The Quest for Schemas in Graph Databases

Angela Bonifati
Lyon 1 University & CNRS Liris (France)
Keynote@AMW 2023, Santiago (Chile)
About myself

- Professor at Lyon 1 University & Head of the DB group at Liris CNRS lab (France)
- Adjunct Professor at the University of Waterloo (Canada)
- Research interests: graph data management, data integration, dataset discovery and curation, log analysis, data management with applications to healthcare, D&I etc.
- Co-authored several publications in major venues of the data management field, two books and an invited paper in ACM Sigmod Record 2018
- Program Chair of ACM Sigmod 2022 and currently Chair of the EDBT Executive Committee (2020-2025).
- Program Director of the International Master DISS (Data and Intelligence for Smart Systems) at Lyon 1 University
Roadmap of my talk

- **Motivations** behind Property Graphs (PG) and Graph Database Systems
- **PG Schema Languages**: from openCypher to PG-Schema compliance
- **PG Constraints**: Schemas and Constraints for Property Graphs
- **Schema discovery**: from Big Data to Machine Learning approaches
- **Concluding remarks**: future directions of investigation
Graphs are everywhere!

Graphs provide a universal and simple blueprint for how to look at the world and make sense of it.

Everyone* uses graphs!

Tech-driving applications = data science + multi-hop relationships

*not yet :-(

[Cartoon by David Somerville, based on a two pane version by Hugh McLeod.]
Graphs as unifying abstractions

- Graphs are **natural abstractions** for representing interconnected objects when encoding, explaining and predicting real-world and digital-world phenomena.

- Graphs are underpinning several **data management ecosystems**, in societal, scientific, RDF, product and digital domains.

- There is **no unique killer application** for graphs, but several exist.

- Nevertheless, the **data models, query languages and system requirements** needed for graphs are constantly evolving.
A plethora of applications

- Among which, the covidgraph.org initiative aiming at building the Covid19 knowledge graph:
  - Collecting patents, publications about the human coronaviruses
  - Biomedical data (genomics and omics)
  - Experimental data about clinical trials
  - Key demographic indicators

- Practical use case in many data-oriented tasks:
  - Property graph schema discovery [Bon22b, Bon22c]
  - Threshold queries in Theory and in the Wild [Bon22a]
Example of threshold graph queries on the Covid19 graph

Find each country that does not have three reports for some age group (in openCypher).

MATCH (c: Country) -[e: CURRENT_FEMALE | CURRENT_MALE | CURRENT_TOTAL]-(a: AgeGroup)
WITH c, a, COUNT(type(e)) AS ecount
WHERE ecount < 3 RETURN c, a;

Find each protein that has more than 43917 associated gene ontology terms.

MATCH (p: Protein) -[: MAPS * . : HAS_ASSOCIATION . (: IS_A *| PART_OF *):]->(t: GOTerm)
WITH p, COUNT(DISTINCT t) AS count_go
WHERE count_go > 43917 RETURN p;

Several graph database engines on the rise

- The number of graph engines is growing over the years as well as their popularity
A Property graph by example

10 : Novice
   salary = 1000
20 : knows
    year = 2016

11 : Expert
    salary = 3000
    year = 2017

21 : worksFor

12 : Apprentice
    salary = 2000
    level = 'A'

13 : Apprentice

22 : knows

23 : knows
    year = 2017

24 : knows

25 : knows

26 : worksFor

27 : worksFor

14 : Apprentice

28 : worksFor

15 : Expert

29 : knows

30 : worksFor
    since = 1997

16 : Expert
Property graphs

Assume pairwise-disjoint sets of:
\( \mathcal{O} \) (objects), \( \mathcal{L} \) (labels), \( \mathcal{K} \) (property keys), and \( \mathcal{N} \) (values)

A property graph is a structure \( V, E \) where

- \( V \subseteq \mathcal{O} \): finite set of objects (vertices)
- \( E \subseteq \mathcal{O} \): finite set of objects (edges)
- \( \eta : E \rightarrow V \times V \):
  assignment of an ordered pair of vertices to each edge
- \( \lambda : V \cup E \rightarrow \mathcal{P}(\mathcal{L}) \):
  assignment of a finite set of labels to each object
- \( \nu : (V \cup E) \times \mathcal{K} \rightarrow \mathcal{N} \):
  partial assignment of values for properties to objects
A lattice of data models

