Big Graph Processing Systems

Part II: Property Graphs

▶ **Chapter 2:** Schemas and Constraints

Christopher Spinrath

CNRS – LIRIS – Lyon 1 Université

DISS Master 2025

This presentation is an adaption of slides from Angela Bonifati

Schemas?

Logical Schema of a Database

- A schema describes the structure or/and is a blueprint for database instances in a formal language
- **this lecture**

Schemas?

Logical Schema of a Database

- A schema describes the structure or/and is a blueprint for database instances in a formal language
- **this lecture**

Physical Schema of a Database

- A physical schema describes how a database is represented and stored (data structures, in memory, on disk, in files)
- Bonifati, G. H. L. Fletcher, et al., *Querying Graphs*, 2018, Chapter 6

Why do we need Schemas for Property Graphs

1. Data exploration
   - letting the user making sense of the data without delving into the intricacies of the graph instances

Why do we need Schemas for Property Graphs

1. Data exploration
   - letting the user making sense of the data without delving into the intricacies of the graph instances
2. Data visualization
   - visualizing a smaller graph (i.e. the schema) instead of visualizing the entire graph (i.e. the instance)

Why do we need Schemas for Property Graphs

1. Data exploration
   - letting the user making sense of the data without delving into the intricacies of the graph instances
2. Data visualization
   - visualizing a smaller graph (i.e. the schema) instead of visualizing the entire graph (i.e. the instance)
3. Query formulation
   - formulating a query by selecting the schema concepts instead of navigating labels/set of properties in the instance
   - accessing the instance only for formulating the predicates (constants).

Why do we need Schemas for Property Graphs

1. Data exploration
   - letting the user making sense of the data without delving into the intricacies of the graph instances
2. Data visualization
   - visualizing a smaller graph (i.e. the schema) instead of visualizing the entire graph (i.e. the instance)
3. Query formulation
   - formulating a query by selecting the schema concepts instead of navigating labels/set of properties in the instance
   - accessing the instance only for formulating the predicates (constants).
4. Query optimization
   - a query that retrieves nodes connected by a path might be optimized by using the schema elements of the path (labels of the vertices of the paths and labels of the edges are in the schema)

Why do we need Schemas for Property Graphs

1. Data exploration
   - letting the user making sense of the data without delving into the intricacies of the graph instances
2. Data visualization
   - visualizing a smaller graph (i.e. the schema) instead of visualizing the entire graph (i.e. the instance)
3. Query formulation
   - formulating a query by selecting the schema concepts instead of navigating labels/set of properties in the instance
   - accessing the instance only for formulating the predicates (constants).
4. Query optimization
   - a query that retrieves nodes connected by a path might be optimized by using the schema elements of the path (labels of the vertices of the paths and labels of the edges are in the schema)
5. Graph transformations
   - mappings between different graph databases are guided by the schemas (source and target schemas)

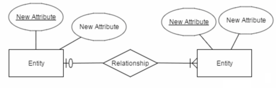
Why do we need Schemas for Property Graphs

1. Data exploration
   - letting the user making sense of the data without delving into the intricacies of the graph instances
2. Data visualization
   - visualizing a smaller graph (i.e. the schema) instead of visualizing the entire graph (i.e. the instance)
3. Query formulation
   - formulating a query by selecting the schema concepts instead of navigating labels/set of properties in the instance
   - accessing the instance only for formulating the predicates (constants).
4. Query optimization
   - a query that retrieves nodes connected by a path might be optimized by using the schema elements of the path (labels of the vertices of the paths and labels of the edges are in the schema)
5. Graph transformations
   - mappings between different graph databases are guided by the schemas (source and target schemas)
6. Data Integration, Data Quality
   - graph database sources to be integrated
7. Data Quality
   - monitor graph database for quality

Schemas for Graphs: A Fragmented Landscape

ER Models

- Chen ER
- Extended ER
- Enhanced ER
- ORM2
- UML Class Diagram



RDF Schemas

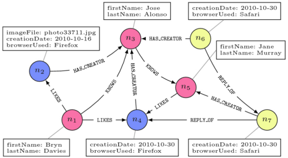
- RDFS
- OWL
- SHACL
- ShEx

Tree-shaped Schemas

- DTD/XML Schema
- JSON Schema
- RELAX NG

Graph Schemas

- GraphQL
- openCypher
- SQL/PGQ

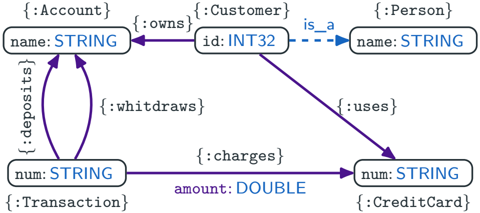


(Limited) Schemas in Graph DBs

- AgensGraph
- ArangoDB
- DataStax
- JanusGraph
- Nebula Graph/nGQL
- Neo4j
- Oracle/PGQL
- OrientDB/SQL
- Sparksee
- TigerGraph/GSQL
- TypeDB/TypeQL

The Design of Property Graph Schemas

Towards GQL-Compliant Property Graph Schemas



Support of Schemas in Contemporary Graph Databases

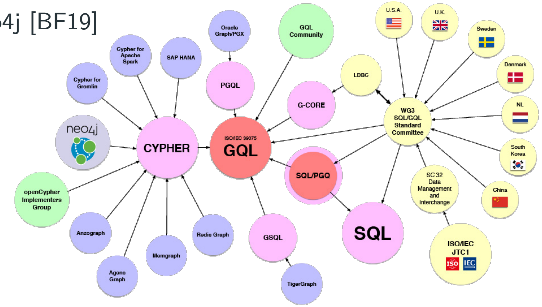
Limited Support of Schemas in Contemporary Graph Databases

