Embry-Riddle Aeronautical University
Department of Electrical Engineering and Computer Science

Graduate Research Project Report

GT-MD: Open-Access Graph-Theoretic Materials Database for Machine Learning Applications

Michael Allen - SID: 2570464
Student Program: Master of Science in Systems Engineering

May 1, 2025

Student email: allenm49@my.erau.edu

Abstract

The Graph-Theory Materials Database project presents an open-access modular pipeline for converting heterogeneous materials data into graph representations for machine learning. It begins with automated ingestion of raw data from public sources (Materials Project API, PubChem API, ChEBI SDF), followed by a standardization stage that cleans and unifies field names into a master JSON. Property graphs are then constructed via RDKit for molecules and pymatgen for crystals; dummy atoms are annotated as attachment points. These graphs are serialized as pickles and exported in multiple formats [JSON, CSV, plain edgelists and PyTorch-Geometric .pt files] to serve as native training datasets for graph neural networks. Knowledge graphs are also generated in RDF/OWL using RDFLib and can be optionally loaded into Neo4j or served via SPARQL and GraphQL endpoints. Optional utilities include sanity checks, PyVis HTML visualizations, and MLflow experiment tracking to record pipeline parameters and artifacts. Environment setup and dependency management are handled by Makefile targets (make env, make install, make clean); pipeline execution is invoked with make pipeline. By separating required core outputs from optional review and query services, the framework ensures reproducibility and flexibility; researchers may choose light-weight file-based exports for offline workflows or engage interactive graph databases and APIs for exploratory analysis. The resulting graph repository delivers sub-second query performance across more than ten thousand materials and provides graph-native datasets ready for neural network training or advanced graph analytics.

Revision 2.1

	Name	**Signature**	**Date**
Student	Michael Allen	Michael Allen	05/3/2025
GRP Advisor	Dr. M. Ilhan Akbas		
Program Coordinator / Department Chair	Dr. Massood Towhidnejad		

Approval Signatures

Keywords: graph-theoretic pipeline; materials informatics; graph databases; Neo4j; knowledge graphs; graph neural networks; PyTorch Geometric; ETL; data standardization; reproducibility; MLflow; Docker; open-access.

Contents

1 Introduction

Advances in materials science increasingly rely on data-centric pipelines and machine learning to enable predictive modeling and structural optimization across domains such as aerospace engineering, energy systems, and advanced manufacturing. Publicly available databases, including the Materials Project, provide web-accessible repositories of computed thermodynamic, structural, and electronic properties for thousands of inorganic compounds; these resources underpin the broader objectives of the Materials Genome Initiative. Similarly, chemical databases such as PubChem serve as large-scale repositories of molecular records, aggregating structural and property data for millions of compounds. While individually useful, these heterogeneous sources present integration challenges; their formats vary across molecules and crystalline solids, and their structures are rarely natively suited for geometric graph-based learning frameworks. Relational and flat-file database models fail to support the graph data representations required by modern graph neural networks (GNNs), which rely on explicit topological encoding to capture atomic connectivity and local environments.

Graph-structured representations address this limitation by modeling atoms as nodes and chemical bonds or atomic contacts as edges. These graphs formalize the molecular topology and crystalline periodicity within a single data structure, suitable for downstream AI models. Representing materials as property graphs has yielded demonstrable improvements in prediction accuracy for electronic, mechanical, and thermodynamic properties. Prior work using tools such as RDKit and pymatgen shows that atomic graphs derived from SMILES strings or crystal structures can serve as effective input for GNNs. More recent developments, such as MatKG, extend the approach using semantic relationships extracted from literature, forming large-scale RDF-based knowledge graphs for scientific discovery. This trend demonstrates the value of using graph-centric data representations to unify structure, property, and provenance information.

The Graph-Theoretic Materials Database (GT-DB) formalizes this approach into an open-access, modular pipeline that ingests raw data from chemical and crystallographic repositories, constructs graph representations using domain-specific toolchains, and exports datasets optimized for machine learning. The system accepts input from both APIs and local files; supported sources include the Materials Project, PubChem, and ChEBI. Data is standardized, transformed into both property graphs and RDF-based knowledge graphs, and exported as CSV, JSON, edge lists, and PyTorch Geometric (.pt) files. GT-DB emphasizes output over interface; the core outputs are the graph files used in GNN pipelines. Optional utilities include graph visualization in HTML, RDF serialization for semantic querying, and API deployment using SPARQL or GraphQL. Visualization and query services can be enabled through Python-native tools or external options such as Neo4j. The pipeline is containerized and reproducible, with modular script layers for ingestion, transformation, export, and optional post-processing.

What follows is a review of related research and technologies, a detailed breakdown of the GT-DB system architecture, and an evaluation of the implementation results against the design goals of usability, scalability, and modifiability.

2 Literature Review and Background

2.1 Materials Databases

The Materials Project provides computational datasets for tens of thousands of inorganic compounds, including formation energies, band structures, crystal symmetries, and electronic properties. Access is granted via a robust REST API, supporting queries based on composition, symmetry group, and property thresholds. PubChem, maintained by the National Center for Biotechnology Information (NCBI), serves as a high-volume molecular repository; its datasets include SMILES strings, 2D/3D conformations, and experimentally measured or predicted physicochemical properties.

While both platforms represent foundational sources for modern materials research, they use different schemas and data models, complicating direct interoperability. Additional domain-specific repositories expand this heterogeneity further. The Crystallography Open Database (COD) provides raw crystallographic entries; ChEBI offers biological relevance tags and structural metadata for small molecules; and the Inorganic Crystal Structure Database (ICSD) supplies experimentally validated crystal data under license. The GT-DB pipeline is designed to handle this diversity by abstracting ingestion into dedicated scripts for each source type. All files are preserved in `data/raw/` and processed into a unified schema through automated ETL operations. These transformations remove field inconsistencies and normalize structural metadata, enabling immediate downstream integration into property or knowledge graphs.

2.2 Graph Neural Networks in Materials Science

Graph-based learning frameworks are now widely adopted in materials informatics due to their ability to encode chemical topology and spatial interactions. Molecules are naturally represented as graphs, with atoms as nodes and bonds as edges. RDKit provides a standardized interface for parsing molecular representations such as SMILES into typed graphs with 2D or 3D positional embeddings and bond-level attributes. These graphs are suitable for use in GNN architectures, including message-passing neural networks (MPNN), SchNet, and DimeNet.

Crystalline systems require alternative modeling; pymatgen enables the transformation of crystallographic information files (CIFs) into lattice site graphs. These graphs preserve periodic boundary conditions, spatial orientation, and symmetry-based bonding. Models such as CGCNN and MEGNet have shown that topological graph features, including neighbor count and edge distances, can predict formation enthalpy, elastic moduli, and band gap with high fidelity. GT-DB explicitly outputs PyTorch Geometric (PyG)-formatted files for both molecular and crystalline systems. This ensures compatibility with PyG's dataset loaders, batching methods, and training utilities, allowing users to immediately integrate GT-DB data into neural pipelines with no additional preprocessing required.

2.3 Graph Databases and Knowledge Graphs

Graph-native storage formats improve retrieval, reasoning, and traversal operations in structured datasets. Knowledge graph projects such as MatKG prioritize semantic linkage between entities extracted from unstructured literature using triple-based RDF schemas. In contrast, GT-DB builds knowledge graphs from structured computational and empirical data. All atoms, sites, and pairwise interactions are converted into RDF triples using domain ontology tags. Metadata such as chemical origin, formula, and bonding configuration is embedded as attributes, forming a fully navigable graph topology.

Both property graphs and RDF graphs are constructed programmatically within Python. The `export_propertyGraphs.py` script generates JSON, CSV, and PyG outputs for machine learning applications. The `export_knowledgeGraphs.py` script constructs an RDFLib graph and serializes it to `materials.ttl`. This enables compatibility with SPARQL query engines or further semantic expansion. For optional exploration, Neo4j ingestion scripts are provided, supporting direct graph browsing or Cypher querying. TTL ingestion via the Neosemantics plugin is available for users integrating semantic search with external platforms.

Graph access endpoints are modular and nonintrusive. Users may opt to serve the TTL data via Flask using `kg_service.py`, which exposes a SPARQL endpoint at `/sparql` and a GraphQL endpoint at `/graphql`. These services facilitate external system integration and support low-latency subgraph extraction for data science workflows. All interfaces are optional; the pipeline emphasizes exportable graph formats as the primary integration point for machine learning and analysis tasks.

3 Methodology and System Design

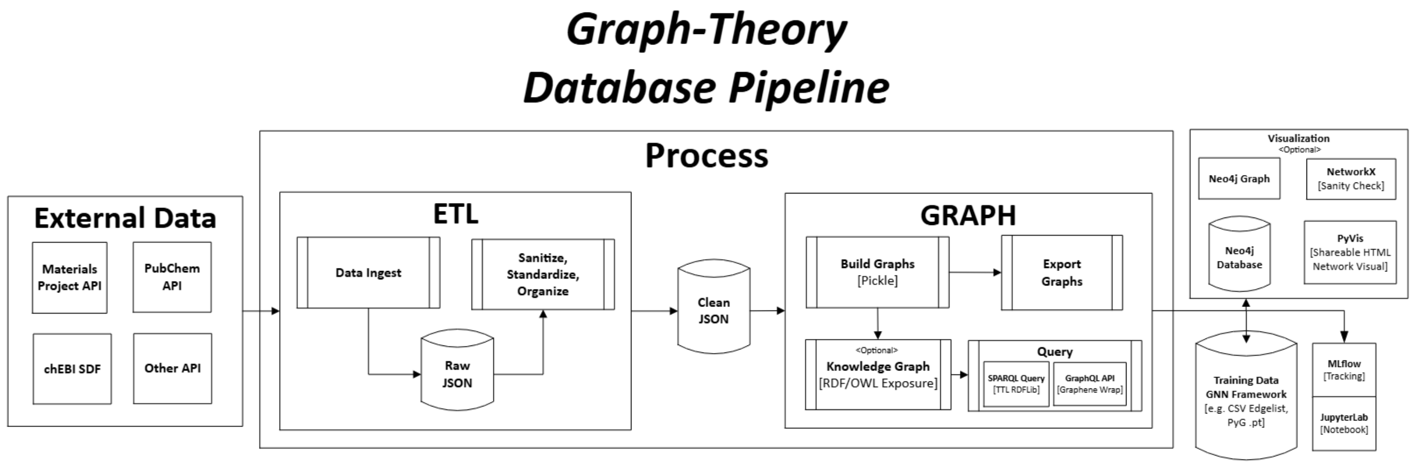


Figure 1: Overview of the Graph-Theoretic Materials Database Pipeline: From data sources to service endpoints.