- A data model per use case
- How expressive and human-friendly is a data model?
- Need of making different data models interoperable via mappings or direct translations
Schemas for Graphs: a Fragmented Landscape

(1) ER Models:
   (a) Chen ER
   (b) Extended ER
   (c) Enhanced ER
   (d) ORM2
   (e) UML Class Diagram

(2) RDF Schemas:
   (a) RDFS
   (b) OWL
   (c) SHACL
   (d) ShEx

(3) Tree-shaped Schemas:
   (a) DTD/XML Schema
   (b) JSON Schema
   (c) RELAX NG

(4) Graph Schemas
   (a) GraphQL
   (b) openCypher
   (c) SQL/PGQ

(5) (Limited) Schemas in Graph DBs
   (d) AgensGraph
   (e) ArangoDB
   (f) DataStax
   (g) JanusGraph
   (h) Nebula Graph/nGQL
   (i) Neo4j
   (j) Oracle/PGQL
   (k) OrientDB/SQL
   (l) Sparksee
   (m) TigerGraph/GSQL
   (n) TypeDB/TypeQL
Why do we need Schemas for Property Graphs?

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

- **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)
- **Graph transformations**: mappings between different graph databases are guided by the schemas (source and target schemas)
- **Data integration, Data Quality**: graph database sources to be integrated and monitored for quality
The Design of Property Graph Schemas
From OpenCypher to PG-Schemas
The quest for schemas in graph databases

- **Graph Databases are mainly schema-less**
  - No a priori schema constraints → error-prone data integration & metadata management

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

- **Focus on design of a standard schema language for property graphs**
  - Activity carried out within the LDBC community* ([https://ldbcouncil.org/gql-community/pgswg/](https://ldbcouncil.org/gql-community/pgswg/))

---

[BF19] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, Hannes Voigt: Schema Validation and Evolution for Graph Databases. ER 2019: 448-456

Schema validation for Cypher-like Schemas

- Design of a Cypher-based DDL
- The obtained schema is a Property Graph
- Schema validation via graph homomorphisms
  - Schema nodes define node types
  - Schema relations define relations allowed between types
  - Properties on schema elements define sets of allowed properties
- Schema-instance homomorphism allows to use existing graph rewriting tools.
openCypher Schemas: Syntax

CREATE GRAPH TYPE snb (
  // element types
  Person {
    firstName : STRING, lastName : STRING
  },
  Message {
    creationDate : TIMESTAMP, browserUsed : STRING
  },
  Comment <: Message {},
  Post <: Message {
    imageFile : STRING?
  },
  // node types
  (Person), (Post), (Comment),
  // edge types
  (Person)-[KNOWS]->(Person),
  (Person)-[LIKES]->(Message),
  (Message)-[HAS_CREATOR]->(Person),
  (Comment)-[REPLY_OF]->(Message)
)
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 $L$ ($L \leftarrow P \rightarrow R$)
Toward GQL-compliant Property Graph Schemas

- Schema support is limited in the first version of the GQL standard [GQL-22]
- Ongoing ISO’s Working Group for Database Languages (ISO/IEC JTC1 SC32 W3G [GQL-ISO])

[GQL-22] A. Deutsch et Al. Graph Pattern Matching in GQL and SQL/PGQ. SIGMOD Conference 2022: 2246-2258
[GQL-ISO] https://www.gqlstandards.org/
Toward GQL-compliant Property Graph Schemas

- Limited Support of Schemas in Contemporary Graph Databases
  - 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
- Ongoing standardization of PG schemas (already implemented in SQL/PGQ)

[PGS23] R. Angles et al. PG-Schemas: Schemas for Property Graphs To Appear in ACM Sigmod 2023
[GQL-22] A. Deutsch et Al. Graph Pattern Matching in GQL and SQL/PGQ. SIGMOD Conference 2022: 2246-2258
[GQL-ISO] https://www.gqlstandards.org/
Toward GQL-compliant Property Graph Schemas

- Three possible scenarios:
  - Schema-First (or Prescriptive)
    - schema provided during setup
  - Flexible Schema (or Descriptive)
    - users can use schema as description of what is in the data
  - Partial Schema (Prescriptive and Descriptive co-exist)
    - both prescriptive and descriptive are allowed for the same property graph
PG-Schemas: The Interactive Graph Exploration Use Case

- **(Step 1)** Andrea connects to a data catalog 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.