- Graph databases are mainly schema-less
  - No a priori schema constraints
  - Thus, error-prone data integration and metadata management
- 11 graph databases are reviewed in the paper [PGS23]
  - AgensGraph, ArangoDB, DataStax, JanusGraph, Nebula Graph/nGQL, Neo4j, Oracle/PGQL, OrientDB/SQL, Sparksee, TigerGraph/GSQL, TypeDB/TypeQL
- Need of a consensus on design requirements of PG Schemas

[PGS23] Angles, Bonifati, Dumbrava, G. Fletcher, Green, et al., "PG-Schema: Schemas for Property Graphs", *Proc. ACM Manag. Data*, 2023

The Quest for Schemas in Graph Databases

- Design of Cypher-like property graph schemas in 2019
  - Activity carried out in collaboration with Neo4j [BF19]

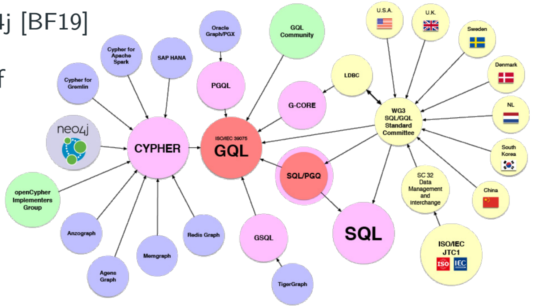


[BF19] Bonifati, Furniss, et al., "Schema Validation and Evolution for Graph Databases", *ER 2019*, 2019
[GQL-22] Deutsch et al., "Graph Pattern Matching in GQL and SQL/PGQ",. *Proceedings of SIGMOD*,. 2022
[PGS23] Angles, Bonifati, Dumbrava, G. Fletcher, Green, et al., "PG-Schema: Schemas for Property Graphs", *Proc. ACM Manag. Data*, 2023

[GQL-ISO] https://www.gqlstandards.org/

The Quest for Schemas in Graph Databases

- Design of Cypher-like property graph schemas in 2019
  - Activity carried out in collaboration with Neo4j [BF19]
- Schema support is limited in the first version of the GQL standard [GQL-22]
  - Ongoing ISO's Working Group for Database Languages [GQL-ISO]

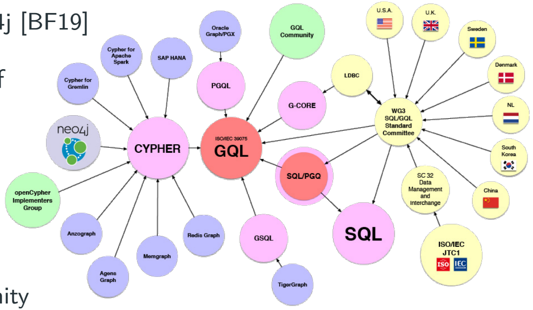


[BF19] Bonifati, Furniss, et al., "Schema Validation and Evolution for Graph Databases", *ER 2019*, 2019
[GQL-22] Deutsch et al., "Graph Pattern Matching in GQL and SQL/PGQ",. *Proceedings of SIGMOD*,. 2022
[PGS23] Angles, Bonifati, Dumbrava, G. Fletcher, Green, et al., "PG-Schema: Schemas for Property Graphs", *Proc. ACM Manag. Data*, 2023

[GQL-ISO] https://www.gqlstandards.org/

The Quest for Schemas in Graph Databases

- Design of Cypher-like property graph schemas in 2019
  - Activity carried out in collaboration with Neo4j [BF19]
- Schema support is limited in the first version of the GQL standard [GQL-22]
  - Ongoing ISO's Working Group for Database Languages [GQL-ISO]
- Focus on design of a standard schema language for property graphs in 2023
  - Activity carried out within the LDBC community (`https://ldbcouncil.org/gql-community/pgswg/`)
  - Proposal: PG-Schema [PGS23]



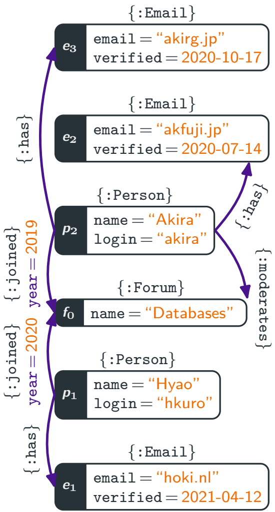
[BF19] Bonifati, Furniss, et al., "Schema Validation and Evolution for Graph Databases", *ER 2019*, 2019
[GQL-22] Deutsch et al., "Graph Pattern Matching in GQL and SQL/PGQ",. *Proceedings of SIGMOD*,. 2022
[PGS23] Angles, Bonifati, Dumbrava, G. Fletcher, Green, et al., "PG-Schema: Schemas for Property Graphs", *Proc. ACM Manag. Data*, 2023

[GQL-ISO] `https://www.gqlstandards.org/`

PG-Schema: An Example

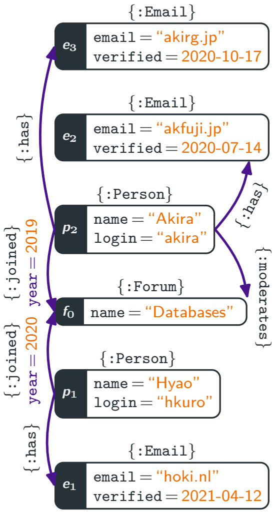
```
CREATE GRAPH TYPE messageBoard LOOSE {
  // node types
  (personType: Person {name STRING, login STRING}),
  (forumType: Forum {title STRING OPEN}),
  (emailType: Email {
    email STRING, OPTIONAL verified DATE}),

  // edge types
  (:personType)-[hasType: has]->(:emailType),
  (:personType)
    -[joinedType: joined {year: INT}]->(:forumType),
  (:personType)
    -[moderatedType: moderated]->(:forumType),
}
```