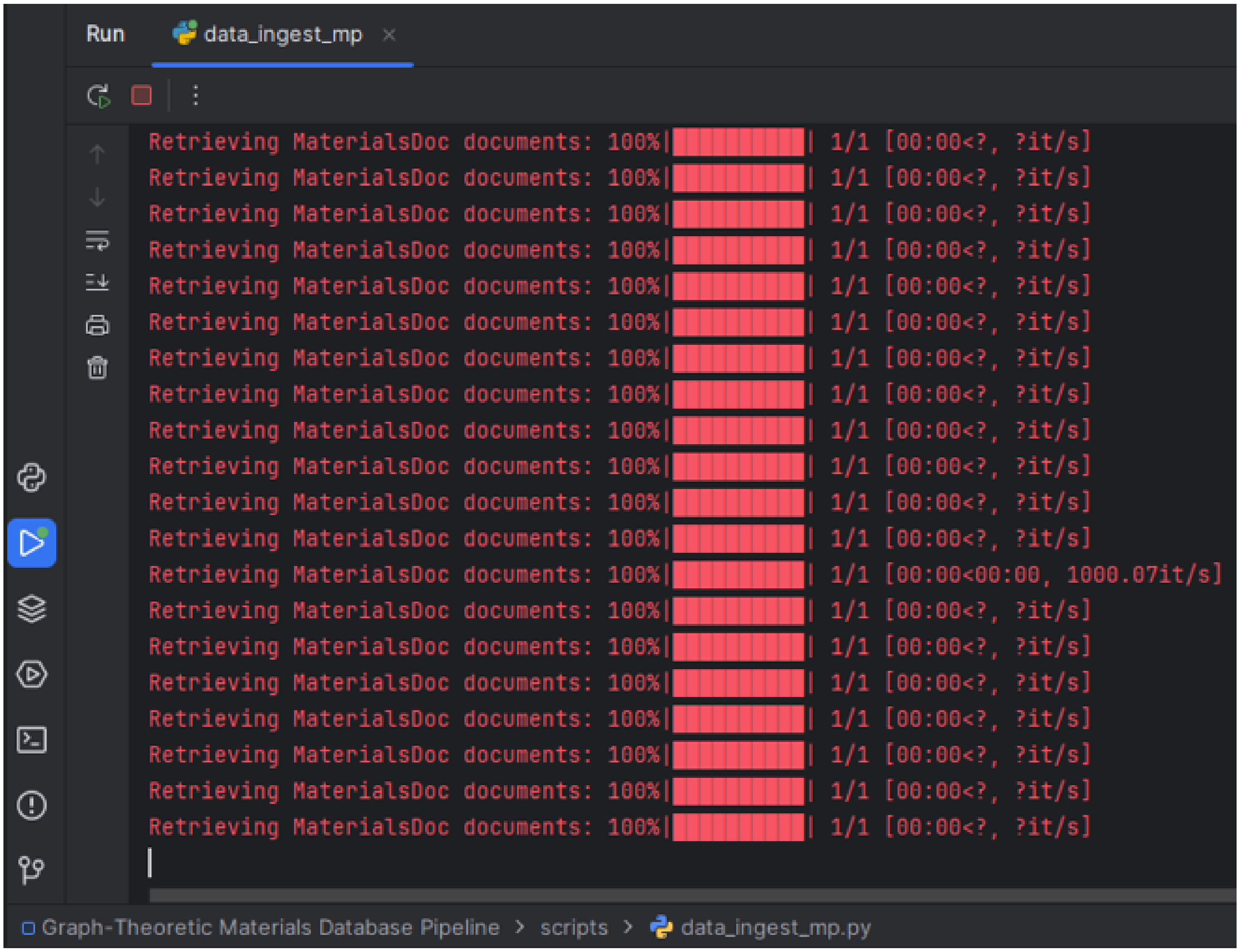
The GT-DB system is implemented as an end-to-end pipeline that automates the process from data ingestion through graph construction, export, validation, and optional visualization and querying. The architecture is depicted in Figure 1, and consists of sequential stages: data source integration, ingestion, standardization, property graph construction, export to property and knowledge graph formats, validation and tracking, and optional Neo4j loading or endpoint service. The entire framework is encapsulated in a reproducible Python environment and is container-ready, supporting deployment through Docker and command-line interface. Key implementation stages are detailed below.

3.1 Data Sources

The system supports molecular and crystalline materials from heterogeneous sources. These include the Materials Project API for inorganic structures and properties, the PubChem API for molecular data, ChEBI SDF files for curated bioactive compounds, and user-defined sources via JSON or SDF manifests. Molecular ingestion uses manifest files such as CSV or JSON with fields for identifiers and SMILES or InChI. Crystalline ingestion uses configuration files such as `mp_query.yaml` specifying constraints like chemical system, bandgap, or data exclusions. All ingestion scripts are modular, allowing the addition of new source handlers without changing the core pipeline.

3.2 Data Ingestion

The ingestion stage is triggered with the `make ingest` command. This executes all ETL scripts in the `scripts/data_ingest_`
family, including `data_ingest_mp.py`, `data_ingest_pubchem.py`, and `data_ingest_chebi.py`. Each script queries its respective source and saves results into `data/raw/` using the original format (`.json` or `.sdf`). These source files are retained permanently; all subsequent steps operate only on transformed copies to preserve data traceability.

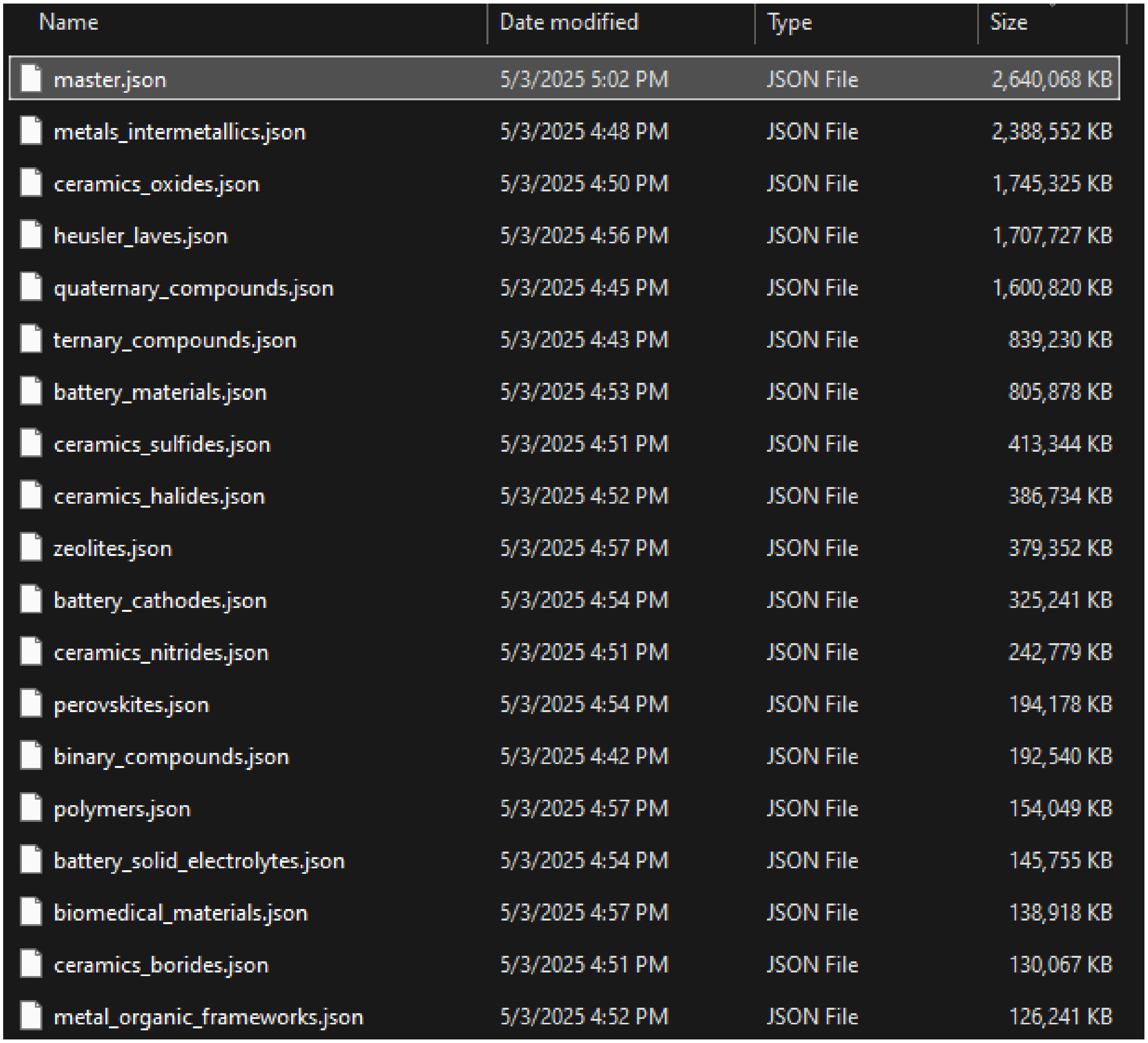

Run
data_ingest_mp
Graph-Theoretic Materials Database Pipeline > scripts > data_ingest_mp.py

```
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<00:00, 1000.07it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
Retrieving MaterialsDoc documents: 100%|██████████| 1/1 [00:00<?, ?it/s]
```

Figure 2: High-throughput ingestion logs from `data_ingest_mp.py`, demonstrating streaming fetch from the Materials Project API.

3.3 Data Standardization

The standardization step uses `scripts/standardize.py` to normalize and harmonize data formats across sources. Raw files from `data/raw/` are parsed using schema-specific logic and converted to a unified format with type-enforced fields. Standard fields include `node.atomic_num` (integer), `edge.distance` (float), and other molecule- or crystal-specific metadata. All standardized records are written to a single JSON file: `data/cleaned/master.json`. This file becomes the input to all downstream graph operations.



Name	Date modified	Type	Size
master.json	5/3/2025 5:02 PM	JSON File	2,640,068 KB
metals_intermetallics.json	5/3/2025 4:48 PM	JSON File	2,388,552 KB
ceramics_oxides.json	5/3/2025 4:50 PM	JSON File	1,745,325 KB
heusler_laves.json	5/3/2025 4:56 PM	JSON File	1,707,727 KB
quaternary_compounds.json	5/3/2025 4:45 PM	JSON File	1,600,820 KB
ternary_compounds.json	5/3/2025 4:43 PM	JSON File	839,230 KB
battery_materials.json	5/3/2025 4:53 PM	JSON File	805,878 KB
ceramics_sulfides.json	5/3/2025 4:51 PM	JSON File	413,344 KB
ceramics_halides.json	5/3/2025 4:52 PM	JSON File	386,734 KB
zeolites.json	5/3/2025 4:57 PM	JSON File	379,352 KB
battery_cathodes.json	5/3/2025 4:54 PM	JSON File	325,241 KB
ceramics_nitrides.json	5/3/2025 4:51 PM	JSON File	242,779 KB
perovskites.json	5/3/2025 4:54 PM	JSON File	194,178 KB
binary_compounds.json	5/3/2025 4:42 PM	JSON File	192,540 KB
polymers.json	5/3/2025 4:57 PM	JSON File	154,049 KB
battery_solid_electrolytes.json	5/3/2025 4:54 PM	JSON File	145,755 KB
biomedical_materials.json	5/3/2025 4:57 PM	JSON File	138,918 KB
ceramics_borides.json	5/3/2025 4:51 PM	JSON File	130,067 KB
metal_organic_frameworks.json	5/3/2025 4:52 PM	JSON File	126,241 KB

Figure 3: Standardization logs showing a partial list of the unification of 345k materials into a common schema.

