Transforming Data into Property Graphs with Graflo

Create Property Graphs in 5 minutes with GraFlo

UNAM, Mexico

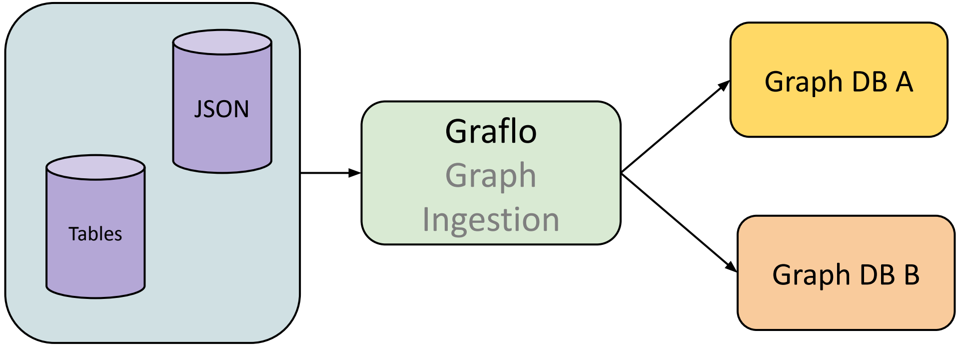
20 November 2025

https://tinyurl.com/graflo

Alexander Belikov, GrowGraph
alexander-belikov.github.io

End-of-Workshop Achievements

1. install graflo
2. ingest a graflo example
3. create a new working schema
4. identify a limitation graflo
5. find a graflo bug
6. contribute to graflo
7. ⭐⭐⭐



Plan

0. Setup
1. Knowledge Graphs: What, Why and How
2. GraFlo : User Perspective
3. GraFlo Internals
4. NB: Graph Database Zoo
5. Practicum

Glossary

Labeled Property Graph (LPG):
a data model to represent (typed) nodes, relationships, and their properties

Graph Database:
an engine to store and query LPGs

Vertex (or Node):
A fundamental unit of data representing an entity (e.g., a **Person**, a **Publication**). Has a type and properties.

Edge (or Relationship):
A directed connection between two vertices (e.g., **owns**, **citedBy**). May have properties.

Graph Schema:
A blueprint defining the allowed types of nodes, relationships, and their properties in the graph.

ETL (Extract, Transform, Load)
The process of moving data from sources to a target system. GraFlo simplifies the Transformation and Loading for graphs.

Declarative Ingestion
Specifying what the graph should be (via schema) rather than writing step-by-step code for how to build it.

Community Detection
the process of grouping nodes into densely connected subgroups within a network

LPG: What, Why, How

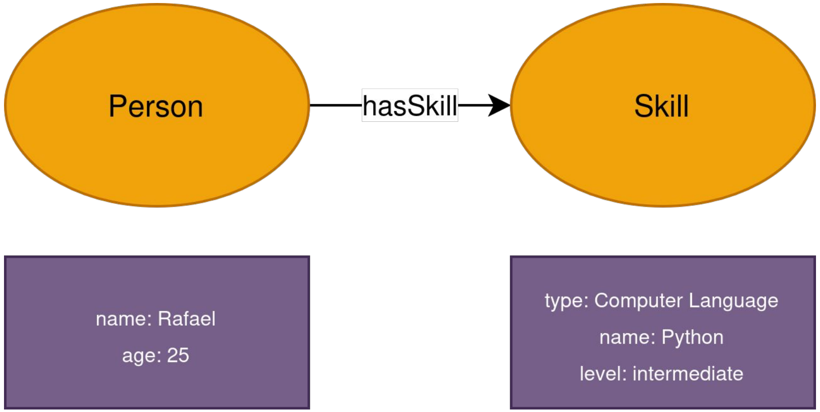
Vertices have types (labels)
Directed relationships have types

Both have properties

Uniqueness constraints (for edges)