![Diagram of a fraud graph schema](image)

*Figure 1: A diagram of a fraud graph schema.*
PG-Schemas: The Interactive Graph Exploration Use Case

- (Step 3) The graph explorer automatically construct a search form for Customer, including name and id (inherited from Person type in the schema)
- (Step 4) Andrea needs to understand connections between customers

**Step 4: Connection patterns of interest**

```plaintext
CREATE GRAPH TYPE fraudGraphType STRICT {
  ( personType : Person {name STRING } ),
  ( customerType : personType & Customer {id INT32 } ),
  ( transactionType : Transaction { num STRING } ),
  ( accountType : Account {id INT32 } ),
  ...
  (: transactionType )
  -: [ activityType : deposit | withdraw ]->( : accountType )
}```
PG-Schemas: The Interactive Graph Exploration Use Case

- **(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 lets the user classify the fraudulent cases.

**Step 5: First connection pattern chosen**

(x: Customer) -[: uses]-(: CreditCard)
<-[[: uses]-(y: Customer),
(x: Customer) -[: uses]-(: CreditCard)
<-[[: charges ]-[t: Transaction] -[: charges ]->
(: CreditCard) <-[[: uses ]-(y: Customer)
Requirements: property graph types

- **R1 Node Types**
  - Schemas must allow defining types for nodes that specify their labels and properties.
    
    e.g. `(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**
  - Edge types. Schemas must allow defining types for edges that specify their labels and properties as well as the types of incident nodes.
    
    e.g. `(: personType )-[ friendType : Knows & Likes { since DATE}] - >(: personType )`
    
    `(: personType | customerType )-[ friendType : Knows & Likes {since DATE}]-->(: personType | customerType )`

- **R3 Content Types**
  - Schemas must support a practical repertoire of data types in content types.
  - Support for GQL 1.0 content types (STRING, DATE, INT*) and any other sets of data types
    
    * But also BOOL, FLOAT, TIME, DURATION etc.
Requirements: property graph constraints

- **R4** Key Constraints
  - Schemas must allow specifying key constraints on sets of nodes or edges of a given type.

- **R5** Participation Constraints
  - Schemas must allow specifying participation constraints (e.g. as in ER diagrams).

- **R6** Type Hierarchies
  - Schemas must allow specifying type hierarchies.

  \[ \text{e.g.} (\text{salariedType} : \text{Salaried} \{ \text{salary INT} \}) (\text{employeeType} : \text{personType} \& \text{salariedType}) \]
Requirements: flexibility

- **R7** Evolving data.
  - Schemas must allow defining node, edge, and content types with a finely-grained degree of flexibility in the face of evolving data.

  e.g

  ```
  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.
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.

More details about syntax and formal semantics in [PGS23].

[PGS23] R. Angles, A. Bonifati et al. PG-Schemas: Schemas for Property Graphs To Appear in ACM Sigmod 2023
[GQL-22] A. Deutsch et Al. Graph Pattern Matching in GQL and SQL/PQ. SIGMOD Conference 2022: 2246-2258
[GQL-ISO] https://www.gqlstandards.org/
PG-Schemas: Extensibility

- Range Constraints

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

- Complex datatypes (e.g. STRING ARRAY { 1, 2 })

- Intersections and Unions for Content types:

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

- Cardinality constraints:

```sql
FOR (d: Department ) COUNT 2.. OF e
WITHIN (e: Employee ) -[: worksIn ]->(d) .
```
The Design of Property Graph Constraints
For quality control in graph databases
Key Constraints for Property Graphs

**Keys are ... key in data management** for identifying, referencing and constraining objects. They are main components of PG-Schemas.

For example, Person nodes

- 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)

and discussion Forum nodes are **identified** by the forum’s name and the person who moderates the forum

\[ (:Person) \leftarrow [:hasModerator] - (:Forum) \]
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
PG-Keys

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 of the key and its values in your favourite PG query language (a parameter of PG-Keys). Here we use Cypher-like syntax.

For instance

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

says that “each person is identified by their login”, and

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

says that “each forum with a member is identified by its name and moderator”.