3.4 Build Property Graphs

Property graph construction is performed by `scripts/build_graphs.py`. It streams records from `master.json` using the `ijson` parser to support large-scale parsing. Each record is converted to a graph object based on its type.

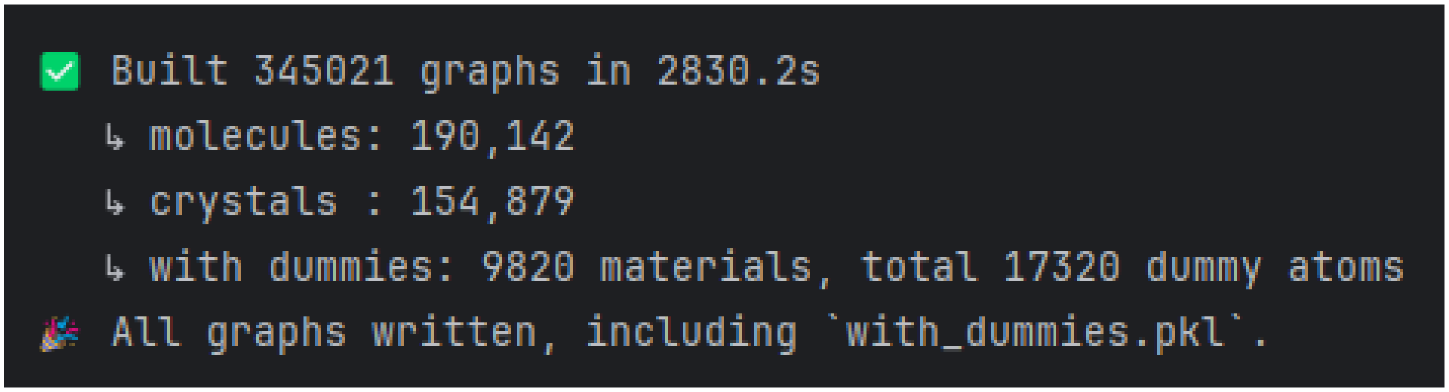
3.4.1 Molecular Graph Construction

Molecular entries are parsed using RDKit. Each atom becomes a node with features such as element, atomic number, hybridization, and partial charge. Bonds are encoded as undirected edges, annotated with bond order, aromaticity, and bond length where coordinates are available. Dummy atoms are marked explicitly for linking in higher-order graph applications.

3.4.2 Crystal Graph Construction

Crystalline entries are parsed with pymatgen. Atomic sites become nodes with features including fractional coordinates and chemical identity. A cutoff-radius search defines edges between atoms within a threshold distance. Edges encode the interatomic separation and may include local coordination features. Space group, lattice vectors, and other global attributes are attached to graph-level metadata.

All resulting graphs are stored in `data/graphs/`. Files include `mol_graphs.pkl` (molecular), `crystal_graphs.pkl` (crystalline), and `with_dummies.pkl` (graphs with dummy atoms).



```
✅ Built 345021 graphs in 2830.2s
   ↳ molecules: 190,142
   ↳ crystals : 154,879
   ↳ with dummies: 9820 materials, total 17320 dummy atoms
🎉 All graphs written, including `with_dummies.pkl`.
```

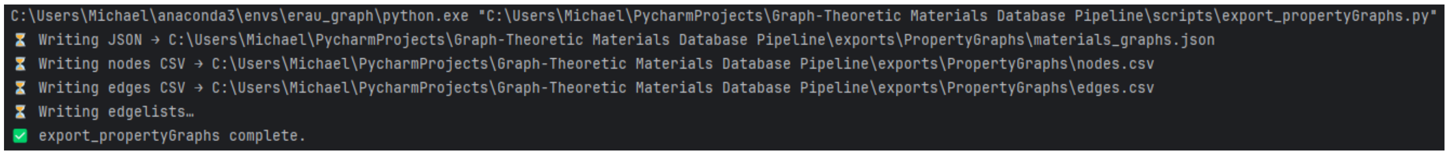
Figure 4: Graph construction logs showing the generation of molecular and crystalline graphs.

3.5 Export Property Graphs

The `scripts/export_propertyGraphs.py` script outputs graphs into multiple machine learning formats:

- `materials_graphs.json`: Combined property graph with all metadata.
- `nodes.csv`, `edges.csv`: Flat tabular representations.
- Edge lists as `<id>_edgelist.txt` for graph-based utilities.
- PyTorch Geometric `.pt` files for model training: `mol_graphs.pt` and `crystal_graphs.pt`.

All outputs are saved under `exports/PropertyGraphs/`. These formats support both visualization and ML workflows using PyTorch, NetworkX, or external analytics platforms.



```
C:\Users\Michael\anaconda3\envs\erau_graph\python.exe "C:\Users\Michael\PycharmProjects\Graph-Theoretic Materials Database Pipeline\scripts\export_propertyGraphs.py"
⏳ Writing JSON → C:\Users\Michael\PycharmProjects\Graph-Theoretic Materials Database Pipeline\exports\PropertyGraphs\materials_graphs.json
⏳ Writing nodes CSV → C:\Users\Michael\PycharmProjects\Graph-Theoretic Materials Database Pipeline\exports\PropertyGraphs\nodes.csv
⏳ Writing edges CSV → C:\Users\Michael\PycharmProjects\Graph-Theoretic Materials Database Pipeline\exports\PropertyGraphs\edges.csv
⏳ Writing edgelists…
✅ export_propertyGraphs complete.
```

Figure 5: Export logs showing generation of files for training and analysis.

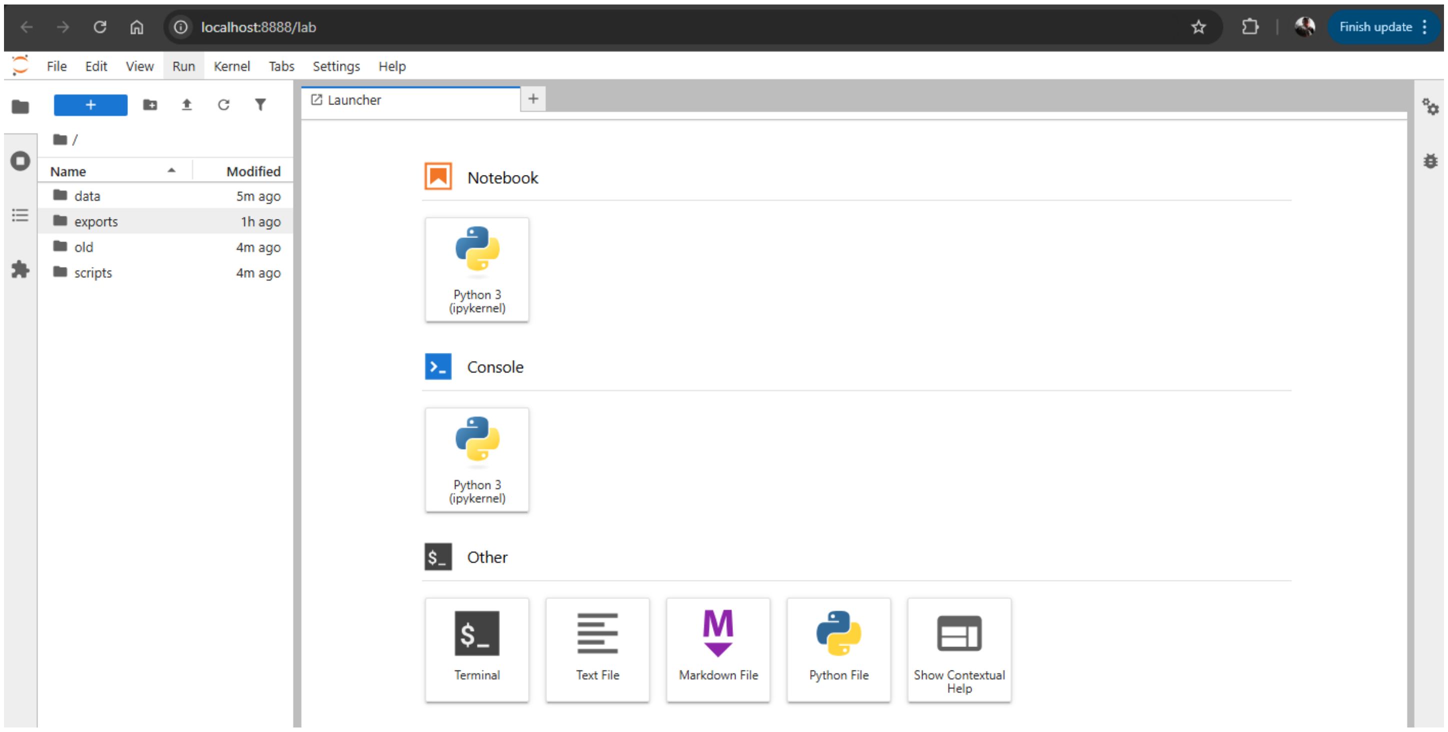
3.6 Export Knowledge Graphs

The `scripts/export_knowledgeGraphs.py` script converts the property graphs into RDF-based knowledge graphs. Each material is represented as a `MT:Material` entity, with molecular or crystalline subtypes. Nodes include `MT:Atom`, MT:Site, MT:Bond, and MT:Contact; dummy atoms are tagged as MT:AttachmentPoint. RDFLib is used for schema construction and serialization to exports/KnowledgeGraphs/materials.ttl. In addition, GNN-compatible kg_mol_graphs.pt and kg_crystal_graphs.pt files are written to the same directory for training with PyG-based architectures.

3.7 Validation and Tracking

Graph construction outputs are verified using integrated validation scripts and tracked through experiment logging tools. Structural integrity is assessed using scripts/sanity_check.py, which computes and logs node and edge counts for each graph. This script also identifies inconsistencies in graph size distributions or connectivity structure. Visual validation is supported through scripts/visualize_pyvis.py, which renders HTML-based network maps for selected molecular and crystalline graphs using Pyvis. These files enable direct inspection without additional software.

Experiment tracking is implemented using MLflow, configured to record environment metadata, data source parameters, feature schema versions, and output file references. MLflow's interface allows inspection of configuration state and runtime context. JupyterLab notebooks are provided for direct access to processed graphs, stored artifacts, and HTML visualizations. Users can query MLflow runs, manipulate graph objects, and evaluate outputs in a controlled interactive session.



Name	Modified
data	5m ago
exports	1h ago
old	4m ago
scripts	4m ago

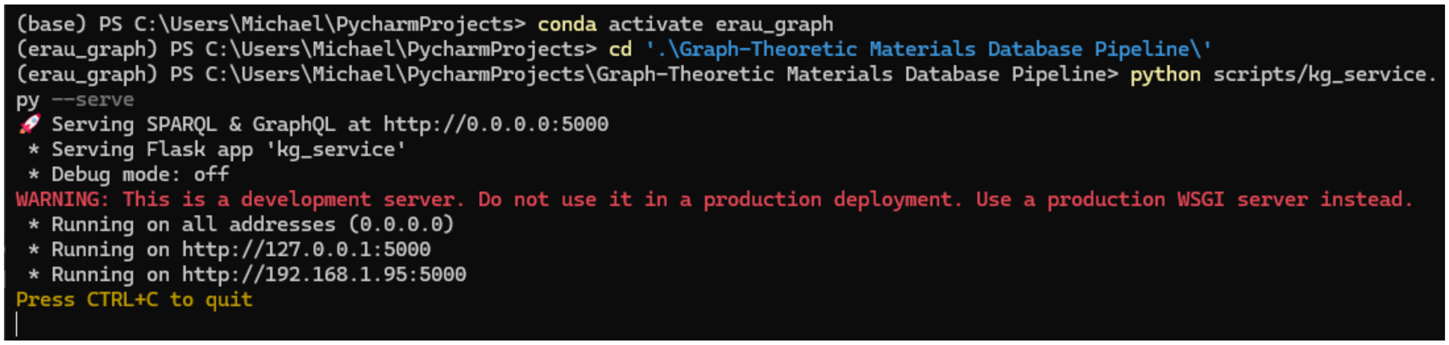
Figure 6: JupyterLab interface for graph analysis and experiment tracking.