Data Structure + Scaffolding

Scaffolding: Query Language, Data Science tools



LPG: What, Why, How

Graphs can be represented as Tables… But

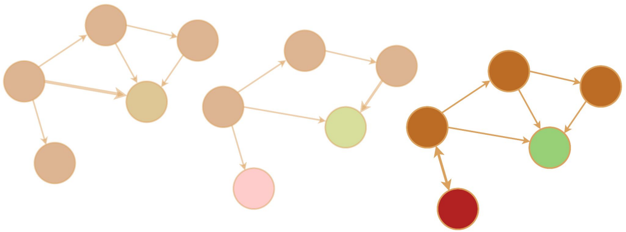
1. 🔗 Relationships: Implicit in SQL (consider 3NF), Explicit in PKG.
2. 🌊Schema fluidity: SQL schemas are structured and require migrations for change in a PKG, these concepts are native.
3. 🚄Superior Performance for Connections, Paths, and Patterns: recursive JOINs for SQL, that degrade exponentially, natural for Graph DBs.
4. 🔭Holistic, Un-siloed Views of Data
5. Native AI/ML Readiness

    SQL: Not designed for graph-native machine learning or structural reasoning.

    PKG: The graph topology is the feature. PKGs directly enable:

    - Graph Neural Networks (GNNs)
    - Community and cluster detection
    - Centrality and influence analysis
    - Knowledge-graph grounding for LLMs
    - Multi-hop reasoning and retrieval

Typical Graph ML tasks



World Model, Digital Twin

Node-Level Tasks

- **Node Classification (Node Labeling):** Predict labels or categories for nodes (e.g., sector classification, fraud detection)
- **Node Regression:** a continuous value for each node (e.g., employee churn probability, asset risk score)
- **Node Embedding:** dense vector representations that capture graph context

Edge-Level Tasks

- **Link Prediction (Edge Prediction):** Predict missing or future edges (e.g., latent relationships between analysts & firms)
- **Edge Classification / Regression:** Predict type/weight of relationship (e.g., sentiment strength, transaction volume)

Graph-Level Tasks

- **Graph Classification:** Predict a label for the entire graph (e.g., classify firm networks by risk profile)
- **Graph Regression:** Predict a continuous value (e.g., fund performance based on its holdings graph)

Subgraph Tasks

- **Subgraph Matching / Querying:** Identify occurrences of a given pattern (e.g., insider trading-like motifs)
- **Subgraph Embedding / Evaluation:** Assess local structure (e.g., influence zones, team dynamics)

Unsupervised & Generative Tasks

- **Community Detection:** Discover tightly connected groups (e.g., analyst echo chambers)
- **Graph Generation:** Model how graphs evolve or simulate new structures (e.g., synthetic networks for stress tests)

LPG: What, Why, How

1. Native Graph Data Structures

- **Node & Edge Storage:** Instead of rows and columns, data is stored as nodes (vertices) and relationships (edges), both capable of holding key-value properties.
- **Pre-Wired Connections:** Relationships are stored as direct pointers or links between nodes. This allows for **constant-time traversal** - jumping from one node to its neighbors is incredibly fast, unlike costly SQL JOINs.
- **Example:** A `Customer` node is directly linked to an `Order` node via a `:PURCHASED` relationship.

2. Scaffolding

- **Indexes:** Accelerate finding the search space for traversals. **Constraints:** Ensure data integrity and uniqueness (e.g., "Ensure `ProductID` is unique for all `Product` nodes"). This prevents duplicate data and maintains a clean graph.

3. Intuitive Graph Query Language

- **Declarative Pattern Matching:** You describe the **shape of the sub-graph** you're looking for, and the engine finds it for you. **Purpose:** Makes complex relationship queries intuitive and readable, turning what would be a multi-paragraph SQL query into a few clear lines of code.

Motivational Problem: Eigenfactor calculation

Goal:
Measure a journal's *true* scientific influence, not just how many papers it publishes.

Idea:
A journal is influential if it is cited by **other influential journals**.
Influence "flows" through the citation network - like reputation spreading.

How it's computed (conceptually):

1. Look at **all citations between journals** over a time window.
2. Count how often *Journal A* cites *Journal B*.
   This forms a **journal-to-journal influence matrix** (a network).
3. Repeatedly pass influence along citation link (similar to PageRank).
4. The stable flow gives each journal an **Eigenfactor score**.

Key point:
The calculation starts by counting **all 3-hop paths**: Journal A ←Pub -(cites)→ Pub →Journal B.

Why Graph Databases Make Eigenfactor Easy

The problem:
To build the influence matrix, we must follow millions of "who-cites-whom" links across several steps.

With files or SQL databases, this involves many complex JOINs and huge intermediate tables.