- $e_3$ : Email
  - email: akir@g.jp
  - verified: 17.10.20

- $e_2$ : Email
  - email: ak@fuji.jp
  - verified: 14.07.20

- $p_2$ : Person
  - name: Akira
  - login: akira

- $f_0$ : Forum
  - title: Databases
  - joined: 2019

- $p_1$ : Person
  - name: Hayao
  - login: hkuro

- $e_1$ : Email
  - email: h@oki.nl
  - verified: 12.04.21
Keys for nodes, edges, and properties

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

For instance,

```
FOR p WITHIN (p:Person) IDENTIFIER p.login;
```
says that “each Person node is identified by the value of property login”, and

```
FOR e WITHIN (:Person)-[e:joined]->(:Forum)
IDENTIFIER p,f WITHIN (p:Person)-[e:joined]->(f:Forum);
```
says that “each joined edge is identified by its endpoints (i.e., no other joined edge has the same endpoints, so one cannot join the same forum twice)”.
Identify, reference, and constrain objects

Identification is provided by **IDENTIFIER**:

\[
\text{FOR } f \text{ WITHIN } (f:\text{Forum}) \leftarrow [-[:\text{joined}]-(p:\text{Person})]
\]

**IDENTIFIER** \(f\).\text{name}, \(p\) WITHIN \((f)\) \leftarrow [-[:\text{moderates}]-(p:\text{Person})]

**IDENTIFIER** means:

- **EXCLUSIVE** - no objects in the scope share a key value;
- **MANDATORY** - each object in the scope has at least one key value;
- **SINGLETON** - each object 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. In property graphs, all three are needed.
Easy to validate

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

For instance,

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

holds if both queries below return nothing:

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

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

Incremental validation or batching will require additional mechanisms.

Schema Discovery
From Big Data to Machine Learning
Schema inference for PGs

- Existing schema inference mechanisms (e.g. for RDF) are basic [KK22]:
  - no hierarchies
  - no complex types
- Schema inference using MapReduce on Spark (MRSchema) [Lb21]:
  - considers either node labels or node properties → trade-off
  - property co-occurrence information loss (label-oriented approach) vs. extraneous type inference (property-oriented approach).
- Schema inference using hierarchical clustering [Bon22]
  - Can handle labels and properties at the same time

Code Base: https://gitlab.com/Hgit/pgsinference
Inferring a schema for property graphs: the problem
PG-Schema Definition

\((BT, NT, ET)\)

- **Element Types** \((L, P, M)\)
  - Labels
  - Properties (key, value data type)
  - Mandatory Properties

- **Node Types** \((b, H)\)
  - \(b \in BT\), \(H \subseteq BT\) supertypes

- **Edge Types**
  - Ordinary Edge Types
  - Inheritance Edge Types

- **Subtypes**: \(a \in BT\) s.t. it inherits from another element type (the supertype).
- **Edge Types**:
  - Ordinary Edge Type: \((s, e, t, c)\), with \(s \in NT\) the source node, \(t \in NT\) the target node, \(e \in BT\) the arc, \(c = ((i, k), (j, l)) \in \{0, 1\} \times \{1, N\}\) the cardinality.
  - Inheritance Edge Type: \((s, e, t)\), where \(s = (b, H) \in NT\), \(t \in H\), \(e \in BT\) with a label "SubtypeOf".

With:
- **Subtypes**: \(a \in BT\) s.t. it inherits from another element type (the supertype).
- **Edge Types**:
  - Ordinary Edge Type: \((s, e, t, c)\), with \(s \in NT\) the source node, \(t \in NT\) the target node, \(e \in BT\) the arc, \(c = ((i, k), (j, l)) \in \{0, 1\} \times \{1, N\}\) the cardinality.
  - Inheritance Edge Type: \((s, e, t)\), where \(s = (b, H) \in NT\), \(t \in H\), \(e \in BT\) with a label "SubtypeOf".
Overview of the MRSchema method

Two variants:
- **Labels-oriented**: label sets characterize types
- **Properties-oriented**: labels are properties, property key sets characterize types
MRSchema - Step 1 and Step 2

- **Step 1: Preprocessing & Cardinalities**
  - Convert input PG to proper format
  - Infer edge cardinality constraints

- **Step 2: Types and Data Types Inference (MapReduce*)**
  - Label sets characterize types
MRSchema - Step 3 (cont’d)