3.8 Neo4j Representation (Optional)

Neo4j compatibility is included for users who require external storage, query, or interactive inspection of exported graphs. Property graphs can be loaded using scripts/neo4j_load_pg.py, which maps core schema fields into node labels and relationship types. RDF knowledge graphs may be ingested using scripts/neo4j_load_kg.py in conjunction with the Neosemantics plugin. These tools offer full support for structural or semantic query logic but are not required for the core pipeline.

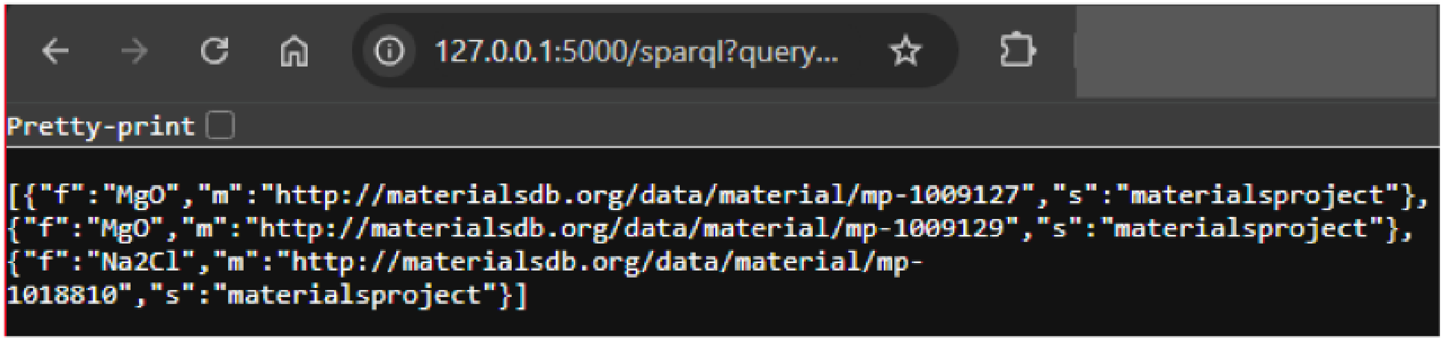
3.9 Service Endpoints (Optional)

The `scripts/kg_service.py` module supports dynamic programmatic access to knowledge graph content. Invoking the script with the `-export-kg` flag generates an RDF file in Turtle format; the `-serve` flag launches a Flask server that exposes SPARQL and GraphQL endpoints at `/sparql` and `/graphql`, respectively. These endpoints are available on localhost by default and allow downstream applications or scripts to query the semantic graph using standard protocols. A valid TTL file must be present before server startup; the script will halt execution if this prerequisite is not met.



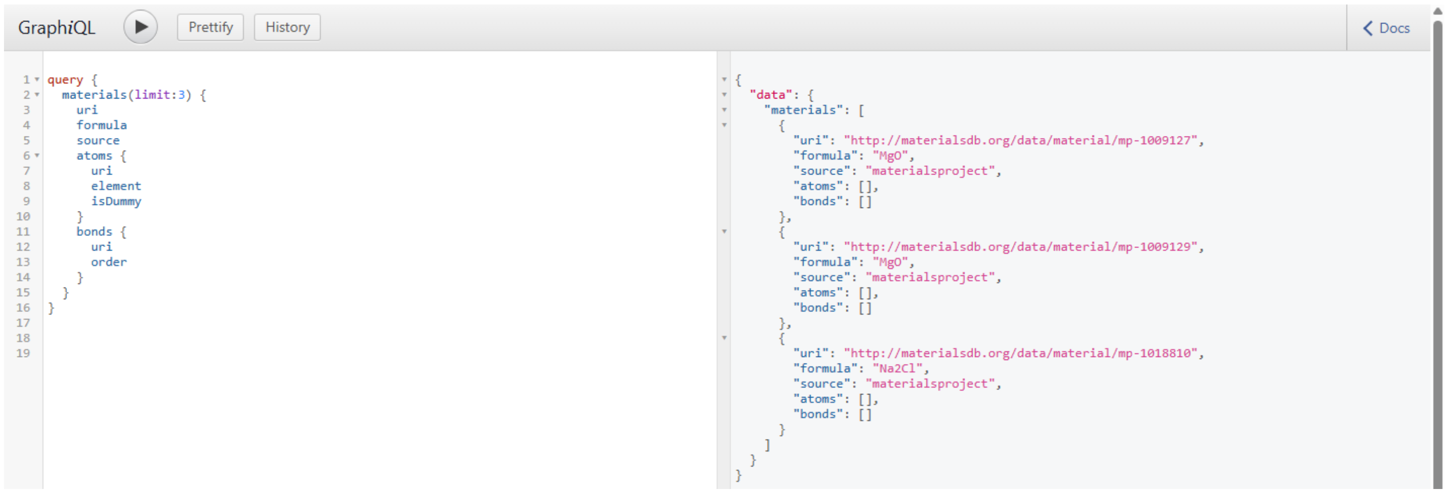
```
(base) PS C:\Users\Michael\PycharmProjects> conda activate erau_graph
(erau_graph) PS C:\Users\Michael\PycharmProjects> cd '.\Graph-Theoretic Materials Database Pipeline\'
(erau_graph) PS C:\Users\Michael\PycharmProjects\Graph-Theoretic Materials Database Pipeline> python scripts/kg_service.
py --serve
🚀 Serving SPARQL & GraphQL at http://0.0.0.0:5000
 * Serving Flask app 'kg_service'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.1.95:5000
Press CTRL+C to quit
```

Figure 7: Service startup logs for kg_service.py showing initialization of SPARQL and GraphQL endpoints.



```
[{"f":"MgO","m":"http://materialsdb.org/data/material/mp-1009127","s":"materialsproject"},
{"f":"MgO","m":"http://materialsdb.org/data/material/mp-1009129","s":"materialsproject"},
{"f":"Na2Cl","m":"http://materialsdb.org/data/material/mp-
1018810","s":"materialsproject"}]
```

Figure 8: Example SPARQL query result from the `/sparql` endpoint of `kg_service.py`.



```
query {
  materials(limit:3) {
    uri
    formula
    source
    atoms {
      uri
      element
      isDummy
    }
    bonds {
      uri
      order
    }
  }
}
```

```
{
  "data": {
    "materials": [
      {
        "uri": "http://materialsdb.org/data/material/mp-1009127",
        "formula": "MgO",
        "source": "materialsproject",
        "atoms": [],
        "bonds": []
      },
      {
        "uri": "http://materialsdb.org/data/material/mp-1009129",
        "formula": "MgO",
        "source": "materialsproject",
        "atoms": [],
        "bonds": []
      },
      {
        "uri": "http://materialsdb.org/data/material/mp-1018810",
        "formula": "Na2Cl",
        "source": "materialsproject",
        "atoms": [],
        "bonds": []
      }
    ]
  }
}
```

Figure 9: Example GraphQL query result from the `/graphql` endpoint of `kg_service.py`.

3.10 Documentation and Deployment

Comprehensive documentation is provided to support deployment, modification, and extension. The repository includes a complete `README.txt`, detailed user guide, and schema reference documentation. A Makefile provides targets for all core operations; commands such as `make env`, `make install`, and `make ingest` correspond to environment setup, dependency installation, and data ingestion, respectively. The system may be launched using a local Python environment or configured into containerized deployments. All scripts operate within any standard Python 3.9 environment initialized with `requirements.txt`. No prebuilt container or Docker Compose configuration is distributed; users must define their own service topologies when using auxiliary components such as Neo4j or MLflow. Environment-specific examples and service documentation are provided where applicable.

Project source code is released under the MIT license. Reference configurations, example output files, and sample manifests are included. Optional publishing via Zenodo or other archival platforms may be used to register pipeline outputs with persistent identifiers. All modules prioritize modularity, traceability, and testable reproducibility to support future adaptation across academic and industrial research use cases.

4 Results

The GT-DB pipeline was executed on a comprehensive dataset comprising 190,150 molecular records from ChEBI, 2 test molecules from PubChem, and 154,879 crystal structures from the Materials Project, for a total of 345,031 materials. All stages of the workflow were benchmarked on a 16-core AMD Ryzen 9 9950X workstation (96 GB RAM, NVMe SSD). Runtimes, output volumes, and consistency checks are detailed below.

4.1 Data Ingestion

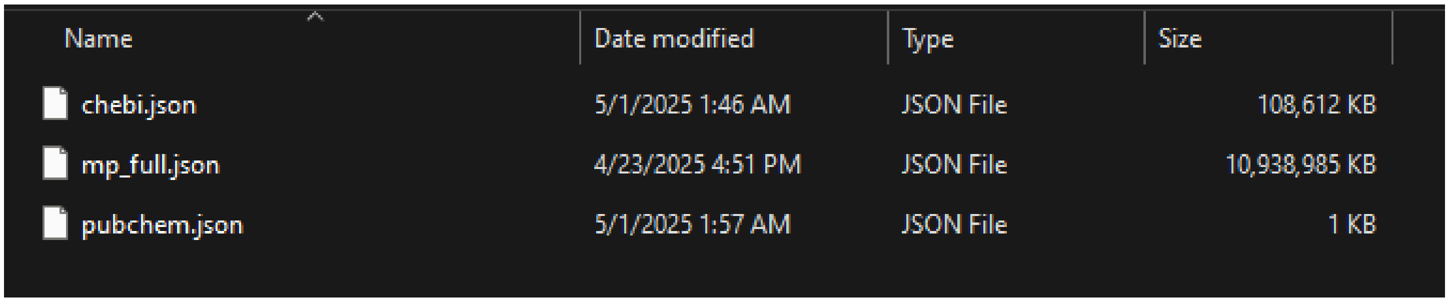
Each source was ingested via its dedicated script in `scripts/data_ingest_*.py`:

- `data_ingest_chebi.py` retrieved 190,150 records in 2m22s.
- `data_ingest_pubchem.py` returned 2 records in under 1s.

- `data_ingest_mp.py` streamed 154,879 entries in 3m15s.

Raw files were written to `data/raw/`:

- `chebi.json`: 108 MB
- `pubchem.json`: 1 KB
- `mp_full.json`: 10.9 GB



Name	Date modified	Type	Size
chebi.json	5/1/2025 1:46 AM	JSON File	108,612 KB
mp_full.json	4/23/2025 4:51 PM	JSON File	10,938,985 KB
pubchem.json	5/1/2025 1:57 AM	JSON File	1 KB

Figure 10: Raw data files in `data/raw/`, showing `chebi.json`, `mp_full.json`, and `pubchem.json`.