Representing Schemas as Property Graphs

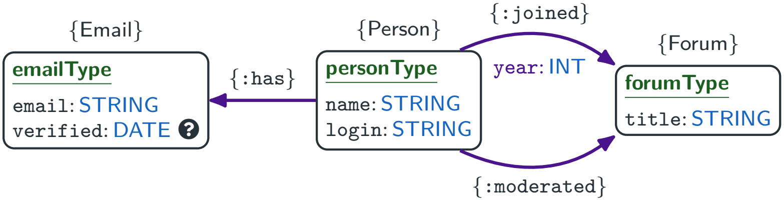
- Idea: Represent a schema as a (small!) property graph (inherited from openCypher schemas [BF19])
  - Schema nodes define node types
  - Schema relations define relations allowed between types
  - Properties on schema elements define sets of allowed properties

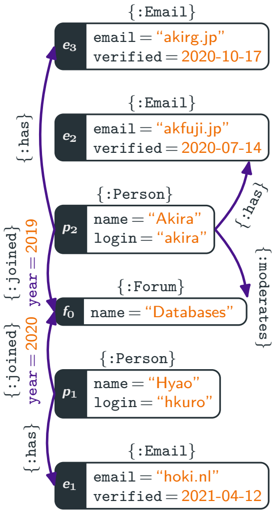


[BF19] Bonifati, Furniss, et al., "Schema Validation and Evolution for Graph Databases", *ER 2019*, 2019

Representing Schemas as Property Graphs

- Idea: Represent a schema as a (small!) property graph (inherited from openCypher schemas [BF19])
  - Schema nodes define node types
  - Schema relations define relations allowed between types
  - Properties on schema elements define sets of allowed properties

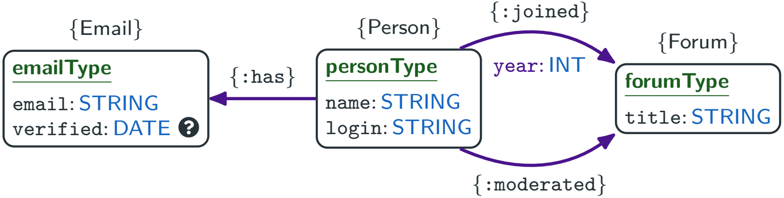


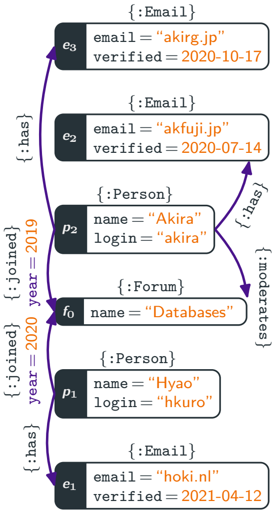


[BF19] Bonifati, Furniss, et al., "Schema Validation and Evolution for Graph Databases", *ER 2019*, 2019

Representing Schemas as Property Graphs

- Idea: Represent a schema as a (small!) property graph (inherited from openCypher schemas [BF19])
  - Schema nodes define node types
  - Schema relations define relations allowed between types
  - Properties on schema elements define sets of allowed properties
- Schema validation via graph homomorphisms!





[BF19] Bonifati, Furniss, et al., "Schema Validation and Evolution for Graph Databases", *ER 2019*, 2019

Three Possible Scenarios

1. Schema-First (or Prescriptive)
   - schema provided during setup

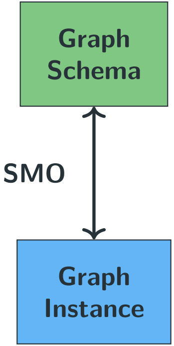
2. Flexible Schema (or Descriptive)
   - users can use schema as description of what is in the data

3. Partial Schema (Prescriptive and Descriptive co-exist)
   - both prescriptive and descriptive are allowed on the same property graph

Prescriptive and Descriptive Schemas

Prescriptive updates

- Deletion of schema elements can be propagated to data
- We can clone schema nodes (split concepts)



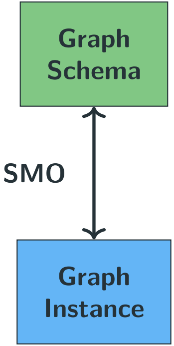
Prescriptive and Descriptive Schemas

Prescriptive updates

- Deletion of schema elements can be propagated to data
- We can clone schema nodes (split concepts)

Descriptive updates

- Creation of data elements can be propagated to schema
- We can merge nodes of different types



Prescriptive and Descriptive Schemas

Prescriptive updates

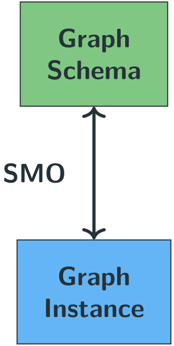
- Deletion of schema elements can be propagated to data
- We can clone schema nodes (split concepts)

Descriptive updates

- Creation of data elements can be propagated to schema
- We can merge nodes of different types

Schema manipulation operations (SMO)

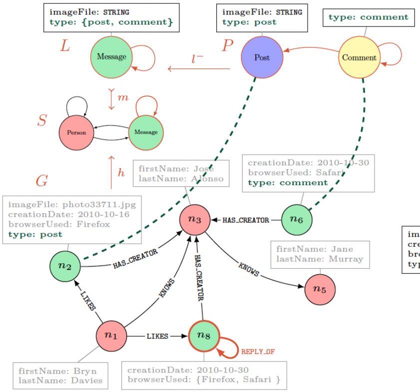
- Create: add a new node/edge types
- Drop: remove some node/edge types
- Split: partition a node/edge type into more fine-grained types
- Join: merge node/edge types into more coarse-grained types



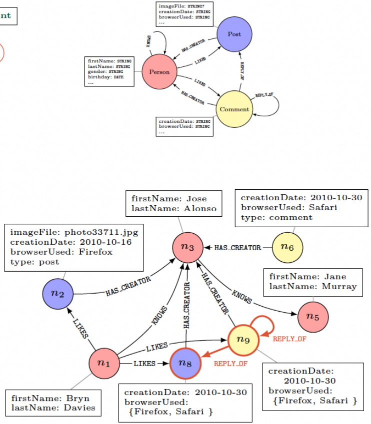
Rewriting Rules of the Form LPR

- Rewriting R given a rule P and a matching of its left-hand side L ($L \leftarrow P \rightarrow R$)