● Step 3: Node Hierarchies Inference (Labels variant)
**MRSchema - Step 3 (cont’d)**

**Step 3: Node Hierarchies Inference (Labels variant)**

1. Supertypes inference:
   - Pairwise intersection of label sets.
2. Subtypes inference:
   - Node type with label set A is a subtype of node type with label set B if B \( \subset \) A.
Property-oriented Variant

Labels are properties, property key sets characterize node types:

- Step 1: Unlabeled nodes are also matched
- Step 2: Identification of property co-occurrence information but not optional properties
- Step 3: Property key sets are used for subtypes and supertypes inference.
Comparing the Label-Oriented Variant and the Property-Oriented Variant

Schema from the **labels-oriented** variant:

- Good precision but loss of property-related information

Schema from the **properties-oriented** variant:

- Good recall but many spurious types inferred
Time performances (per step)

Cypher Queries & Python Methods

Preprocessing & Cardinalities

Types and Data Types Inference

Node Hierarchies Inference

Python Methods
A New Clustering-based Method: the DiscoPG system

- **Need of combining labels and properties for type inference with improved precision and recall**

- **Static Case**: discover the schema of a static graph dataset $G$
  - GMM-S: novel *hierarchical clustering* algorithm
    - Based on fitting a Gaussian Mixture Model (GMM).
    - Accounts for **both node label & property** information.
  -

- **Dynamic Case**: update the schema of $G$ upon modifications.
  - I-GMM-D: incremental approach; reuses GMM-S clustering.
  - GMM-D: recomputation approach; memoization-based.


**Code Base**: https://github.com/PI-Clustering/code
A GMM Schema Pipeline

- Gaussian Mixture Model (GMM*) to discover hierarchical node types.
- For every node label, run GMM algorithm to fit a mixture of normal distributions and use the resulting model for clustering.
- Re-iterate procedure in each sub-cluster.

From parent node base type Post to subtypes Post1 and Post2
Schema Quality wrt. Baseline (MRSchema)

- 2-3 discovered types/label (avg) & 3 orders of magnitude more edge types.
- GMMSchema infers up to 3 times more node types, up to 3 orders of magnitude more edge types, up to 7 orders of magnitude more subtype edges (for mb6) and up to double the hierarchy depth.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Node Types</th>
<th>Edge Types</th>
<th>Subtype Edges</th>
<th>Hierarchy Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDBC</td>
<td>17</td>
<td>72</td>
<td>51</td>
<td>5</td>
</tr>
<tr>
<td>Mb6</td>
<td>68</td>
<td>795</td>
<td>786</td>
<td>9</td>
</tr>
<tr>
<td>Fib25</td>
<td>47</td>
<td>427</td>
<td>418</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Node Types</th>
<th>Edge Types</th>
<th>Subtype Edges</th>
<th>Hierarchy Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDBC</td>
<td>7</td>
<td>21</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mb6</td>
<td>5</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Fib25</td>
<td>5</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Node Types</th>
<th>Edge Types</th>
<th>Subtype Edges</th>
<th>Hierarchy Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDBC</td>
<td>17</td>
<td>36</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>Mb6</td>
<td>19</td>
<td>27</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>Fib25</td>
<td>26</td>
<td>106</td>
<td>21</td>
<td>6</td>
</tr>
</tbody>
</table>
GMM Schema Discovery Runtimes wrt. Baseline

(a) LDBC, Fib25, Mb6

(b) Covid19
Concluding remarks

Our vision on Graph Processing Ecosystems
Our vision [CACM 21]

The Future Is Big Graphs: A Community View on Graph Processing Systems


Communications of the ACM, September 2021, Vol. 64 No. 9, Pages 62-71
10.1145/3434642

https://cacm.acm.org/magazines/2021/9/255040-the-future-is-big-graphs/
Graph processing ecosystems

- Complex workflows combining OLTP and OLAP processing are needed in order to handle heterogeneous data and heterogeneous queries and algorithms in full-fledged graph ecosystems.
Graph Processing Ecosystems: Underlying Principles and Knobs

- **Standardized data models**, query languages and schemas, common graph algebra and graph calculus

- **Complex workloads** combining pre-processing, OLAP/OLTP application-driven components

- **Reference Architecture**

- **Scale-up vs. Scale-out**

- **Dynamic and Streaming** endeavours

Thanks for your attention