URLIFE-COMMON: A Graph-Relational Scheme for Representing Functional Data Structures

Greg Coppola

July 15, 2025

Abstract

This paper introduces a unifying framework for modeling common digital information structures—such as folder hierarchies, documents, spreadsheets, and project management schemas—using graph-relational databases. While these structures are traditionally treated as distinct and domain-specific, we argue that they share a common underlying form: hierarchical or relational data with typed entities and labeled relationships. By expressing these structures as graphs composed of nodes (representing objects such as files, paragraphs, or tasks) and edges (capturing containment, ordering, dependency, or linkage), we demonstrate how a graph-relational model provides a flexible, expressive, and queryable representation that transcends domain boundaries. This approach supports recursive queries, semantic linking, and content reuse, offering a foundation for unified tooling and storage systems across productivity and knowledge applications.

1 Introduction

Modern computing environments rely heavily on structured information systems: file systems organize data hierarchically, documents embed content in nested blocks, spreadsheets lay out data in two-dimensional grids, and project plans coordinate goals, dependencies, and outcomes. Though these systems appear distinct in interface and semantics, they share a common foundation: they consist of identifiable entities linked by typed relationships. In this paper, we refer to these as *functional data structures*.

Traditionally, such systems have been implemented using relational databases, which model entities as rows in normalized tables and encode relationships using foreign keys. While relational models are robust and well-suited to tabular data, they can become rigid or inefficient when dealing with deeply nested, recursive, or heterogeneous structures.

Graph-relational databases offer a compelling alternative. In a graph model, entities are represented as nodes, and relationships between them as labeled edges. This approach naturally captures containment, ordering, dependency, and reference relationships—features that underlie many of the data structures

described above. Moreover, graph-relational systems preserve the power of relational models while supporting flexible traversal and schema evolution.

This paper explores how a unified graph-relational framework can model a range of functional data structures: folders and file hierarchies, hierarchical documents, spreadsheets, and project plans. By mapping these seemingly unrelated systems to a shared graph-based schema, we enable more expressive querying, more consistent tooling, and the potential for integrated platforms that bridge traditional application boundaries.

2 Background on Graph-Relational Databases

In this section, we introduce the core data modeling concepts used throughout the paper. We begin with relational databases, which structure data using tables and foreign keys. We then describe graph-based models, which represent data as nodes and edges. Next, we outline common graph structures such as trees, DAGs, and general graphs. Finally, we explain how these models come together in graph-relational databases, which combine the strengths of both paradigms.

2.1 Relational Databases

A *relational database* is a type of database management system (DBMS) that organizes data into structured collections of tables, also known as *relations*. Each table consists of rows and columns, where:

- Each **row** represents a single record or tuple.
- Each **column** corresponds to an attribute or field of the data.

Relational databases use a strict schema to define the structure of data and the types of each attribute. Relationships between different entities are encoded using *foreign keys*, which reference primary keys in other tables. These constraints enable the database to enforce referential integrity and support complex queries across multiple tables.

Data in relational databases is typically queried using *Structured Query Language* (SQL), which allows users to perform joins, filters, aggregations, and updates in a declarative manner.

Relational databases are well-suited for applications with clearly defined, tabular data models and transactional workloads, such as financial systems, inventory tracking, and customer relationship management (CRM). However, they become increasingly cumbersome for data with deep, recursive, or flexible relationships, such as social networks or hierarchical file systems, where complex joins and schema rigidity can lead to inefficiencies.

Common relational database systems include **PostgreSQL**, **MySQL**, **SQLite**, and **Microsoft SQL Server**.

2.2 Graphs as Data Models

Graphs provide a flexible and expressive way to model interconnected data. A graph consists of a set of *nodes* (also called *vertices*) and a set of *edges* (or *links*) that connect pairs of nodes. Each node represents an entity or object, while each edge represents a relationship between two entities. In modern graph-based data models, both nodes and edges can carry associated *properties* in the form of key-value pairs.

Nodes represent discrete units of data. In the context of this paper, nodes can correspond to files, folders, paragraphs, spreadsheet cells, project plans, goals, or other conceptual entities. Each node can have a `type` label (e.g., `Folder`, `Sentence`, `Goal`) and a set of properties (e.g., `name`, `created_at`, `text`).