Schema update



Result

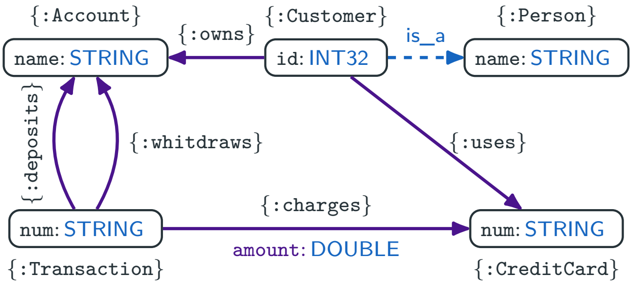


Use Case Study

Step 1 Andrea connects to a data catalogue and loads the schema information

Step 2 To detect fraud, Andrea wants to identify suspicious customers

- By leveraging the schema, Andrea selects the involved schema types, e.g. Customer, Account etc.



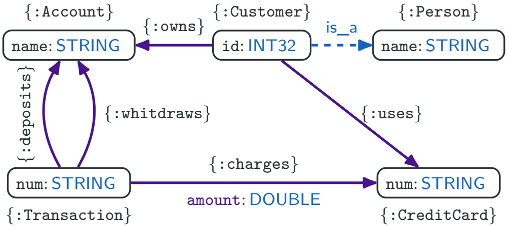
Use Case Study

Step 3 The graph explorer automatically construct a search form for customers, including name and id (inherited from Person type in the schema)

Step 4 Andrea needs to understand connections between customers

Patterns of interest in Step 4

```
//Pattern 1
(x:Customer)-[:uses]->(:CreditCard)
               <-[:uses]-(y:Customer)
//Pattern 2
(x:Customer)-[:uses]->(:CreditCard)
  <-[:charges]-(t:Transaction)
  -[:charges]->(:CreditCard)
  <-[:uses]-(y:Customer)
```



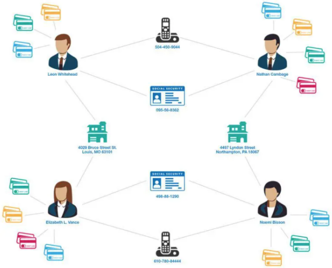
Use Case Study

Step 5 Andrea selects the first connection pattern from Step 4 and the schema-based application executes an efficient query to retrieve the results

Step 6 The graph explorer visualizes the results of the query and let the user classifies the fraudulent cases

Chooses the first pattern in Step 5

```
//Pattern 1
(x:Customer)-[:uses]->(:CreditCard)
               <-[:uses]-(y:Customer)
//Pattern 2
(x:Customer)-[:uses]->(:CreditCard)
  <-[:charges]-(t:Transaction)
  -[:charges]->(:CreditCard)
  <-[:uses]-(y:Customer)
```



Schema Requirements: Types

R1 Node Types

Schemas must allow for defining types for nodes that specify their labels and properties

Schema Requirements: Types

R1 Node Types

Schemas must allow for defining types for nodes that specify their labels and properties

```
(personType: Person { name STRING , OPTIONAL birthday DATE})
(personType: Person OPEN { name STRING , OPTIONAL birthday DATE})
(personType: Person { name STRING , OPTIONAL birthday DATE, OPEN})
```

Schema Requirements: Types

R1 Node Types

Schemas must allow for defining types for nodes that specify their labels and properties

```
(personType: Person { name STRING , OPTIONAL birthday DATE})
(personType: Person OPEN { name STRING , OPTIONAL birthday DATE})
(personType: Person { name STRING , OPTIONAL birthday DATE, OPEN})
```

R2 Edge Types

Schemas must allow for defining types for edges that specify their labels and properties as well as the types of their endpoints

Schema Requirements: Types

R1 Node Types

Schemas must allow for defining types for nodes that specify their labels and properties

```
(personType: Person { name STRING , OPTIONAL birthday DATE})
(personType: Person OPEN { name STRING , OPTIONAL birthday DATE})
(personType: Person { name STRING , OPTIONAL birthday DATE, OPEN})
```

R2 Edge Types

Schemas must allow for defining types for edges that specify their labels and properties
as well as the types of their endpoints

```
(:personType)-[friendType: Knows & Likes {since DATE}]->(:personType)
(:personType | customerType)
  -[friendType: Knows & Likes {since DATE}]->(:personType | customerType)
```

Schema Requirements: Types Cont'd

R3 Content Types

- Schemas must support a practical repertoire of data types in content types.
- Support for GQL content types and any other sets of data types
  - `STRING`
  - `DATE`
  - `INT`
  - Lists
  - etc.

Schema Requirements: Constraints

R4 Key Constraints

Schemas must allow for specifying key constraints

- in particular, “primary keys”

Schema Requirements: Constraints

R4 Key Constraints

Schemas must allow for specifying key constraints

- in particular, "primary keys"

R5 Participation Constraints

Schemas must allow for specifying participation constraints (as in ER diagrams)

- e.g. nodes of a given type participate in a relationship of a given type

Schema Requirements: Constraints

R4 Key Constraints

Schemas must allow for specifying key constraints

- in particular, "primary keys"

R5 Participation Constraints

Schemas must allow for specifying participation constraints (as in ER diagrams)

- e.g. nodes of a given type participate in a relationship of a given type

R6 Type Hierarchies

Schemas must allow for specifying type hierarchies

```
(salariedType: Salaried { salary INT })
(employeeType: personType & salariedType)
```

Schema Requirements: Flexibility

R7 Evolving Data

Schemas must allow for defining node, edge, and content types with a finely-grained degree of flexibility in the face of evolving data