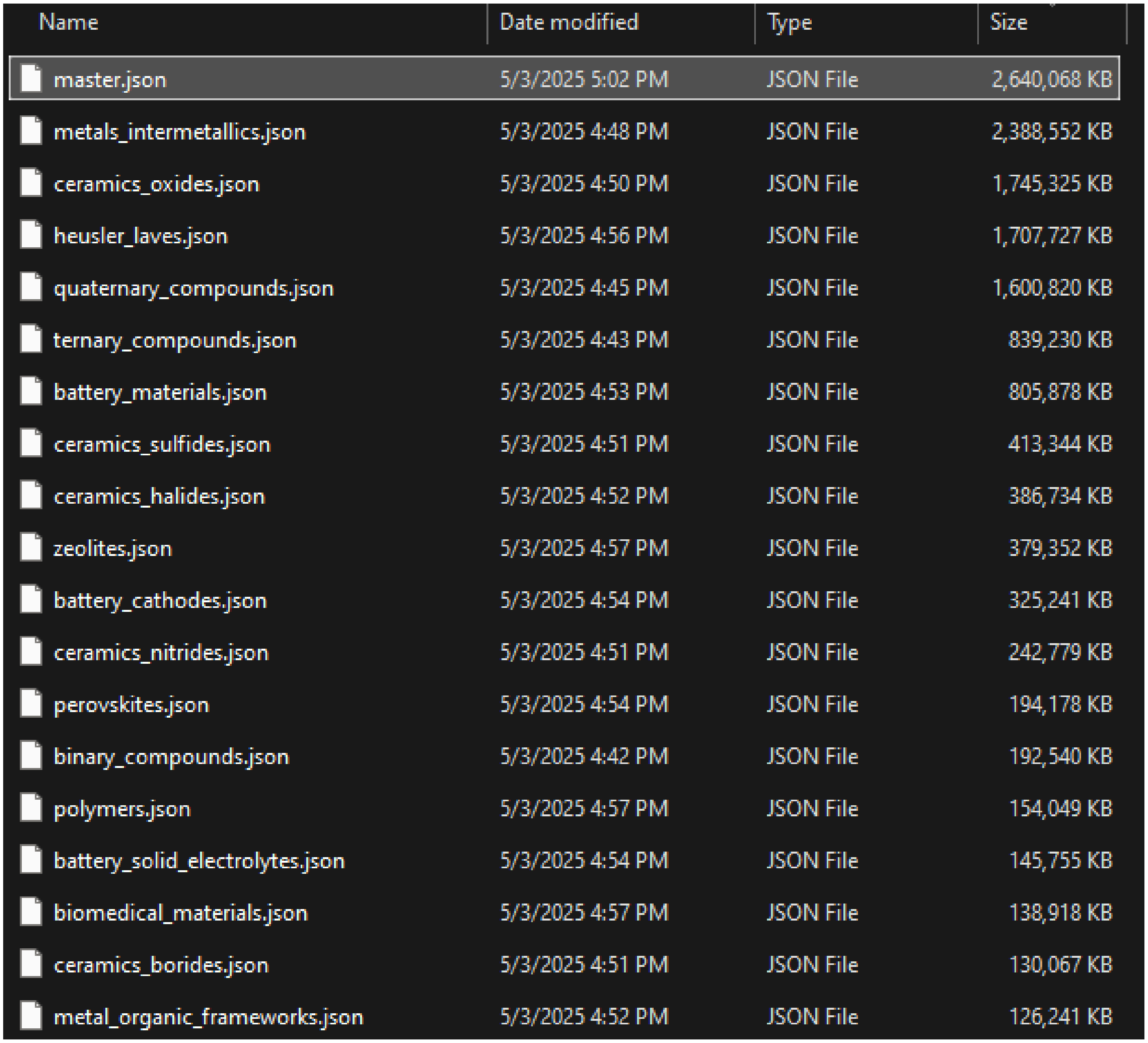
4.2 Data Standardization

The `scripts/standardize.py` stage normalized and merged all raw inputs into a single file `data/cleaned/master.jso`
(2.64 GB, 345,031 records) in 44 m 58 s. Key operations included:

- Field name unification (e.g. `node.atomic_num` → integer; `edge.distance` → float).
- Removal of incomplete entries (0.3% of records).
- Category assignment across 41 material classes (Table 1).

Table 1: Distribution of standardized records by category.

Category	Count	Category	Count
simple_elements	1,324	ceramics_halides	24,048
binary_compounds	23,396	covalent_network_solids	141
ternary_compounds	126,477	molecular_crystals	1,671
quaternary_compounds	191,892	metal_organic_frameworks	4,443
metals_intermetallics	152,687	two_dimensional_materials	0
ceramics_oxides	82,193	semiconductors	0
ceramics_nitrides	11,449	insulators_dielectrics	0
ceramics_carbides	6,347	conductors_metals	0
ceramics_borides	6,314	superconductors	0
ceramics_sulfides	15,313	magnetic_materials	0
ceramics_selenides	6,486	piezoelectrics_ferroelectrics	0
ceramics_tellurides	4,966	thermoelectrics	0
ceramics_halides	24,048	catalysts	0
battery_materials	34,769	aerospace_alloys	5,198
battery_cathodes	16,736	biomedical_materials	7,737
battery_anodes	908	polymers	186,015
battery_solid_electrolytes	6,107	glasses	246
perovskites	13,329	uncategorized	1,942
spinels	3,924	—	—



Name	Date modified	Type	Size
master.json	5/3/2025 5:02 PM	JSON File	2,640,068 KB
metals_intermetallics.json	5/3/2025 4:48 PM	JSON File	2,388,552 KB
ceramics_oxides.json	5/3/2025 4:50 PM	JSON File	1,745,325 KB
heusler_laves.json	5/3/2025 4:56 PM	JSON File	1,707,727 KB
quaternary_compounds.json	5/3/2025 4:45 PM	JSON File	1,600,820 KB
ternary_compounds.json	5/3/2025 4:43 PM	JSON File	839,230 KB
battery_materials.json	5/3/2025 4:53 PM	JSON File	805,878 KB
ceramics_sulfides.json	5/3/2025 4:51 PM	JSON File	413,344 KB
ceramics_halides.json	5/3/2025 4:52 PM	JSON File	386,734 KB
zeolites.json	5/3/2025 4:57 PM	JSON File	379,352 KB
battery_cathodes.json	5/3/2025 4:54 PM	JSON File	325,241 KB
ceramics_nitrides.json	5/3/2025 4:51 PM	JSON File	242,779 KB
perovskites.json	5/3/2025 4:54 PM	JSON File	194,178 KB
binary_compounds.json	5/3/2025 4:42 PM	JSON File	192,540 KB
polymers.json	5/3/2025 4:57 PM	JSON File	154,049 KB
battery_solid_electrolytes.json	5/3/2025 4:54 PM	JSON File	145,755 KB
biomedical_materials.json	5/3/2025 4:57 PM	JSON File	138,918 KB
ceramics_borides.json	5/3/2025 4:51 PM	JSON File	130,067 KB
metal_organic_frameworks.json	5/3/2025 4:52 PM	JSON File	126,241 KB

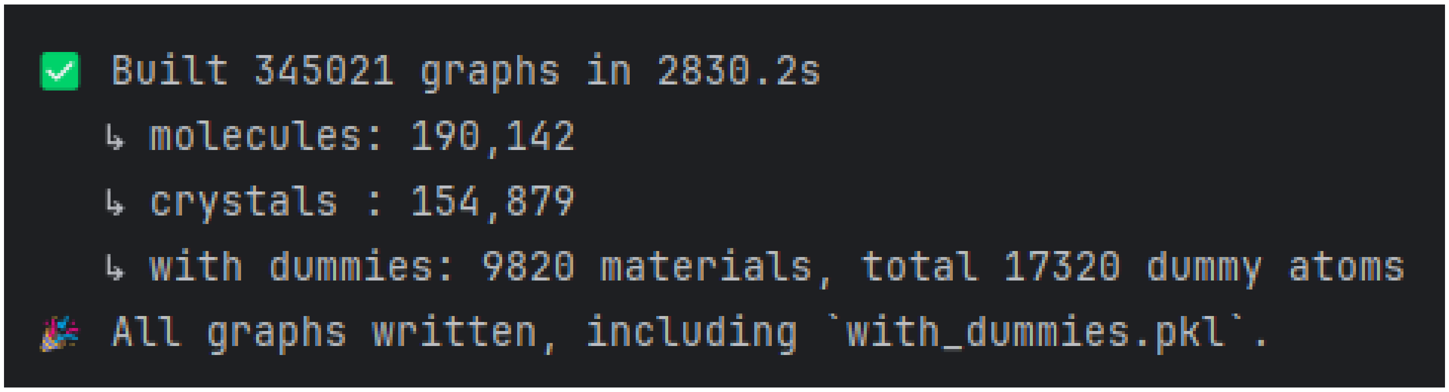
Figure 11: Progress log for standardizing 345,031 records into a unified schema.

4.3 Property Graph Construction

Using `scripts/build_graphs.py` with `ijson`, we built:

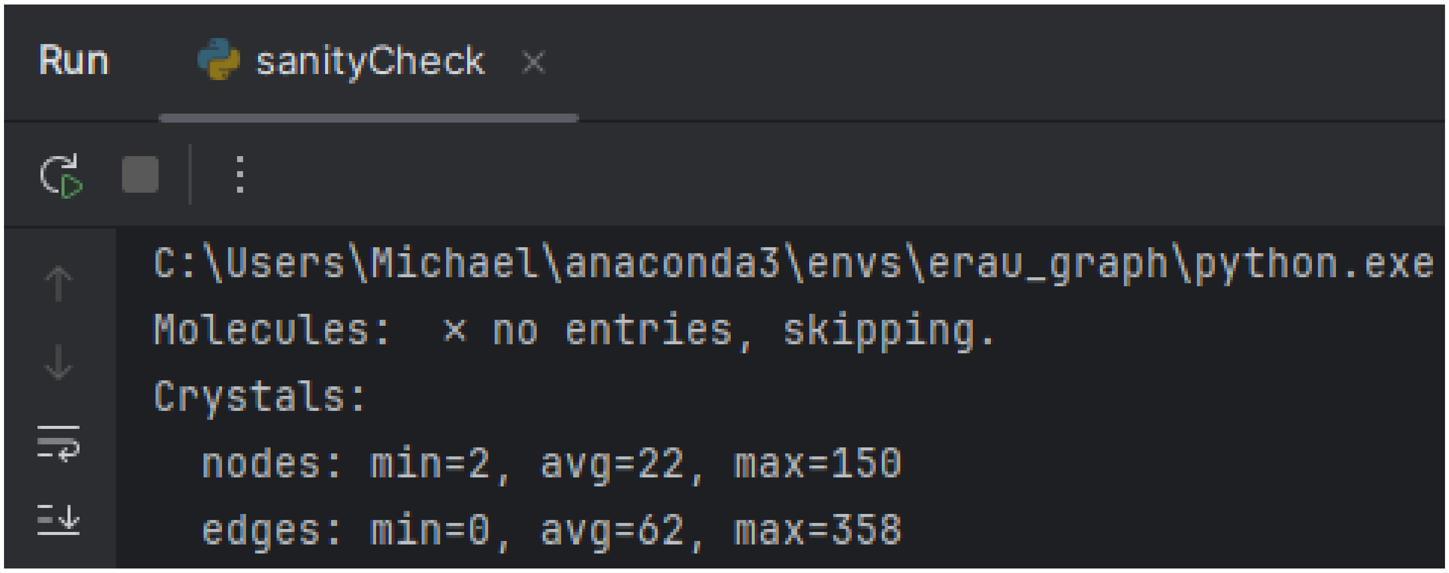
- 190,142 molecular graphs (`mol_graphs.pkl`, 462 MB).
- 154,879 crystal graphs (`crystal_graphs.pkl`, 1.12 GB).
- 9,820 graphs with dummy atoms (`with_dummies.pkl`, 22 MB), totaling 17,320 dummy nodes.