Edges represent labeled, directed relationships between nodes. These may include containment (e.g., `HAS_CHILD`), ordering (e.g., `FOLLOWS`), linkage (e.g., `LINKS_TO`), or dependency (e.g., `DEPENDS_ON`). Each edge can also carry properties, such as timestamps, annotations, or metadata relevant to the relationship.

Typed Graphs are graphs in which both nodes and edges are labeled with types, enabling structured querying and inference. This is especially useful in modeling functional data structures, where we might distinguish between a `Plan` node and a `File` node, or between a `HAS_SECTION` edge and a `DEPENDS_ON` edge.

Graphs are particularly well suited for modeling *semantic*, *hierarchical*, and *relational* structures that evolve over time and contain deeply nested relationships—characteristics shared by all the functional data structures discussed in this work.

2.3 Types of Graph Structures

Graph-relational models can express a wide variety of structural patterns found in real-world data. While the general definition of a graph permits arbitrary connections between nodes, many functional data structures encountered in practice follow more specific graph topologies. Understanding these categories helps in choosing appropriate query strategies and constraints for each domain.

Trees A tree is a directed, acyclic graph in which each node (except the root) has exactly one parent. Trees are ideal for representing strictly hierarchical structures, such as:

- Folder hierarchies in a file system
- Document outlines (e.g., chapters → sections → paragraphs)
- Nested JSON or XML structures

Tree structures simplify traversal and allow for efficient operations such as computing paths or aggregating data within subtrees.

Directed Acyclic Graphs (DAGs) A DAG is a directed graph with no cycles, but unlike trees, nodes may have multiple parents. This allows for shared substructures and converging dependencies. DAGs are well-suited for:

- Project plans, where tasks may have multiple dependencies
- Spreadsheets with formulas referencing multiple cells
- Document version histories or workflow pipelines

DAGs support topological sorting, dependency resolution, and reuse of components across multiple higher-level structures.

General Graphs A general graph permits cycles, undirected edges, and arbitrary relationships between nodes. This structure is required when:

- Cross-references exist between sections of a document
- Files are linked via symbolic links or symlinks
- Notes, concepts, or entities are interconnected via backlinks or citations

General graphs are the most expressive, supporting complex query patterns and semantic search, but may require additional logic to avoid infinite traversals or cycles.

Hybrid Models In practice, many systems exhibit a hybrid of these graph structures. For instance, a document may be a tree in its outline form, but also include cross-links (forming a DAG or general graph). A folder structure may follow a tree, but include symbolic links to create cycles. The graph-relational model accommodates such flexibility by encoding both structure and meaning in labeled, typed edges.

2.4 Graph-Relational Databases

Graph-relational databases are a class of data management systems that combine the strengths of traditional relational databases with the expressive power of graph-based models. These systems are designed to represent and query richly interconnected data more naturally and efficiently than purely tabular approaches.

In a graph-relational model, data entities are represented as *nodes*, and the relationships between them as directed *edges*, both of which can carry associated *properties* as key-value pairs. This design enables highly flexible modeling of complex structures, including hierarchies, networks, and semantic relationships.

This paper focuses on a specific insight: many core digital structures—such as file systems, documents, spreadsheets, and project plans—can all be expressed using this node-and-edge paradigm. Despite differences in interface and

function, these systems share common traits: they are composed of identifiable units (e.g., files, paragraphs, cells, tasks) and explicit relationships (e.g., containment, dependency, sequence). A graph-relational approach provides a unified and semantically rich representation for all of them.

Graph-relational databases typically fall into two broad categories:

1. **Graph-native databases**, such as Neo4j, TigerGraph, and Amazon Neptune, which are built around graph theory and expose specialized query languages like Cypher or Gremlin.
2. **Relational databases with graph extensions**, such as Microsoft SQL Server and PostgreSQL with extensions like `pg_graph` or `AGE`, which allow graph operations to be executed within a relational environment using extended SQL.

These systems excel at executing recursive and relationship-centric queries, including:

- Traversing multi-level hierarchies (e.g., folder trees or document outlines)
- Resolving dependencies (e.g., spreadsheet formulas or project plans)
- Discovering transitive relationships (e.g., document references or task chains)

For example, in a social network, a query like "friends of friends who liked a post" can be expressed concisely in a graph query language, while the same logic in a relational system might require multiple self-joins and recursive common table expressions (CTEs).