Schema Requirements: Flexibility

R7 Evolving Data

Schemas must allow for defining node, edge, and content types with a finely-grained degree of flexibility in the face of evolving data

```
CREATE GRAPH TYPE fraudGraphType LOOSE {
  (personType: Person {name STRING, OPTIONAL birthday DATE}),
  (customerType: Person & Customer {name STRING, OPTIONAL since DATE}),
  (suspiciousType: Suspicious OPEN {reason STRING, OPEN}),
  (:personType | customerType )
      -[friendType : Knows & Likes]->(:personType | customerType)
}
```

Schema Requirements: Flexibility

R7 Evolving Data

Schemas must allow for defining node, edge, and content types with a finely-grained degree of flexibility in the face of evolving data

```
CREATE GRAPH TYPE fraudGraphType LOOSE {
  (personType: Person {name STRING, OPTIONAL birthday DATE}),
  (customerType: Person & Customer {name STRING, OPTIONAL since DATE}),
  (suspiciousType: Suspicious OPEN {reason STRING, OPEN}),
  (:personType | customerType )
      -[friendType : Knows & Likes]->(:personType | customerType)
}
```

R8 Compositionality

Schemas must provide a fine-grained mechanism for compositions of compatible types of nodes and edges

Schema Requirements: Usability

R9 Schema Generation

There should be an intuitive easy-to-derive constraint-free schema for each property graph that can serve as a descriptive schema in case one is not specified.

Schema Requirements: Usability

R9 Schema Generation

There should be an intuitive easy-to-derive constraint-free schema for each property graph that can serve as a descriptive schema in case one is not specified.

R10 Syntax and Semantics

The schema language must have an intuitive declarative syntax and a well-defined semantics

Schema Requirements: Usability

R9 Schema Generation

There should be an intuitive easy-to-derive constraint-free schema for each property graph that can serve as a descriptive schema in case one is not specified.

R10 Syntax and Semantics

The schema language must have an intuitive declarative syntax and a well-defined semantics

R11 Validation

Schemas must allow efficient validation and validation error reporting

Schema Requirements: Usability

R9 Schema Generation

There should be an intuitive easy-to-derive constraint-free schema for each property graph that can serve as a descriptive schema in case one is not specified.

R10 Syntax and Semantics

The schema language must have an intuitive declarative syntax and a well-defined semantics

R11 Validation

Schemas must allow efficient validation and validation error reporting

References

More details about syntax and formal semantics are available:

- Angles, Bonifati, Dumbrava, G. Fletcher, Green, et al., "PG-Schema: Schemas for Property Graphs", *Proc. ACM Manag. Data*, 2023
- Deutsch et al., "Graph Pattern Matching in GQL and SQL/PGQ",. *Proceedings of SIGMOD*,. 2022
- `https://www.gqlstandards.org/`

Beyond PG-Schemas: Extensibility

Range Constraints

```
(bookType: Book {
  title STRING (100),
  genre ENUM("Prose", "Poetry", "Dramatic"),
  isbn STRING ^(?=(?:\ D*\d) {10}(?:(?:\ D*\d){3})?)[−]+ })
```

Beyond PG-Schemas: Extensibility

Range Constraints

```
(bookType: Book {
  title STRING (100),
  genre ENUM("Prose", "Poetry", "Dramatic"),
  isbn STRING ^(?=(?:\ D*\d) {10}(?:(?:\ D*\d){3})?)[−]+ })
```

Complex Datatypes

```
STRING ARRAY {1,2}
```

Beyond PG-Schemas: Extensibility

Range Constraints

```
(bookType: Book {
  title STRING (100),
  genre ENUM("Prose", "Poetry", "Dramatic"),
  isbn STRING ^(?=(?:\ D*\d) {10}(?:(?:\ D*\d){3})?)[−]+ })
```

Complex Datatypes

```
STRING ARRAY {1,2}
```

Intersections and Unions for Content Types

```
(personType: Person
    ({ name STRING } | { givenName STRING , familyName STRING } )
    { height (INT | FLOAT) })
```

Beyond PG-Schemas: Extensibility

Range Constraints

```
(bookType: Book {
  title STRING (100),
  genre ENUM("Prose", "Poetry", "Dramatic"),
  isbn STRING ^(?=(?:\ D*\d) {10}(?:(?:\ D*\d){3})?)[−]+ })
```

Complex Datatypes

```
STRING ARRAY {1,2}
```

Intersections and Unions for Content Types

```
(personType: Person
    ({ name STRING } | { givenName STRING , familyName STRING } )
    { height (INT | FLOAT) })
```

Cardinality Constraints

```
FOR (d:Department) COUNT 2.. OF e
WITHIN (e: Employee )-[:worksIn ]->(d)
```

The Design of Property Graph Constraints

For Quality Control in Graph Databases

(Angles, Bonifati, Dumbrava, G. Fletcher, Hare, et al., “PG-Keys: Keys for Property Graphs”, *SIGMOD '21*, 2021)

Key Constraints for Property Graphs

Keys are ... key in data management

- For identifying, referencing and constraining objects
- They are core components of PG-Schemas

Key Constraints for Property Graphs

Keys are ... key in data management

- For identifying, referencing and constraining objects
- They are core components of PG-Schemas

Example (Person Nodes)

Node representing persons

- are uniquely identified by their login ID
- can be referenced using one of their email addresses (and it is mandatory that each person has at least one email),
- of which at most one can be the preferred email
- have zero or more aliases which are exclusive (i.e., no two people can share an alias)

Key Constraints for Property Graphs

Keys are ... key in data management

- For identifying, referencing and constraining objects
- They are core components of PG-Schemas

Example (Forum Nodes)

Node representing forums

- are uniquely identified by their name and the person who moderates the forum

```
(:Person)<-[:hasModerator]-(:Forum)
```

Key Constraints for Property Graphs

Keys are ... key in data management

- For identifying, referencing and constraining objects
- They are core components of PG-Schemas

Example (Forum Nodes)

Node representing forums

- are uniquely identified by their name and the person who moderates the forum

  ```
  (:Person)<-[:hasModerator]-(:Forum)
  ```