Why it's painful in SQL or CSV files:

- Citations form a **network**, not a table.
- Multi-step ("A cites B which belongs to C…") queries become slow and hard to write.
- Adding time filters or more hops makes queries balloon in complexity.
- Storing papers, journals, and citations in separate tables hides the structure we need.

Why it's easy in a Graph Database:

- Citations are stored as **edges**, not JOINs.
- Walking "Paper → Cites → Paper → Journal" is a **natural graph traversal**.

AQL query

```
FOR j IN media FILTER j.issn in ["2049-3630", ...]
 RETURN MERGE({{ja : j.issn}},
  {{stats : (
   FOR p in 1 INBOUND j publications_media_edges
    FILTER p.year == _year
    FOR p2 in 1 OUTBOUND p publications_publications_edges
     FILTER p2.year < _year AND p2.year >= (_year - delta)
     FOR j2 in 1 OUTBOUND p2 publications_media_edges
      FILTER j2.issn in ["2049-3630", ...]
   COLLECT jbt = j2.issn WITH COUNT INTO size SORT size DESC
   RETURN {{jb : jbt, s : size}}
  ) }}
)
```

How to Ingest Data to Graph Databases?

Tell the a GDB about

- the correspondences about the fields in the data and properties of the vertices and edges
- which fields should be treated as unique
- which pairs of sets of fields form an edge between two vertices

Pain point

Manual ETL Mapping Overhead: Every source requires custom scripts or queries to map raw data to nodes/edges. This becomes unmanageable as sources grow.

Schema Drift and Rigidity: Adapting graph schema when data models change (e.g. new fields or entity types) is error-prone and lacks abstraction.

Duplication of Logic Across Pipelines: Repeating transformation logic in multiple ingestion scripts leads to code duplication, versioning issues, and maintenance debt.

No Separation of Concerns: Data mapping, transformation, and loading are tightly coupled, making reuse or schema redesign difficult.

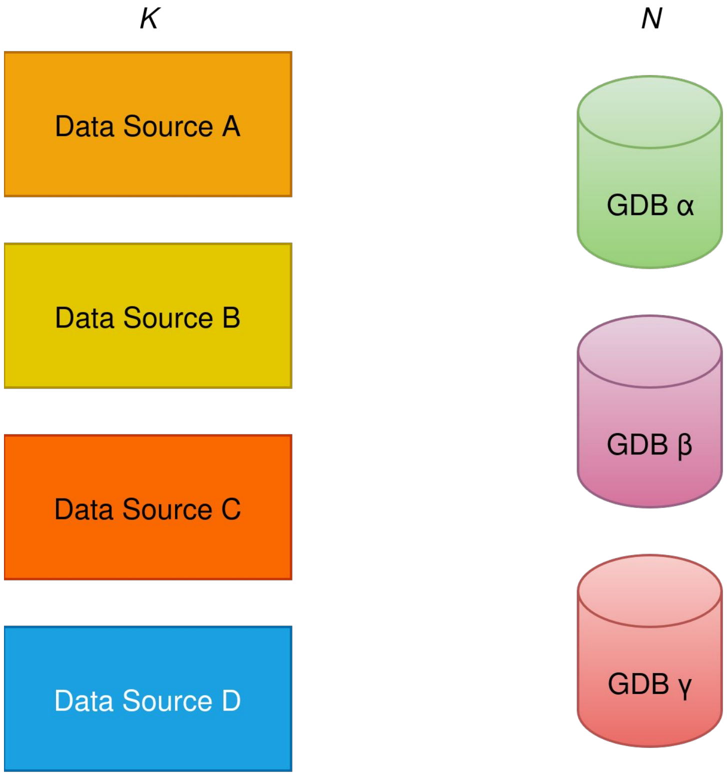
Lack of Reusability and Documentation: Imperative ingestion pipelines (e.g. in GSQL, Cypher, Python) are often undocumented and hard to share or audit.

Hard to Support Multiple Backends: Each target DB (Neo4j, TigerGraph, etc.) requires different ingestion syntax and connectors—no portability across engines.

Poor Scalability Without Parallelism: Ad hoc scripts often process sequentially and can't leverage multiple cores for high-volume data ingestion.

No Declarative Validation or Indexing: Constraints like unique IDs or compound indexes must be manually enforced or scripted, increasing risk of inconsistencies.

How to Ingest Data to Graph Databases?