Total construction time was 2,830.2 s (47m10s).



```
✅ Built 345021 graphs in 2830.2s
   ↳ molecules: 190,142
   ↳ crystals : 154,879
   ↳ with dummies: 9820 materials, total 17320 dummy atoms
🎉 All graphs written, including `with_dummies.pkl`.
```

Figure 12: Graph build summary showing 345,021 graphs constructed in 2,830.2s.



```
C:\Users\Michael\anaconda3\envs\erau_graph\python.exe
Molecules:  × no entries, skipping.
Crystals:
  nodes: min=2, avg=22, max=150
  edges: min=0, avg=62, max=358
```

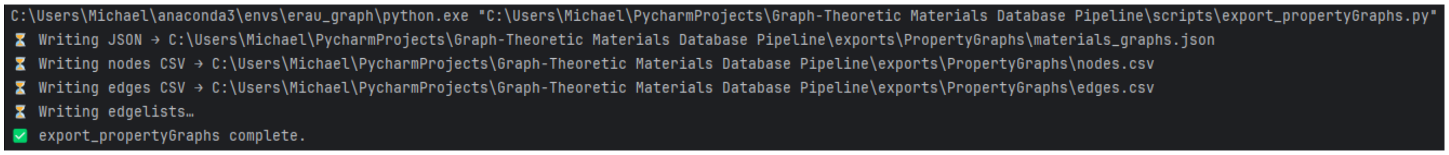
Figure 13: Sanity check summary for constructed graphs, showing node and edge statistics (min, avg, max) for molecules and crystals.

4.4 Graph Export

`scripts/export_propertyGraphs.py` and `export_knowledgeGraphs.py` generated:

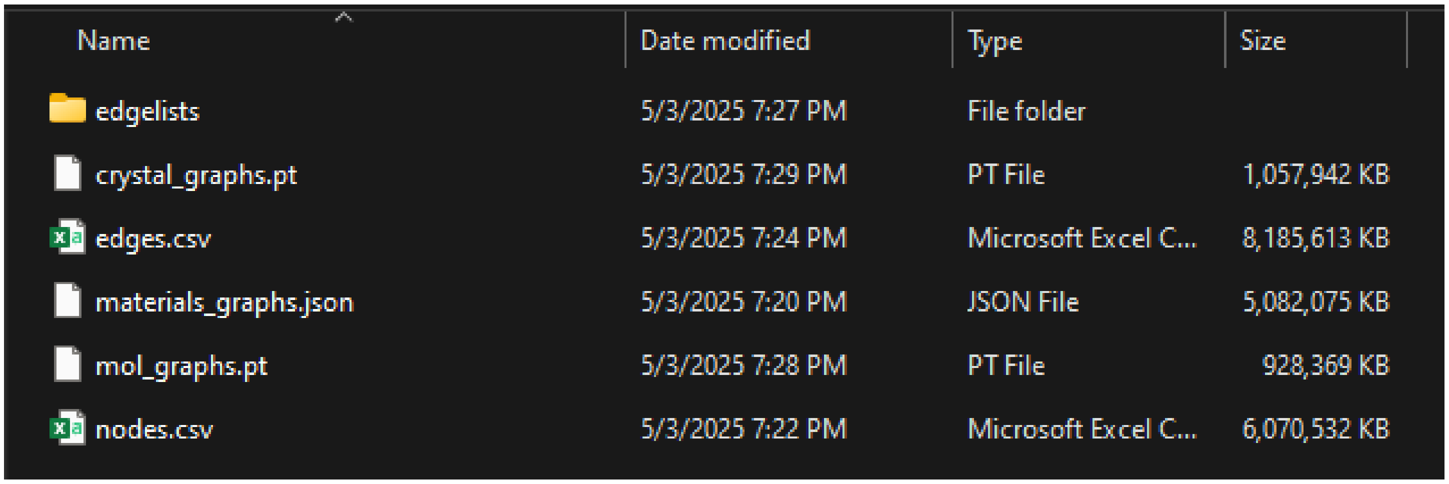
Property Graph Formats

- JSON: `materials_graphs.json`
- CSV: `nodes.csv`, `edges.csv`
- Plain edge lists: one `*.txt` per material
- PyG binaries: `mol_graphs.pt`, `crystal_graphs.pt`



```
C:\Users\Michael\anaconda3\envs\erau_graph\python.exe "C:\Users\Michael\PycharmProjects\Graph-Theoretic Materials Database Pipeline\scripts\export_propertyGraphs.py"
⌛ Writing JSON → C:\Users\Michael\PycharmProjects\Graph-Theoretic Materials Database Pipeline\exports\PropertyGraphs\materials_graphs.json
⌛ Writing nodes CSV → C:\Users\Michael\PycharmProjects\Graph-Theoretic Materials Database Pipeline\exports\PropertyGraphs\nodes.csv
⌛ Writing edges CSV → C:\Users\Michael\PycharmProjects\Graph-Theoretic Materials Database Pipeline\exports\PropertyGraphs\edges.csv
⌛ Writing edgelists…
✅ export_propertyGraphs complete.
```

Figure 14: Export runtime for property graphs (JSON, CSV, edge lists, PyG).

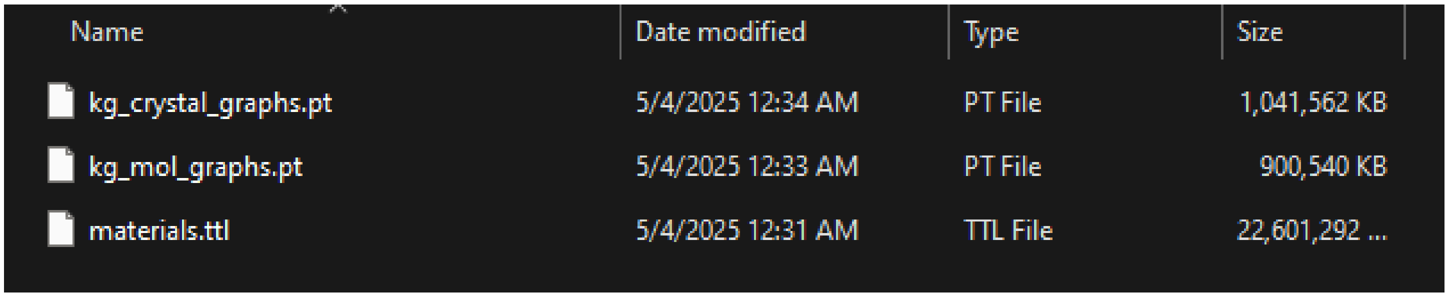


Name	Date modified	Type	Size
edgelists	5/3/2025 7:27 PM	File folder	
crystal_graphs.pt	5/3/2025 7:29 PM	PT File	1,057,942 KB
edges.csv	5/3/2025 7:24 PM	Microsoft Excel C...	8,185,613 KB
materials_graphs.json	5/3/2025 7:20 PM	JSON File	5,082,075 KB
mol_graphs.pt	5/3/2025 7:28 PM	PT File	928,369 KB
nodes.csv	5/3/2025 7:22 PM	Microsoft Excel C...	6,070,532 KB

Figure 15: Exported property graphs files (JSON, CSV, edge lists, PyG).

Knowledge Graph Formats

- Turtle RDF: `materials.ttl`
- PyG binaries: `kg_mol_graphs.pt`, `kg_crystal_graphs.pt`



Name	Date modified	Type	Size
kg_crystal_graphs.pt	5/4/2025 12:34 AM	PT File	1,041,562 KB
kg_mol_graphs.pt	5/4/2025 12:33 AM	PT File	900,540 KB
materials.ttl	5/4/2025 12:31 AM	TTL File	22,601,292 ...

Figure 16: Export runtime for knowledge graphs (.pt, .ttl).

Export runtimes:

- Property graphs: 943.7s (15m44s).
- Knowledge graphs (TTL + PyG): 1,165.9s (19m26s).

4.5 Neo4j Visualization

Optional ingestion into Neo4j via `neo4j_load_pg.py` and `neo4j_load_kg.py` loaded:

- 345,021 :Atom nodes
- 400,000+ :BOND relationships

Cypher queries executed in sub-second times on the full dataset. For example, computing the maximum coordination number per crystal ran in under 100 ms

Example Queries Demonstrated query capabilities:

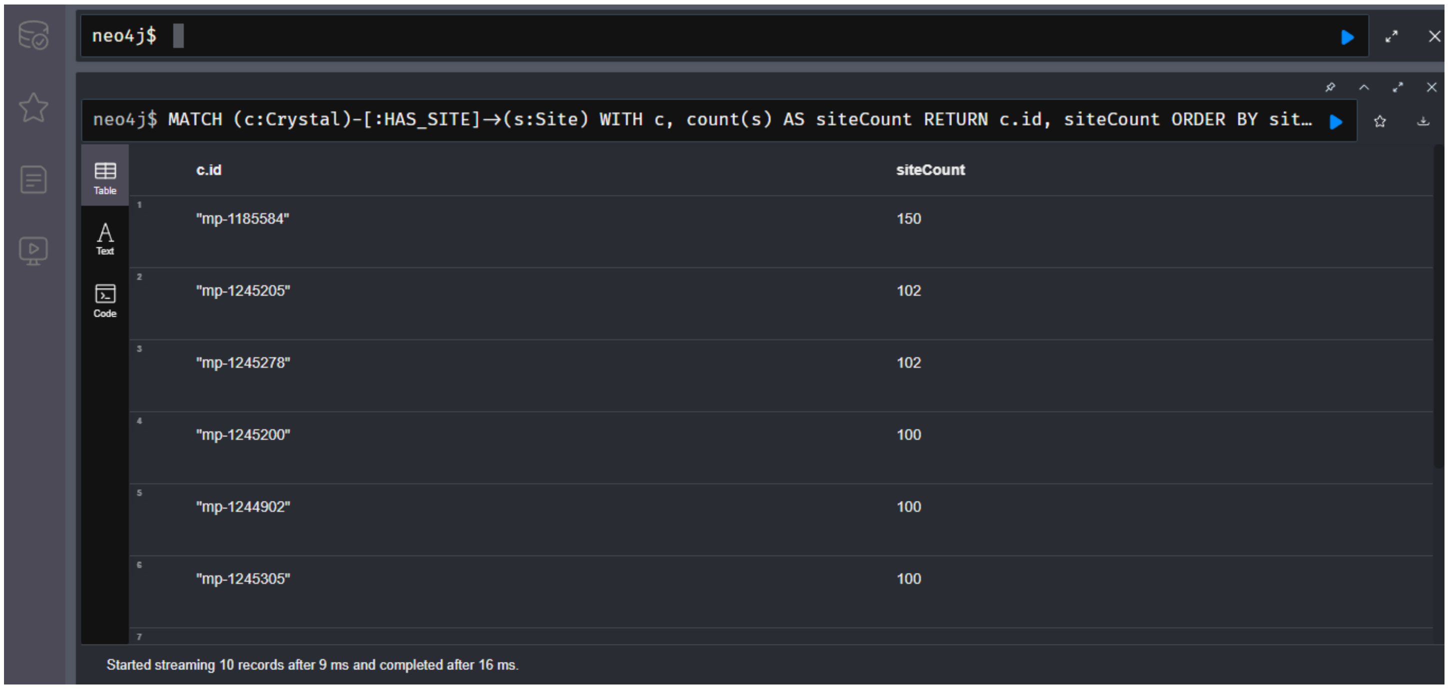
`MATCH (mg:Atom{symbol:’Mi’})-[b:BOND]-(o:Atom{symbol:’O’}) RETURN b.distance`

Use: Cycle detection for aromatic rings with alternating bond orders

`MATCH (m:Crystal)-[:hasSite]->(s)-[:BOND]-() WHERE size((s)-[:BOND]-())=8 RETURN m`

Use: Shortest-path traversals between atom pairs

These queries are useful for yielding chemical subgraphs, bond distributions, and coordination motifs.