- This is not a property-based primary key
- Identity depends on properties, (other) nodes, and edges

Limited Support for Keys in Graph Databases

Limited Support for Keys in Graph Databases

- Landscape is diverse
  - Some systems offer property-based primary keys for nodes
  - Other systems support uniqueness
  - Other systems support mandatoriness

Limited Support for Keys in Graph Databases

Limited Support for Keys in Graph Databases

- Landscape is diverse
  - Some systems offer property-based primary keys for nodes
  - Other systems support uniqueness
  - Other systems support mandatoriness

- Yet we need to support all of these, and more, to satisfy current practical needs

Limited Support for Keys in Graph Databases

Limited Support for Keys in Graph Databases

- Landscape is diverse
  - Some systems offer property-based primary keys for nodes
  - Other systems support uniqueness
  - Other systems support mandatoriness
- Yet we need to support all of these, and more, to satisfy current practical needs
- There is already a significant drift between database vendors
  - Need to get on the same page
  - Need to bring the best of academic work to the attention of industry

Scope and Descriptor

Main Ingredients of a Key

To specify a key, we have to specify

1. a scope and
2. a descriptor

Scope and Descriptor

Main Ingredients of a Key

To specify a key, we have to specify

1. a scope and
2. a descriptor

Scope

A scope is the set of all possible targets of a constraint

Scope and Descriptor

Main Ingredients of a Key

To specify a key, we have to specify

1. a scope and
2. a descriptor

Scope

A scope is the set of all possible targets of a constraint

Example (Forum Nodes)

Node representing forums

- are uniquely identified by their name
- and the person who moderates the forum

```
(:Person)<-[:hasModerator]-(:Forum)
```

Scope and Descriptor

Main Ingredients of a Key

To specify a key, we have to specify

1. a scope and
2. a descriptor

Scope

A scope is the set of all possible targets of a constraint

Example (Forum Nodes)

Node representing forums

- are uniquely identified by their name
- and the person who moderates the forum

```
(:Person)<-[:hasModerator]-(:Forum)
```

- The scope is the set of all nodes representing forums

Scope and Descriptor

Main Ingredients of a Key

To specify a key, we have to specify

1. a scope and
2. a descriptor

Scope

A scope is the set of all possible targets of a constraint

Descriptor

Determines key values for each target in the scope

Example (Forum Nodes)

Node representing forums

- are uniquely identified by their name
- and the person who moderates the forum

```
(:Person)<-[:hasModerator]-(:Forum)
```

- The scope is the set of all nodes representing forums

Scope and Descriptor

Main Ingredients of a Key

To specify a key, we have to specify

1. a scope and
2. a descriptor

Scope

A scope is the set of all possible targets of a constraint

Descriptor

Determines key values for each target in the scope

Example (Forum Nodes)

Node representing forums

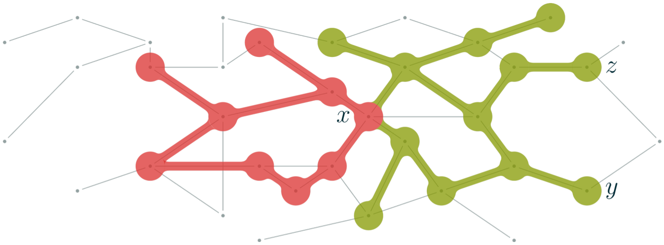
- are uniquely identified by their name
- and the person who moderates the forum

```
(:Person)<-[:hasModerator]-(:Forum)
```

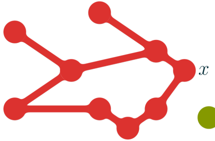
- The scope is the set of all nodes representing forums

- The descriptor assigns every node representing a forum a unique pair of
  - a name, and
  - a person (moderating it)

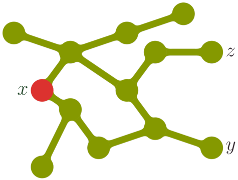
PG-Keys



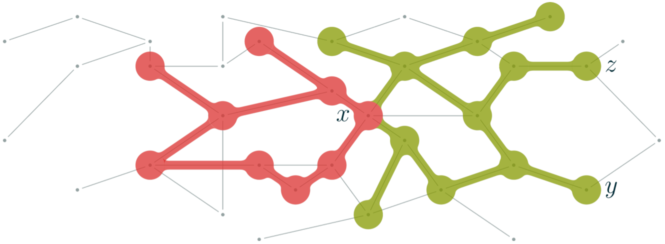

```
FOR x WITHIN
```



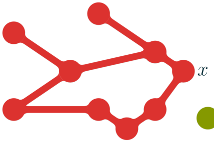

```
IDENTIFIER y, z WITHIN
```



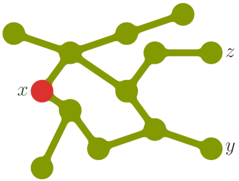
PG-Keys




```
FOR x WITHIN
```




```
IDENTIFIER y, z WITHIN
```

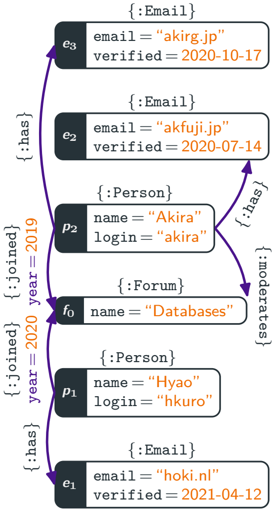


Design requirements

1. Flexible choice of key scope and descriptor of key values
2. Keys for nodes, edges, and properties
3. Identify, reference, and constrain objects
4. Easy to validate

Flexible Choice of Scope and Key Values

- Declaratively specify the scope and descriptor of the key
- In your favourite query language (a parameter of PG-Keys)
- Here we use a GQL-like syntax