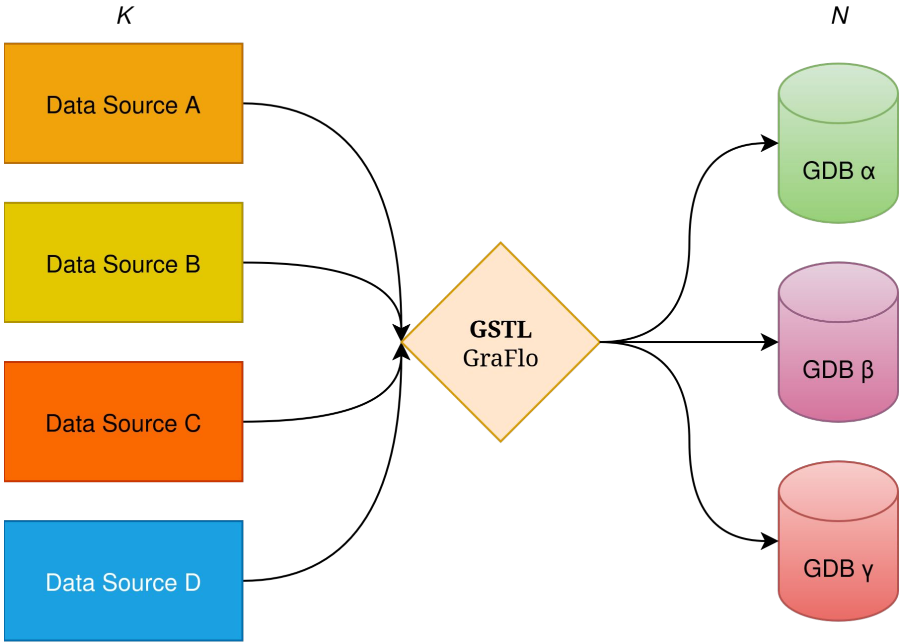


A complex web of *K x N* pipelines

Clearly there must be a Graph Schema & Transformation Language (**GSTL**) that encodes separately

- transformations for each of *K* sources
- graph schema
- load functions for each of *N* databases

Solution: Graflo



Developer Perspective: implement N backends.
User Perspective: implement K data source adapters.

Features:

- **Declarative transformations** vs. custom ETL coding
- Adapters for Neo4j, ArangoDB and TigerGraph: Multi-database adapter eliminates vendor lock-in
- Tested on graphs with billions of edges

GraFlo: User Perspective

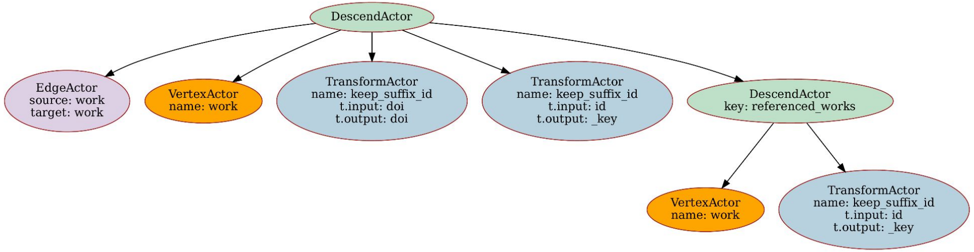
Workflow

- Study your dataset
- Create a mental model
- Create a schema
- Visualize the schema
- Set up the backend
- Ingest
- Check the ingestion results

Schema setup

1. Define Vertices
   a. Properties (optional types)
   b. Indexes
2. Define Edges
   a. Properties/Weights
   b. Indexes
3. Define Resources
   a. Transformations
   b. Vertex Mappings
   c. Edge Mappings (optional)
4. Define a Vocabulary of Transforms
5. Schema metadata

Plotting



plot_schema

This command creates multiple visualizations of the schema:
1. Vertex-to- vertex relationships
2. Vertex fields and their relationships
3. Resource mappings

The visualizations are saved to the specified output path.