By combining the robustness of relational modeling with the flexibility of graph traversal semantics, graph-relational databases offer a powerful foundation for modeling functional data structures—enabling more expressive queries, schema evolution, and structural reuse across traditionally distinct application domains.

2.5 Some Important Use Cases for Graph-Relational Databases

Graph-relational databases (graph databases or relational databases with graph-like capabilities) are especially useful for **modeling and querying complex relationships**. Here are some major **use cases**:

1. **Social Networks**
   - **Use case:** Model users, friendships, likes, comments, etc.
   - **Why graph works well:** Relationships are first-class citizens. You can easily ask questions such as:
     - Who are my friends of friends?
     - What is the shortest connection path between two people?
2. **Fraud Detection**

- **Use case:** Detect suspicious patterns, e.g., multiple accounts linked by the same IP or device.
- **Why graph works well:** You can track indirect connections across multiple hops:
  - Which accounts are linked through shared bank accounts, addresses, or devices?

3. **Recommendation Systems**
   - **Use case:** Suggest products, content, or friends based on a user's behavior and connections.
   - **Why graph works well:** You can model "users bought this," "users viewed that," and traverse the network of interactions.
4. **Knowledge Graphs**
   - **Use case:** Build semantic models of knowledge, such as in search engines, AI assistants, or R&D.
   - **Why graph works well:** Flexible schema; you can model complex hierarchies and ontologies (e.g., "Isaac Newton is a physicist who discovered gravity").
5. **Identity and Access Management (IAM)**
   - **Use case:** Model users, roles, permissions, resources.
   - **Why graph works well:** Permission inheritance, group nesting, and role-based access involve many-to-many and hierarchical relationships.
6. **Supply Chain and Logistics**
   - **Use case:** Model suppliers, shipments, routes, and dependencies.
   - **Why graph works well:** You can ask:
     - What is the most efficient path from supplier A to customer Z?
     - What downstream items are affected if this factory shuts down?
7. **Network and IT Infrastructure**
   - **Use case:** Model physical and virtual network topologies.
   - **Why graph works well:** Helps trace dependencies:
     - If this router fails, what systems are impacted?
8. **Legal or Investigative Intelligence**
   - **Use case:** Link people, companies, transactions, and events in legal investigations (e.g., Panama Papers).

- **Why graph works well:** Helps trace indirect connections and ownership structures.

9. **Biological or Scientific Research**
   - **Use case:** Model protein interactions, gene regulation, or citations between scientific papers.
   - **Why graph works well:** These domains involve dense, highly interconnected data.
10. **Graph-Based Search and Navigation**
   - **Use case:** Provide context-aware or relationship-based search.
   - **Why graph works well:** Enables semantic and path-based searching.

3 File Systems, Documents, Spreadsheets, and Databases

In this section, we examine four common functional data structures—file systems, documents, spreadsheets, and project planning frameworks. We use the term "functional" to emphasize that each structure plays a practical role in organizing work, supporting reasoning, or enabling decision-making. Each subsection describes the structure and typical features of these systems, laying the groundwork for how they can be unified under a graph-relational representation.

3.1 File Systems

A *file system* is a fundamental component of an operating system that defines how data is stored, organized, and accessed on storage media such as hard drives, solid-state drives, and removable devices. It provides a structured way to manage data in the form of files and directories (also known as folders), enabling users and applications to read, write, and modify content reliably.

Most modern file systems adopt a *hierarchical* structure in which:

- Files and directories are arranged in a tree-like format.
- Each directory may contain files or other subdirectories.
- The entire structure originates from a single *root* directory.

In addition to organizing file content, a file system maintains *metadata* associated with each file or directory, such as:

- File name and full path
- Size and creation/modification timestamps

- Access permissions and ownership

Advanced file systems also support features such as *symbolic links* (which point to other files or directories), *hard links* (multiple directory entries referencing the same physical data), journaling, versioning, and access control lists (ACLs).

Common file systems include **NTFS** (used in Windows), **ext4** (common in Linux distributions), **APFS** and **HFS+** (used by macOS), and **FAT32** or **exFAT** (used in portable storage devices). The design and capabilities of a file system significantly influence the performance, reliability, and flexibility of the computing environment.

3.2 Documents as Hierarchical Structure

Documents are structured containers of written content designed to communicate information, ideas, or instructions. While documents may appear as linear sequences of text, they almost always exhibit a well-defined hierarchical structure that organizes content into progressively finer levels of granularity.