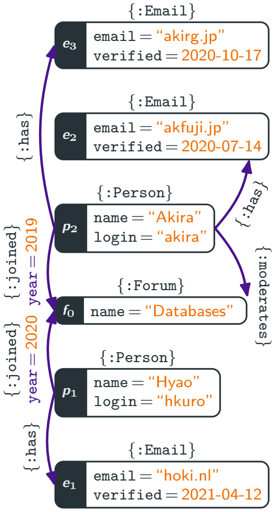
Flexible Choice of Scope and Key Values

- Declaratively specify the scope and descriptor of the key
- In your favourite query language (a parameter of PG-Keys)
- Here we use a GQL-like syntax

Example (Person Nodes)

"Each person is identified by their login"

```
FOR p WITHIN (p:Person) IDENTIFIER p.login
```



Flexible Choice of Scope and Key Values

Flexible Choice of Scope and Key Values

- Declaratively specify the scope and descriptor of the key
- In your favourite query language (a parameter of PG-Keys)
- Here we use a GQL-like syntax

Example (Person Nodes)

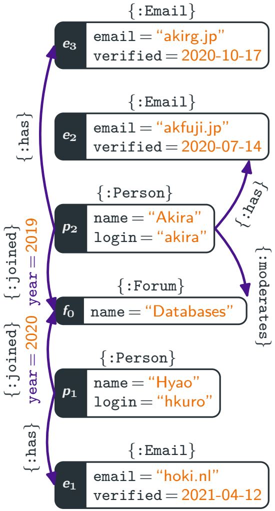
“Each person is identified by their login”

```
FOR p WITHIN (p:Person) IDENTIFIER p.login
```

Example (Forum Nodes)

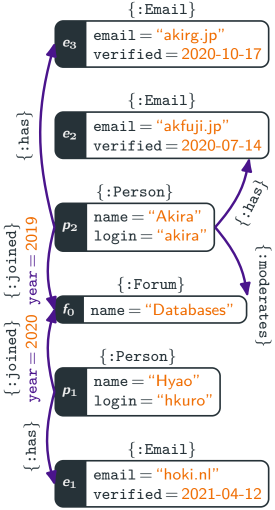
“Each forum with a member is identified by its name and moderator”

```
FOR f WITHIN (f:Forum)<-[:joined]-(:Person)
IDENTIFIER f.name, p WITHIN (f)<-[:moderates]-(p:Person)
```



Keys for Nodes, Edges, and Properties

- The scope query selects a set of nodes, edges, or property values



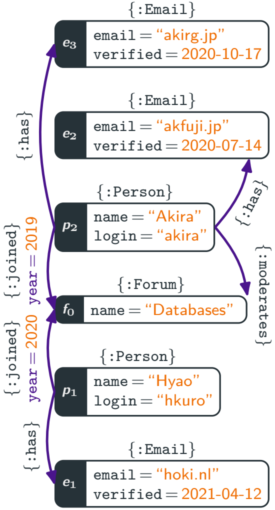
Keys for Nodes, Edges, and Properties

- The scope query selects a set of nodes, edges, or property values

Example (Person Nodes)

“Each node labelled :person is identified by the value of property login”

```
FOR p WITHIN (p:Person) IDENTIFIER p.login
```



Keys for Nodes, Edges, and Properties

- The scope query selects a set of nodes, edges, or property values

Example (Person Nodes)

"Each node labelled :person is identified by the value of property login"

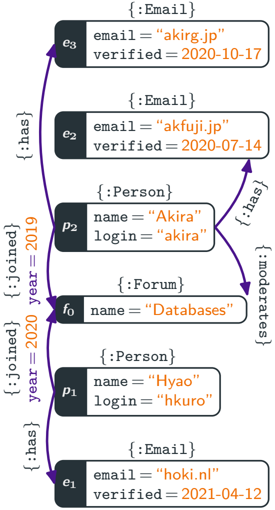
```
FOR p WITHIN (p:Person) IDENTIFIER p.login
```

Example (Joined Relationships)

"Each edge labelled :joined is identified by its endpoints"

- i.e., no other edge labelled :joined has the same endpoints,
- so a person cannot join the same forum twice

```
FOR e WITHIN (:Person)-[e:joined]->(:Forum)
IDENTIFIER p, f WITHIN (p:Person)-[e:joined]->(f:Forum)
```

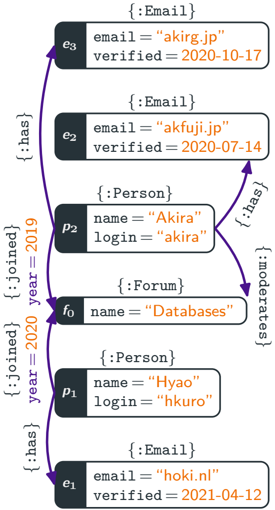


Identify, Reference, and Constrain Objects

- Unique identification can be expressed with the qualifier `IDENTIFIER`

Example (Forum Nodes)

```
FOR f WITHIN (f:Forum)<-[:joined]-(:Person)
IDENTIFIER f.name, p WITHIN (f)<-[:moderates]-(p:Person)
```



Identify, Reference, and Constrain Objects

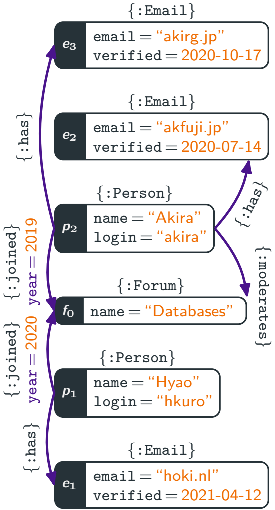
- Unique identification can be expressed with the qualifier IDENTIFIER

Example (Forum Nodes)

```
FOR f WITHIN (f:Forum)<-[:joined]-(:Person)
IDENTIFIER f.name, p WITHIN (f)<-[:moderates]-(p:Person)
```

IDENTIFIER is the combination of the qualifiers

- EXCLUSIVE – no two targets in the scope can have the same key value
- MANDATORY – each target in the scope has at least one key value
- SINGLETON – each target in the scope has at most one key value