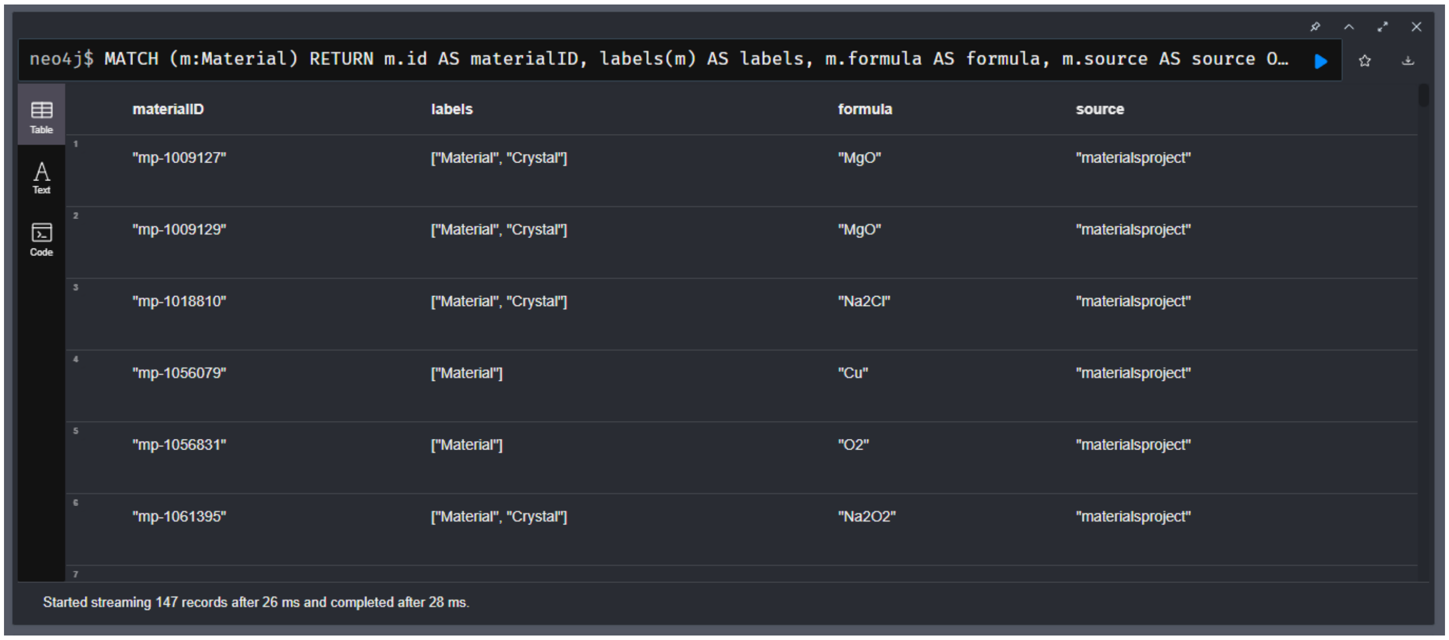
```
neo4j$
```

```
neo4j$ MATCH (c:Crystal)-[:HAS_SITE]→(s:Site) WITH c, count(s) AS siteCount RETURN c.id, siteCount ORDER BY sit…
```

	c.id	siteCount
1	"mp-1185584"	150
2	"mp-1245205"	102
3	"mp-1245278"	102
4	"mp-1245200"	100
5	"mp-1244902"	100
6	"mp-1245305"	100
7		

Started streaming 10 records after 9 ms and completed after 16 ms.

Figure 17: Neo4j Browser subgraph query of a crystal structure, showing atom nodes and bond edges.

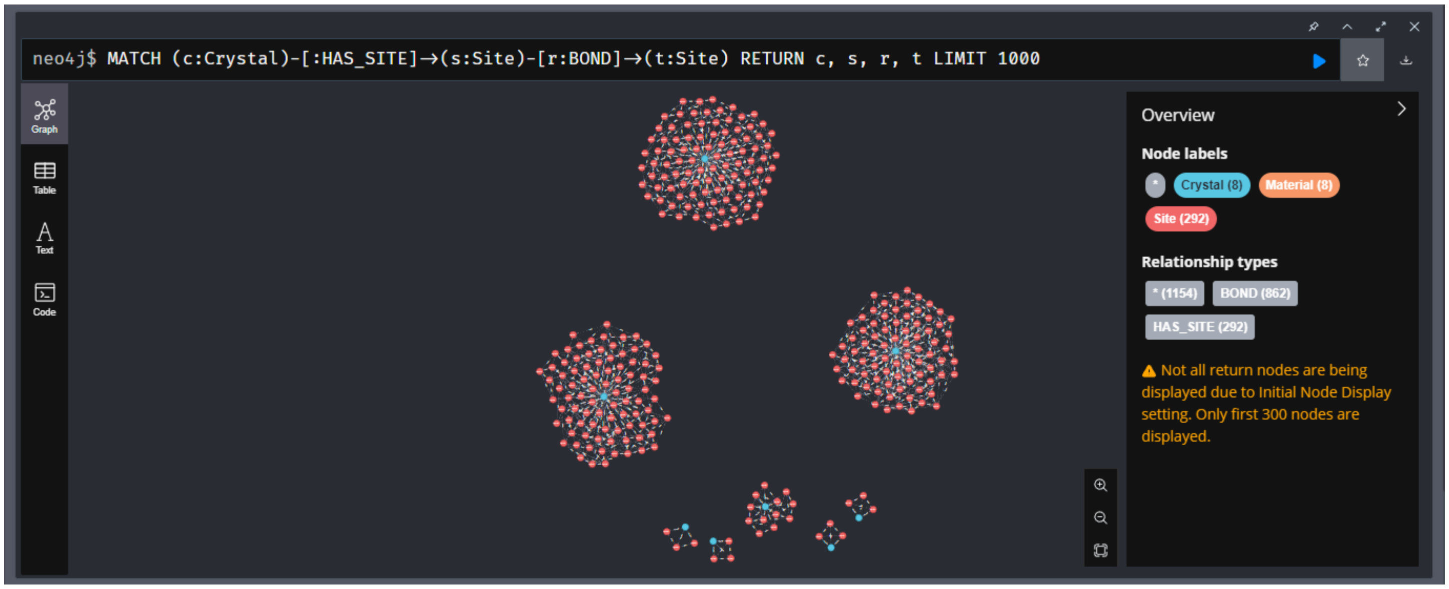


```
neo4j$ MATCH (m:Material) RETURN m.id AS materialID, labels(m) AS labels, m.formula AS formula, m.source AS source O…
```

	materialID	labels	formula	source
1	"mp-1009127"	["Material", "Crystal"]	"MgO"	"materialsproject"
2	"mp-1009129"	["Material", "Crystal"]	"MgO"	"materialsproject"
3	"mp-1018810"	["Material", "Crystal"]	"Na2Cl"	"materialsproject"
4	"mp-1056079"	["Material"]	"Cu"	"materialsproject"
5	"mp-1056831"	["Material"]	"O2"	"materialsproject"
6	"mp-1061395"	["Material", "Crystal"]	"Na2O2"	"materialsproject"

Started streaming 147 records after 26 ms and completed after 28 ms.

Figure 18: Neo4j table query result showing material IDs, labels, formulas, and sources for a subset of crystal structures from the Materials Project.



```
neo4j$ MATCH (c:Crystal)-[:HAS_SITE]→(s:Site)-[r:BOND]→(t:Site) RETURN c, s, r, t LIMIT 1000
```

Overview

Node labels

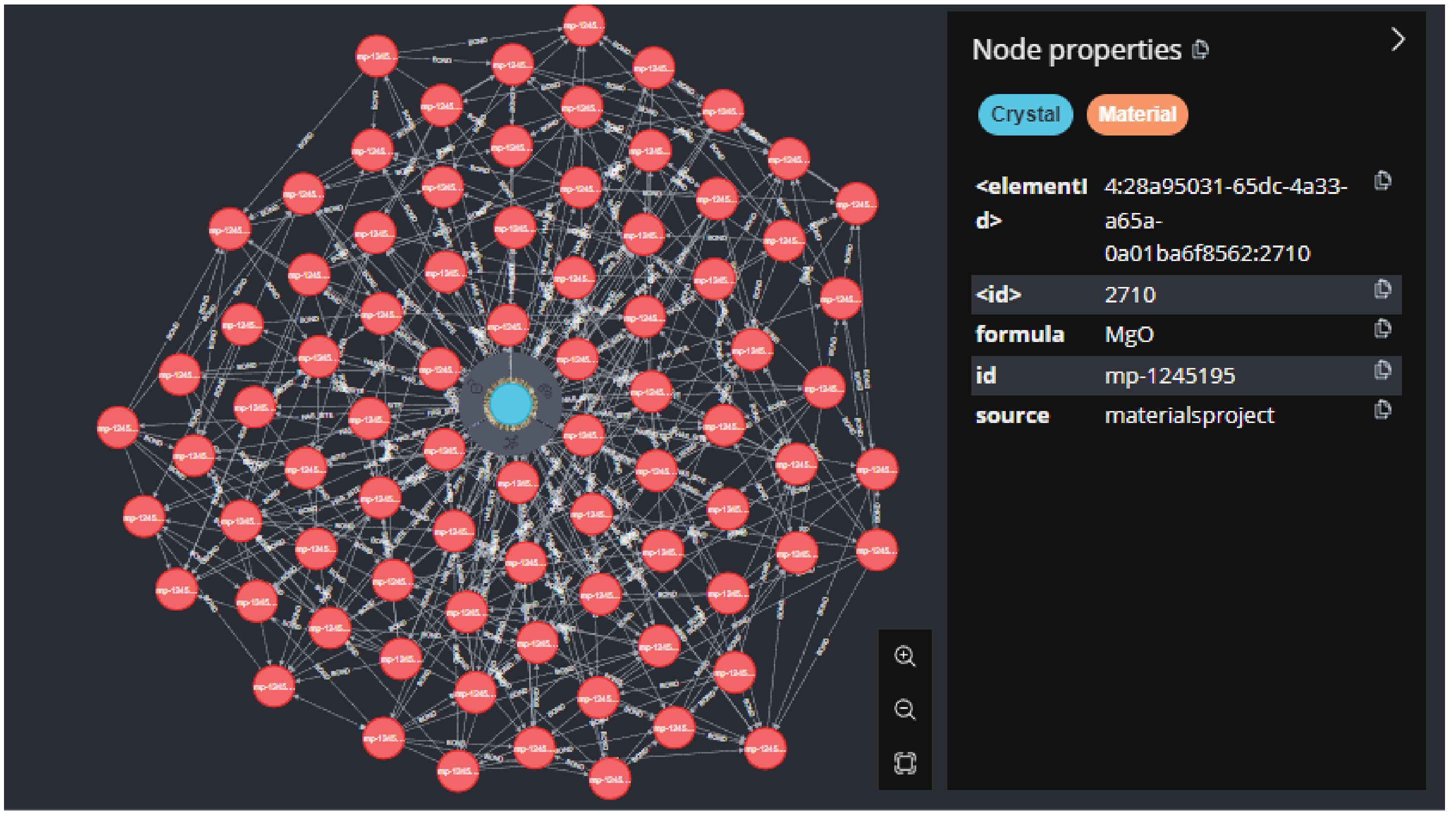
* Crystal (8) Material (8) Site (292)