At the highest level, a document may be divided into *chapters* or *sections*, which are further broken down into *subsections*, *paragraphs*, and *sentences*. This tree-like structure reflects both the logical organization of the document and the syntactic nesting of its components.

- A document consists of chapters or sections.
- Sections consist of subsections and paragraphs.
- Paragraphs contain sentences, lists, or figures.
- Sentences are composed of words, punctuation, and possibly inline elements (e.g., citations or links).

This hierarchical design enables a range of computational and organizational benefits:

- It supports semantic navigation, such as table-of-contents generation and outline views.
- It allows selective rendering or reordering of parts of the document.
- It enables fine-grained access for tasks such as annotation, parsing, and information extraction.

Hierarchical models are particularly important in document-centric applications such as content management systems, editors (e.g., `LaTeX`, Markdown, HTML), and natural language processing pipelines. In many modern systems, this structure is represented explicitly as a tree or graph, with each node corresponding to a structural unit (e.g., a heading, paragraph, or footnote) and edges encoding containment or ordering relationships.

Understanding the hierarchical nature of documents is essential for tasks that require structural reasoning, such as outlining, summarization, and semantic linking between document parts.

3.3 Spreadsheets

Spreadsheets are digital tools used to organize, compute, and analyze data in a tabular format. Each spreadsheet consists of a grid of *cells*, organized into rows and columns. A single spreadsheet file may contain multiple *sheets*, each representing a distinct table or logical grouping of data.

Cells in a spreadsheet can hold various types of content:

- **Text**, such as labels or category names.
- **Numerical values**, for use in calculations.
- **Formulas**, which perform computations or reference other cells (e.g., `=A1 + B1`).
- **Functions**, such as `SUM()`, `AVERAGE()`, or `IF()`.

Spreadsheets support both direct data manipulation and automated computation through cell referencing and formula propagation. For example, a formula in one cell can depend on the values of other cells, creating a dependency graph of computations.

While spreadsheets are often used for straightforward tasks like data entry and formatting, they also support complex analytical workflows. Users can model financial systems, perform statistical analyses, and create dynamic dashboards with charts and pivot tables.

Popular spreadsheet software includes **Microsoft Excel**, **Google Sheets**, **LibreOffice Calc**, and **Apple Numbers**. Due to their accessibility and flexibility, spreadsheets are ubiquitous across business, education, research, and personal productivity.

Despite their visual grid-based interface, spreadsheets can be interpreted as structured data representations—with rows as records, columns as attributes, and cell references forming implicit data flows. This has led to interest in treating spreadsheets as relational tables, logical programs, or even directed graphs in certain analytical contexts.

3.4 Project Management

Project management is the discipline concerned with initiating, planning, executing, controlling, and completing a set of coordinated tasks to achieve specific goals within defined constraints. A *project* is typically characterized as a temporary endeavor undertaken to create a unique product, service, or result. It operates under limitations of time, budget, scope, and resources.

Effective project management involves several key activities:

- Defining clear objectives and deliverables.
- Breaking down work into manageable tasks.
- Scheduling tasks and estimating timelines.
- Allocating resources and assigning responsibilities.
- Monitoring progress and adjusting plans as needed.
- Ensuring that the project is completed on time, within budget, and according to specifications.

Project management practices are supported by various structured methodologies, including:

- **Waterfall**: A linear, phase-based model in which each stage must be completed before the next begins.
- **Agile**: An iterative and incremental approach emphasizing flexibility, collaboration, and continuous feedback.
- **Kanban**: A visual workflow model that helps manage tasks using cards and columns to represent stages.

Project management tools such as **Jira**, **Trello**, **Asana**, and **Microsoft Project** provide platforms for collaborative planning, progress tracking, and resource coordination. These tools often represent projects as task networks or timelines, enabling managers to identify dependencies, optimize scheduling, and respond to changing requirements.

The application of project management principles spans diverse fields, including software development, construction, research, product design, and event planning. It is fundamental to ensuring that complex goals are achieved efficiently and effectively in organizational and team-based contexts.

4 Expressing Functional Data Structures as Graph-Relational Models

In this section, we demonstrate how each of the functional data structures—folders, documents, spreadsheets, and project plans—can be expressed within a graph-relational or relational model. For each case, we describe how the structure's elements map to nodes, edges, or tables, and how this enables flexible querying, linking, and reuse across applications.

