage jobs scheduled precisely to the hour, minute, and second, and the concurrency level can be set by dynamically configuring shard parameters. To enable microservice invocation, a microservice

governance framework can be used to manage and invoke microservices, and a microservice gateway can be employed to expose services externally.

6.9 Summary 

This chapter provides a summary description of the overall architecture and workflow of the SPG-Controller. In terms of specific tasks, the current version primarily focuses on constructing structured data into a knowledge graph, basic knowledge querying and analysis, knowledge reasoning, and knowledge searching. In future versions, these tasks will be further refined and expanded, such as constructing knowledge graphs from unstructured and multimodal data, representation learning for reasoning, and composite indexing. Additionally, other types of tasks will be integrated, such as integrating external plugin systems and instruction systems. Furthermore, future versions will enhance the management functionalities of SPG-Controller, including unified authentication management and exception handling.