Args:
    schema_path: Path to the schema configuration file
    figure_output_path: Path where the visualization will be saved
    prune_low_degree_nodes: Whether to remove nodes with low connectivity from the visualization (default: False)

Example: $ uv run plot_schema -c schema.yaml -o schema.png

Options:
  -c, --schema-path PATH [required]
  -o, --figure-output-path PATH [required]
  -p, --prune-low-degree-nodes BOOLEAN

Schema: Vertex Config and Edge Config

VertexConfig: Defines vertex collections (nodes) in the graph
EdgeConfig: Defines edge collections (relationships) between vertices

VertexConfig(
    vertices: list[Vertex],
    blank_vertices: list[str],
    db_flavor: DBFlavor
)

EdgeConfig(edges: list[Edge])

```yaml
vertex_config:
  vertices:
    - name: person
      fields: [id, name, age]
      indexes: [{fields: [id]}]

edge_config:
  edges:
    - source: person
      target: department
```

Extended Vertex Example

```yaml
vertex_config:
  vertices:
    - name: person
      fields: [id, name, age]
      indexes:
        - fields: [id]
        - unique: false
          fields: [name, age]
      filters: []
      dbname: person
```

Extended Edge Example

```yaml
edge_config:
  edges:
    - source: person
      target: department
      relation_field: role
      weights:
        direct: [date, score]
        vertices:
          - vertex: person
            fields: [age]
      indexes:
        - fields: [date]
```

Transforms

Setting Up Transforms in Schema

1. Global Transform Library (Schema Level)

    Define reusable transforms in the schema's transforms section

2. Inline Transforms (Resource Level)

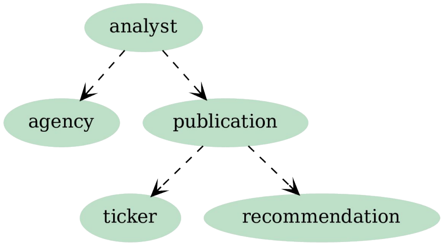
```yaml
transforms:
  keep_suffix_id:
    foo: split_keep_part
    module: graflo.util.transform
    params:
      sep: "/"
      keep: -1
    input: [id]
    output: [_key]
```

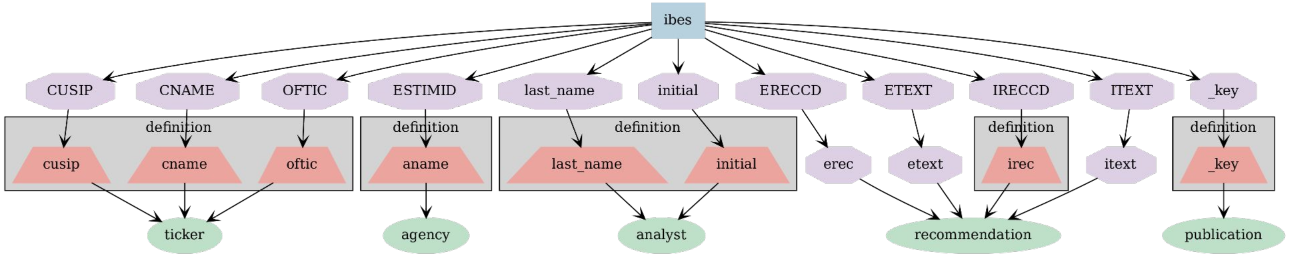
Transforms map and transform fields during ingestion.

1. Declarative mapping: rename/remap fields (no code)
2. Functional transforms: custom Python functions for complex transformations
3. Vertex mappings (special case, check example 1)

Example: Analyst Reports [IBES]

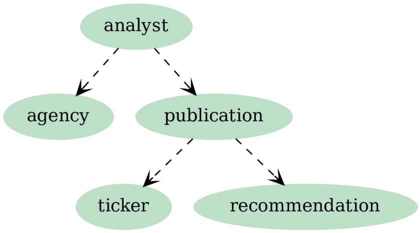
TICKER	CUSIP	CNAME	OFTIC	ACTDATS	ESTIMID	ANALYST	ERECCD	ETEXT	IRECCD	ITEXT
0000	87482X10	TALMER BANCORP	TLMR	20140310	RBCDOMIN	ARFSTROM J	2	OUTPERFORM	2	BUY
0000	87482X10	TALMER BANCORP	TLMR	20140311	JPMORGAN	ALEXOPOULOS S	NaN	OVERWEIGHT	2	BUY
0000	87482X10	TALMER BANCORP	TLMR	20140311	KEEFE	MCGRATTY C	2	OUTPERFORM	2	BUY





[Institutional Brokers' Estimate System (IBES)]