4.1 Modeling Folder Structures with Graphs

A traditional folder structure, such as that used in file systems, exhibits a clear hierarchical organization: directories contain files and subdirectories, forming a tree rooted at a top-level directory. This hierarchical model can be naturally and efficiently represented using a graph-relational database.

In a graph model, both `folders` and `files` are represented as nodes. The parent-child relationship between folders and their contents is captured using directed edges, commonly labeled as `CONTAINS` or `HAS_CHILD`. This structure allows the folder hierarchy to be queried and navigated through graph traversal operations.

- Each **Folder** node may have attributes such as `name`, `created_at`, or `owner`.
- Each **File** node may contain metadata like `name`, `size`, `modified_at`, and `type`.
- A directed edge from a folder to a child folder or file encodes containment.

This model supports advanced file system features such as:

- **Symbolic links**, modeled as additional `LINKS_TO` edges.
- **Versioning**, represented as a chain of `PREVIOUS_VERSION_OF` relationships.
- **Cross-folder references**, using arbitrary edges beyond strict trees.

Graph-relational databases allow efficient recursive queries, such as listing all descendants of a folder or finding the full path to a file, which are typically more expensive in traditional relational databases.

Example Schema

- `(:Folder {name, created_at})`
- `(:File {name, size, modified_at})`
- `(:Folder)-[:CONTAINS]->(:Folder)`
- `(:Folder)-[:CONTAINS]->(:File)`

Example Query (Cypher)

Find all files contained within a given folder, including nested subfolders:

```
MATCH (root:Folder {name: "Projects"})-[:CONTAINS*]->(f:File)
RETURN f.name, f.size
```

This representation generalizes well beyond file systems. Any nested or categorized content—such as notes, tags, projects, or documents—can be modeled with the same graph structure, allowing flexible querying, refactoring, and visualization.

4.2 Modeling Hierarchical Documents as Graphs

Documents often exhibit a rich hierarchical structure, composed of nested units such as sections, subsections, paragraphs, sentences, and references. This structure can be naturally modeled as a graph, in which each unit of content is represented as a node, and containment or ordering relationships are captured as directed edges.

Graph-Based Representation

In a graph representation of a document:

- Each structural element (e.g., `Document`, `Section`, `Paragraph`, `Sentence`) is modeled as a distinct node.
- Containment relationships (e.g., `Section CONTAINS Paragraph`) are modeled as directed edges.
- Ordering relationships (e.g., `Sentence FOLLOWS Sentence`) can be represented with additional edges.
- Cross-references, footnotes, citations, and hyperlinks can be modeled as `LINKS_TO` edges between arbitrary nodes.

This graph-based approach captures both the structural and semantic aspects of documents, enabling flexible querying, version tracking, and reuse of content components.

Example Schema

- `(:Document {title, author})`
- `(:Section {heading, level})`
- `(:Paragraph {order})`
- `(:Sentence {text})`
- `(:Footnote {id, content})`

Example Relationships

- `(:Document)-[:HAS_SECTION]->(:Section)`
- `(:Section)-[:HAS_PARAGRAPH]->(:Paragraph)`
- `(:Paragraph)-[:HAS_SENTENCE]->(:Sentence)`
- `(:Sentence)-[:FOLLOWS]->(:Sentence)`
- `(:Sentence)-[:REFERENCES]->(:Footnote)`

Query Example (Cypher)

To retrieve all sentences in the "Introduction" section of a document:

```
MATCH (doc:Document {title: "Graph Theory in NLP"})
      -[:HAS_SECTION]->(s:Section {heading: "Introduction"})
      -[:HAS_PARAGRAPH]->(:Paragraph)-[:HAS_SENTENCE]->(sent:Sentence)
RETURN sent.text
```

Advantages

Modeling documents as graphs offers several advantages:

- **Flexible navigation**: Allows traversal from any node to any related unit (e.g., sentence to footnote, or paragraph to heading).
- **Content reuse**: Sections or components can be referenced or embedded in multiple documents via edges.
- **Versioning and annotation**: Individual nodes can be versioned or annotated without affecting the entire document.
- **Semantic queries**: Enables complex queries such as "find all paragraphs that reference section 2.1" or "list all sibling sections of the conclusion."