Identify, Reference, and Constrain Objects

- Unique identification can be expressed with the qualifier `IDENTIFIER`

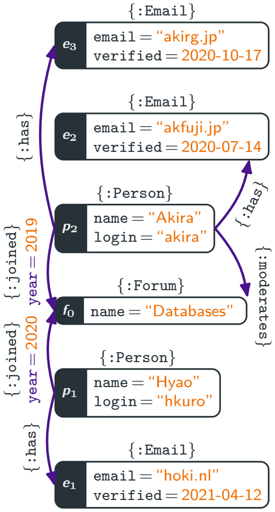
Example (Forum Nodes)

```
FOR f WITHIN (f:Forum)<-[:joined]-(:Person)
IDENTIFIER f.name, p WITHIN (f)<-[:moderates]-(p:Person)
```

`IDENTIFIER` is the combination of the qualifiers

- `EXCLUSIVE` – no two targets in the scope can have the same key value
- `MANDATORY` – each target in the scope has at least one key value
- `SINGLETON` – each target in the scope has at most one key value

In SQL, `EXCLUSIVE` is `UNIQUE`, `MANDATORY` is `NOT NULL`, and `SINGLETON` is always ensured by 1NF. For property graphs, all three are required.



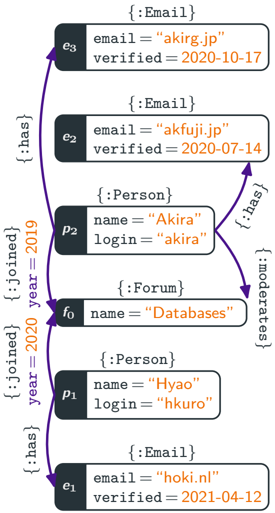
Easy to Validate

- To check that a PG-Key holds, we can run queries to find violations

Example

The key constraint

```
FOR p WITHIN (p:Person)
EXCLUSIVE MANDATORY e WITHIN (p)-[:has]-(e:Email)
```



Easy to Validate

Easy to Validate

- To check that a PG-Key holds, we can run queries to find violations

Example

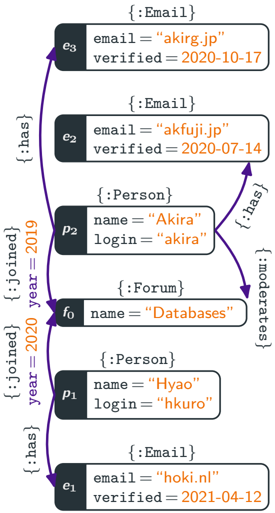
The key constraint

```
FOR p WITHIN (p:Person)
EXCLUSIVE MANDATORY e WITHIN (p)-[:has]-(e:Email)
```

holds if if both queries return no answers

```
MATCH (p1:Person)-[:has]->(:Email)<-[:has]-(p2:Person)
WHERE p1 <> p2 RETURN p1, p2

MATCH (p:Person)
WHERE NOT EXISTS (p1:Person)-[:has]->(:Email)
```



Easy to Validate

- To check that a PG-Key holds, we can run queries to find violations

Example

The key constraint

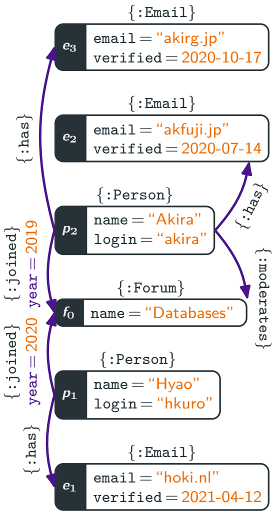
```
FOR p WITHIN (p:Person)
EXCLUSIVE MANDATORY e WITHIN (p)-[:has]-(e:Email)
```

holds if if both queries return no answers

```
MATCH (p1:Person)-[:has]->(:Email)<-[:has]-(p2:Person)
WHERE p1 <> p2 RETURN p1, p2

MATCH (p:Person)
WHERE NOT EXISTS (p1:Person)-[:has]->(:Email)
```

Incremental validation or batching will require additional mechanisms



References

Angles, Renzo, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savkovic, Michael Schmidt, Juan Sequeda, Slawek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Dusan Zivkovic (June 20, 2023). "PG-Schema: Schemas for Property Graphs". In: *Proc. ACM Manag. Data*, pp. 1–25. DOI: 10.1145/3589778.

Angles, Renzo, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W. Hare, Jan Hidders, Victor E. Lee, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Josh Perryman, Ognjen Savkovic, Michael Schmidt, Juan F. Sequeda, Slawek Staworko, and Dominik Tomaszuk (2021). "PG-Keys: Keys for Property Graphs". In: *SIGMOD '21*. Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. ACM, pp. 2423–2436. DOI: 10.1145/3448016.3457561.

Bonifati, Angela, George H. L. Fletcher, Hannes Voigt, and Nikolay Yakovets (2018). *Querying Graphs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers. DOI: 10.2200/S00873ED1V01Y201808DTM051. URL: https://doi.org/10.2200/S00873ED1V01Y201808DTM051.

Bonifati, Angela, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt (2019). "Schema Validation and Evolution for Graph Databases". In: *ER 2019*. Ed. by Alberto H. F. Laender, Barbara Pernici, Ee-Peng Lim, and José Palazzo M. de Oliveira. Vol. 11788. Lecture Notes in Computer Science. Springer, pp. 448–456. DOI: 10.1007/978-3-030-33223-5_37. URL: https://doi.org/10.1007/978-3-030-33223-5%5C_37.

Deutsch, Alin, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar Van Rest, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Fred Zemke (June 10, 2022). "Graph Pattern Matching in GQL and SQL/PGQ". In: *Proceedings of SIGMOD*. SIGMOD/PODS '22: International Conference on Management of Data. Philadelphia PA USA: ACM, pp. 2246–2258. DOI: 10.1145/3514221.3526057. URL: https://dl.acm.org/doi/10.1145/3514221.3526057 (visited on 03/19/2024).