IBES Schema: Vertices



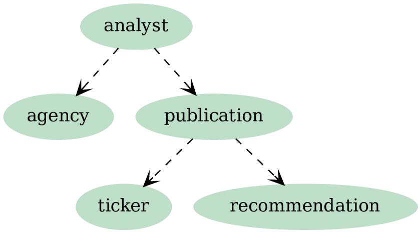
ibes.yaml

```yaml
vertex_config:
    blank_vertices:
    -   publication
    vertices:
    -   name: publication
        dbname: publications
        fields:
        -   datetime_review
        -   datetime_announce
        indexes:
        -   fields:
            -   _key
        -   type: hash
            unique: false
            fields:
            -   datetime_review
        -   type: hash
            unique: false
            fields:
            -   datetime_announce
```

ibes.yaml

```yaml
vertex_config:
    vertices:
    -   name: ticker
        dbname: tickers
        fields:
        -   cusip
        -   cname
        -   oftic
        indexes:
        -   fields:
            -   cusip
            -   cname
            -   oftic
    -   name: agency
        dbname: agencies
        fields:
        -   aname
        indexes:
        -   fields:
            -   aname
```

IBES Schema: Vertices & Edges



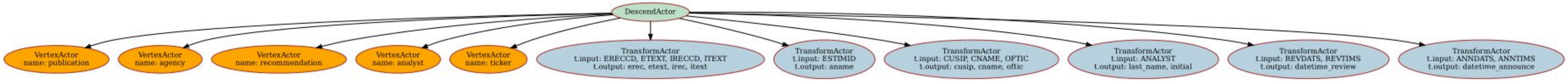

```yaml
vertex_config:
    vertices:
    -   name: analyst
        dbname: analysts
        fields:
        -   last_name
        -   initial
        indexes:
        -   fields:
            -   last_name
            -   initial
    -   name: recommendation
        dbname: recommendations
        fields:
        -   erec
        -   etext
        -   irec
        -   itext
        indexes:
        -   fields:
            -   irec
            -   erec
            -   etext
            -   itext
edge_config:
    edges:
```

```yaml
edge_config:
    edges:
    -   source: publication
        target: ticker
    -   source: analyst
        target: agency
        weights:
            vertices:
            -   name: publication
                keep_vertex_name: false
                fields:
                -   datetime_review
                -   datetime_announce
    -   source: analyst
        target: publication
    -   source: publication
        target: recommendation
```

IBES Schema: Transforms

```yaml
resources:
-   resource_name: ibes
    apply:
    -   vertex: ticker
    -   vertex: analyst
    -   vertex: recommendation
    -   vertex: agency
    -   foo: parse_date_ibes
        module: graflo.util.transform
        input:
        -   ANNDATS
        -   ANNTIMS
        output:
        -   datetime_announce
    -   foo: parse_date_ibes
        module: graflo.util.transform
        input:
        -   REVDATS
        -   REVTIMS
        output:
        -   datetime_review
```

```yaml
resources:
-   resource_name: ibes
    apply:
    -   foo: cast_ibes_analyst
        module: graflo.util.transform
        input:
        -   ANALYST
        output:
        -   last_name
        -   initial
    -   map:
            CUSIP: cusip
            CNAME: cname
            OFTIC: oftic
    -   map:
            ESTIMID: aname
    -   map:
            ERECCD: erec
            ETEXT: etext
            IRECCD: irec
            ITEXT: itext
```



Extra Functions

Beyond ingestion, GraFlo provides utilities for querying, filtering, and managing graph data.

fetch_present_documents() - Check which documents from a batch already exist

keep_absent_documents() — Get documents that don't exist in the database

aggregate() — Perform aggregations on collections

- Supports COUNT, SUM, AVG, MIN, MAX
- Group by a discriminant field
- Apply filters

GraFlo internals: Resource Transformation to Vertices & Edges

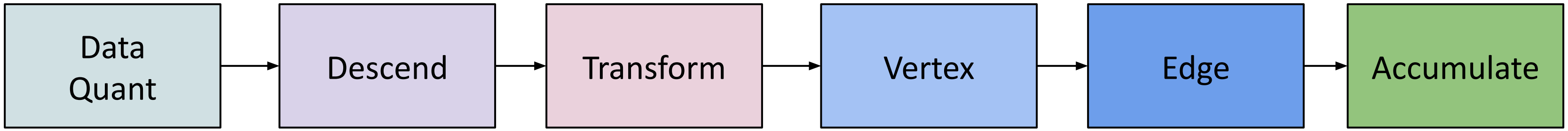
Actors are processing units that transform raw data into graph structures.