This model aligns closely with the needs of semantic publishing, knowledge management systems, and AI-assisted writing tools, where structural awareness and rich interlinking of content are essential.

4.3 Modeling Spreadsheets as Relational or Graphs

At a basic level, a spreadsheet can be naturally expressed using a relational database model. In both paradigms, data is organized into a two-dimensional grid of *rows* and *columns*, where:

- Each **column** in a spreadsheet corresponds to a column (or attribute) in a relational table.
- Each **row** corresponds to a record (or tuple) in that table.

For example, a spreadsheet with columns `Name`, `Age`, and `City` can be directly translated into a table with those same column names in a relational schema. Each row of the spreadsheet becomes a row in the corresponding database table.

This one-to-one mapping supports simple data import/export between spreadsheets and databases. Many database systems provide built-in tools to load data from spreadsheets (e.g., CSV or Excel files) directly into relational tables.

Extending to a Graph-Relational Model

While the relational model captures the tabular structure of spreadsheets, more complex features—such as formulas, cross-sheet references, and dynamic dependencies—can be expressed using a graph-relational model. In such cases:

- Cells may be represented as individual nodes.
- Dependencies between cells (e.g., formulas like `=A1 + B2`) can be modeled as directed edges.
- Sheets and named ranges can be modeled as higher-level nodes or grouping structures.

Nonetheless, for many applications, especially those treating spreadsheets as static tables of data, the direct row-column relational mapping remains both accurate and sufficient.

5 Modeling Project Management as Graphs

Project management involves the organization of goals, strategies, resources, and outcomes in a structured and trackable manner. A relational database provides a natural framework for modeling such systems, using tables to represent entities and foreign keys to define their relationships.

One effective way to represent project structures is to begin with high-level **goals**, each of which may be associated with multiple **options** or alternative plans. These options can be developed into concrete **plans**, which in turn specify their required **input conditions**, expected **outputs**, and any related constraints or resources.

Relational Schema Overview

- **Goal** table: Represents high-level objectives.
  - Columns: `goal_id`, `title`, `description`, `priority`
- **Option** table: Represents strategic options or pathways to achieve a goal.
  - Columns: `option_id`, `goal_id` (foreign key), `title`, `rationale`
- **Plan** table: Represents actionable project plans under each option.
  - Columns: `plan_id`, `option_id` (foreign key), `name`, `start_date`, `end_date`
- **InputCondition** table: Represents preconditions, resources, or requirements needed for a plan.
  - Columns: `input_id`, `plan_id` (foreign key), `type`, `description`

- **Output** table: Represents the expected deliverables or outcomes of a plan.
  - Columns: `output_id`, `plan_id` (foreign key), `name`, `status`

Query Examples

To retrieve all plans for a given goal:

```
SELECT p.*
FROM Plan p
JOIN Option o ON p.option_id = o.option_id
WHERE o.goal_id = ?
```

To list all inputs and outputs associated with a specific plan:

```
SELECT i.type, i.description, o.name, o.status
FROM Plan p
LEFT JOIN InputCondition i ON p.plan_id = i.plan_id
LEFT JOIN Output o ON p.plan_id = o.plan_id
WHERE p.plan_id = ?
```

Benefits of Relational Modeling

This relational structure allows project elements to be queried, tracked, and modified independently while preserving their relationships. It supports project visualization, version control, reporting, and integration with workflow engines or task-tracking tools.

The schema is also extensible: additional tables can model dependencies between plans, task breakdowns, responsible individuals, or progress metrics.

6 Conclusion

In this paper, we identified four core *functional data structures*—file systems, hierarchical documents, spreadsheets, and project plans—that play a foundational role in helping users organize, process, and act on information in everyday digital environments. These structures are "functional" in the sense that they are designed to support specific classes of cognitive and organizational tasks: storing and retrieving files, composing and navigating documents, computing with tabular data, and planning toward goals.

We showed that each of these structures can be formally represented within a unified framework using graph-relational or relational database models. By modeling entities as nodes and their relationships as typed edges—or as tables with foreign keys—we enable a consistent and expressive way to query, traverse, and extend diverse information systems. This graph-relational perspective not only clarifies the underlying structure of these functional systems, but also opens the door to new forms of integration, analysis, and tooling across domains that have traditionally been siloed.