Raw data is a json (potentially nested).

They run in sequence within a Resource, each handling a specific transformation step.

Descend actor is responsible for recursive processing and mapping.

Four actor types, executed in priority order:



GraFlo internals: Resource Transformation to Vertices & Edges

Actors are processing units that transform raw data into graph structures. They run in sequence within a Resource, each handling a specific transformation step.

Actor Types: four actor types, executed in priority order

DescendActor (priority 10):

- Processes nested/hierarchical data structures
- Expands lists or nested dictionaries
- Executes child actors for each nested item
- Maintains location tracking via LocationIndex

TransformActor (priority 20):

- Applies data transformations
- Executes transform functions (e.g., field mapping, data cleaning)
- Results stored in buffer_transforms or buffer_vertex
- Can target specific vertices

VertexActor (priority 50): Creates vertex documents

- Extracts fields from documents
- Merges data from transforms and buffers
- Accumulates vertices in ctx.acc_vertex[vertex_name][lindex]
- Applies filters and field selection

EdgeActor (priority 90): Creates edges between vertices

- Merges vertices first (via merge_vertices())
- Renders edges based on vertex matches
- Calculates edge weights from vertex fields

Graph Databases

Database	License	Query Language	Horizontal Scaling	Comment
JanusGraph	Apache 2.0	Gremlin	Native	Best for massive scales
Nebula	Apache 2.0	nGQL	Native	
Dgraph	Apache 2.0	GraphQL	Native	
Apache AGE	Apache 2.0	Cyper+SQL	PostgreSQL clustering	
Memgraph	BSL	Cypher	Enterprise only	In-memory focus
Neo4j	GPLv3	Cypher	Enterprise only	Mature Ecosystem
ArangoDB	BSL	AQL	Enterprise only	Close to noSQL
TigerGraph	Prop	GSQL	Enterprise only	Typed Schema

Graph DB Extras

Web Interface

- **Neo4j**: Bloom (visual exploration), Browser (Cypher queries)
- **ArangoDB**: Web UI (queries, management, graph visualization)
- **TigerGraph**: GraphStudio (visual schema design, query building)

Graph Data Science

- **Neo4j**: **GDS Library** - 70+ production-grade algorithms (PageRank, community detection, node similarity, machine learning pipelines)
- **ArangoDB**: **Graph Analytics** - Custom implementations using Pregel, integrated with machine learning connectors
- **TigerGraph**: **Built-in GSQL** - 30+ parallel graph algorithms optimized for massive-scale analytics

Key Differentiators

- **Neo4j**: Most mature GDS ecosystem, enterprise support
- **ArangoDB**: Multi-model analytics (document + graph)
- **TigerGraph**: Native parallel execution on distributed data

Graph DB Idiosyncrasy

Neo4j
Pure Labeled Property Graph. Vertices have labels (types). Indexes over multiple properties. Types not enforced but possible to set constraints. Type mismatches may only surface at runtime.
Schema-last.

ArangoDB
Document-Graph Hybrid models. Types are collections (of JSONs). Edges store handles to nodes.
Automatic index on **_key** (can be manipulated). Can represent nested data.
Type agnostic but possible to enable type validation.

Tigergraph
Schema-first, rigid, strongly-typed. Secondary indexes can be only single property. Graphs have to be composed from globally defined vertices and edges.

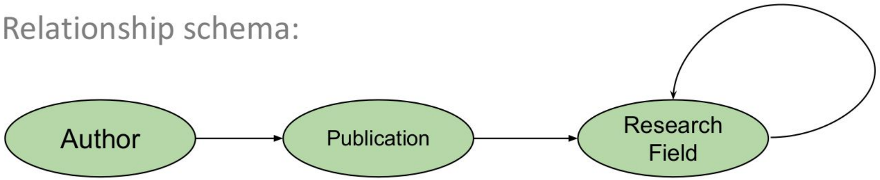
Nomenclature.

Application: Reviewer Recommendation

Authors

```
author_id,FullName,HIndex,research_sector
309238221625,Guillaume Lemaître,10,32057259
747324850364,Patrick L. Meras,4,8258574
987843024183,S. I. Konovalov,5,30262949
```

Relationship schema:



Research Fields

```
field_of_study_id,display_name,level,children
87687168,Digital audio,4,6800068
87687168,Digital audio,4,24579023
87687168,Digital audio,4,30246029
```

Publications

```
PublicationId,authors,topics,publication_year,Doi
465031,"['id:300648343950, name:Tadeusz Kaczorowski'
 'id:566936217113, name:Anna-Karina Kaczorowska'
 'id:214748948560, name:Sebastian Dorawa']",[185592680  89423630],2019,10.3390/V11070657
```

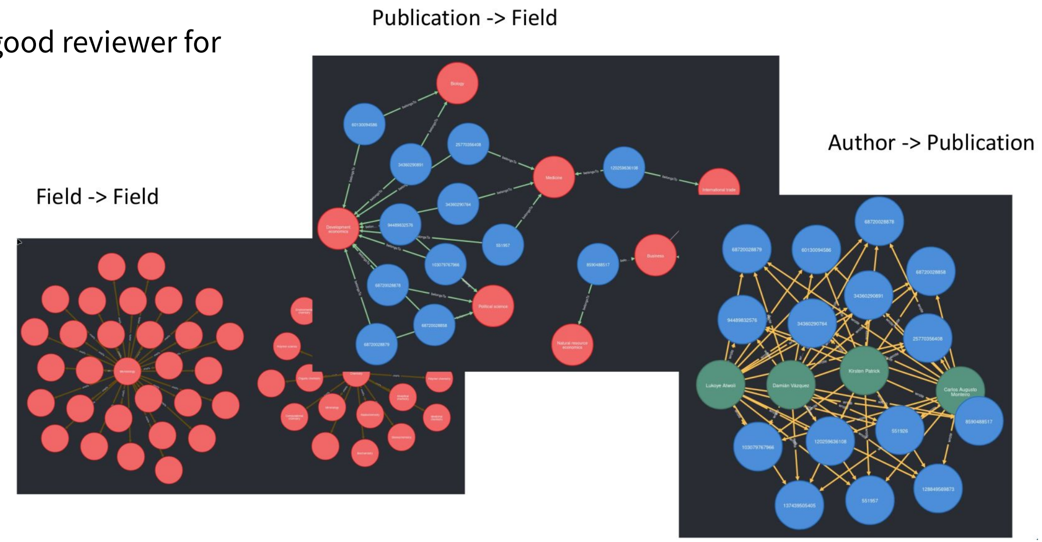
Application: Reviewer Recommendation

Problems

1. The same author may have multiple ids.
2. Given the publication records, who would be a good reviewer for paper?

Solutions

1. Disambiguate
    a. construct **[coAuth] Author ↔ Author** from Author → Publication
    b. derive communities
    c. fuzzy match within communities
2. Recommend Reviewers
    a. construct **[coDomain] Author ↔ Author** from Author → Publication → ResearchField
    b. derive communities in **[coDomain] Author ↔ Author**
    c. For each new publication pick reviewers from the same **[coDomain]** comm id that have a different **[coAuth]** comm id



Practicum

Check you .env files (ports specifically, but also username/password)

ArangoDB Web Interface: http://localhost:8535

Neo4j Web Interface: http://localhost:7475

Try to ingest examples from https://github.com/growgraph/graflo/tree/main/examples

and visualize them using Web Interface.

Practice Creating Schemas

1. grouplens.org/datasets/movielens/20m/
2. https://www.kaggle.com/datasets/rmisra/news-category-dataset
3. https://www.kaggle.com/datasets/mylesoneill/magic-the-gathering-cards
4. [secret dataset]

Roadmap

1. Improve API UX (how easy is it for developer to use the package).
2. Implement SQL schema to GraFlo schema generator
3. Add SQL API: ingestion of SQL resources
4. Add Schema validation and version control
5. Add Nebula and Janus as Graph Database backends.

Conclusion

- The World is a Graph: The most complex and connected data is naturally modeled as a graph.
- GraFlo Makes it Practical: It eliminates the ETL bottleneck, providing a declarative, scalable, and multi-database framework for building your knowledge graph.
- You Are Now Equipped: You can install GraFlo, design schemas, ingest data, and leverage the power of graph databases in minutes, not days.