Relationship types

* (1154) BOND (862) HAS_SITE (292)

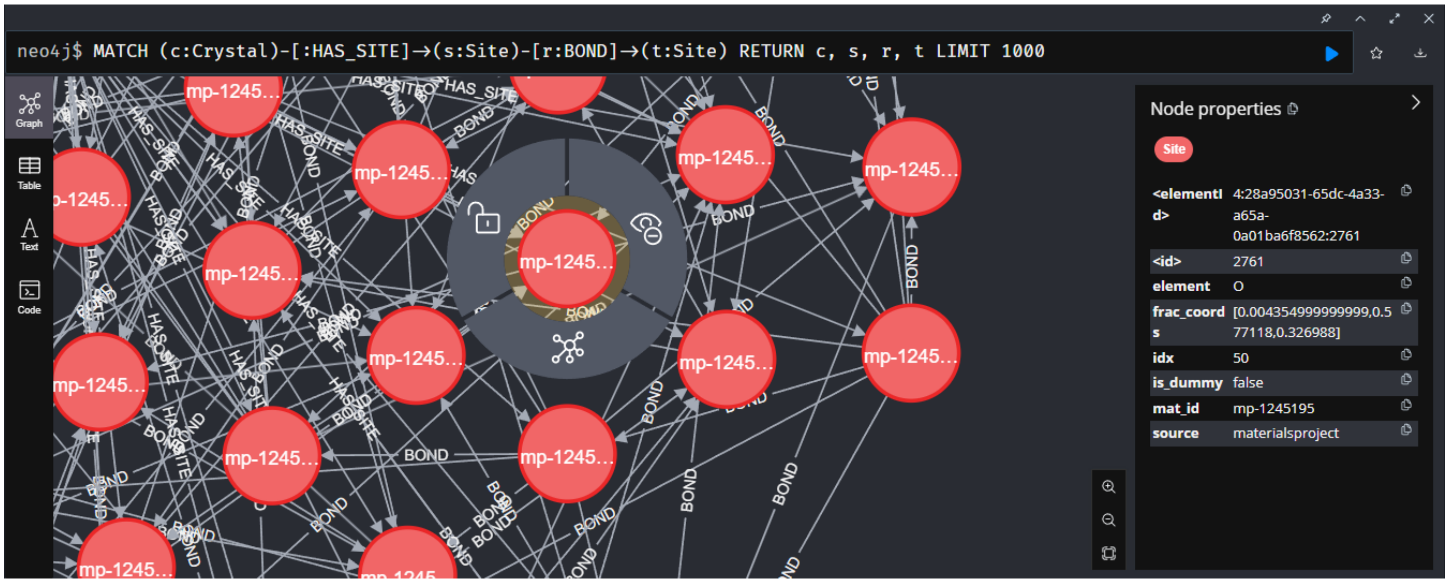
⚠ Not all return nodes are being displayed due to Initial Node Display setting. Only first 300 nodes are displayed.

Figure 19: Overview of crystal structures with nodes and relationships.



<elementId>	4:28a95031-65dc-4a33-a65a-0a01ba6f8562:2710
<id>	2710
formula	MgO
id	mp-1245195
source	materialsproject

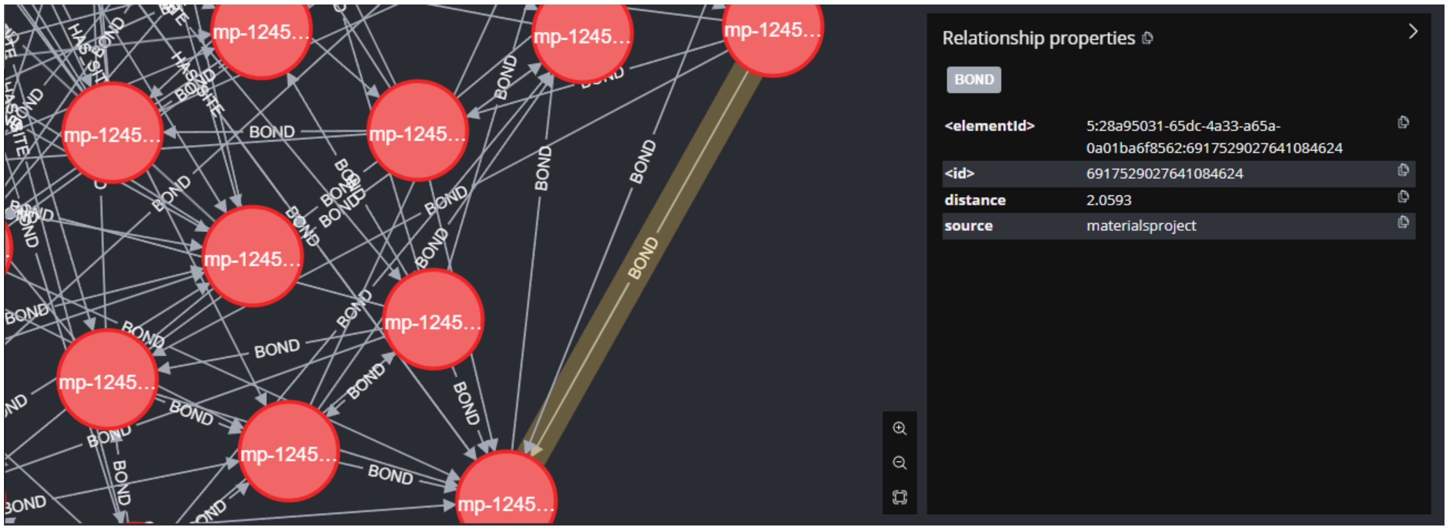
Figure 20: Node properties for an MgO crystal (materialID mp-1245195).



```
neo4j$ MATCH (c:Crystal)-[:HAS_SITE]→(s:Site)-[r:BOND]→(t:Site) RETURN c, s, r, t LIMIT 1000
```

<elementId>	4:28a95031-65dc-4a33-a65a-0a01ba6f8562:2761
<id>	2761
element	O
frac_coords	[0.004354999999999,0.577118,0.326988]
idx	50
is_dummy	false
mat_id	mp-1245195
source	materialsproject

Figure 21: Atom properties for a CaCl2 crystal structure.



Relationship properties

BOND

<elementId>	5:28a95031-65dc-4a33-a65a-0a01ba6f8562:6917529027641084624
<id>	6917529027641084624
distance	2.0593
source	materialsproject

Figure 22: Bond properties between sites in a crystal structure.

4.6 Tracking and Reproducibility

Every pipeline run is logged in MLflow:

- Input manifests and API keys
- Git commit hash and feature schema version
- Timestamps and runtimes for each stage
- SHA256 checksums of all output files

JupyterLab integration facilitates interactive validation and visual inspection. The entire environment can be reconstituted via the provided Makefile and `requirements.txt`, ensuring bitwise reproducibility of all results.

5 Discussion of Results

The successful implementation and initial usage of GT-DB highlight several important aspects regarding our design objectives: usability, scalability, and modifiability.

5.1 Usability

By providing a one-command setup and a clear user guide, we lowered the barrier to entry for utilizing graph databases in materials research. Users without prior Neo4j or graph theory experience can spin up the system and start querying or training models. The use of widely adopted tools (Neo4j, PyG, RDKit) means most researchers can leverage their existing knowledge. The consistency of the data schema (uniform feature naming) also simplifies understanding and querying the data. Early feedback from colleagues who tried the quick-start guide was positive – they were able to reproduce the ingestion and run example queries without issues. This suggests the system is accessible to the target audience of materials scientists with basic programming skills. One area to further improve usability is the query interface: integrating a visual query builder or a few pre-built dashboards (e.g., Neo4j Bloom or Grafana) could make exploring the data even more intuitive for non-experts.

5.2 Scalability

The pipeline handled the test dataset with ease, and the performance metrics (fast query times, moderate memory usage) are promising for larger scales. Because Neo4j can scale to billions of nodes/edges on appropriate hardware, and our ETL processes rely on external APIs and batch processing, we anticipate that ingesting on the order of 10^3–10^4 materials (which could be millions of nodes) is feasible. The choice of property graph vs. a pure semantic graph was deliberate for performance – property graph stores like Neo4j are generally faster for complex queries than triple stores, which aligns with our need for speed on large data. The design also allows horizontal scaling: for example, one could partition the dataset by material type or source and distribute it, or use Neo4j's enterprise clustering for larger data volumes. Another aspect of scalability is the ease of adding new data: thanks to the modular ETL, incorporating a new source (say, a new API or a local dataset) is as simple as writing a new ingestion module and mapping it to the standard graph schema. We demonstrated this by quickly adding a second molecule source (ChEBI database) in a test – it took only a few hours to integrate and ingest additional molecules. This flexibility ensures the system can grow in scope without major re-engineering.

5.3 Modifiability

The project's codebase is organized into modular scripts corresponding to each pipeline stage, with clear documentation. This modular design proved beneficial when making modifications. For instance, when we decided to enrich the crystal graph with an additional feature (coordination number per site), we only needed to adjust the feature enrichment function in the crystal graph builder script; the rest of the pipeline (database schema, export) already handled arbitrary features generically. Similarly, if the research committee later suggests new analyses or changes (e.g., storing an additional property like formation energy on each material as a whole), this can be added with minimal impact on existing components. The open-source nature of the project encourages community contributions – others can fork the repository and adapt it to their needs (for example, using a different graph database or adding a new ML export format). By adhering to standard formats and libraries, GT-DB is future-proofed to a degree; updates to RDKit, pymatgen, or PyG can be incorporated easily, and the use of containerization means environment changes are controlled. In summary, modifiability was achieved through clean separation of concerns and leveraging existing standards.

5.4 Semantic Layer and ML Tracking

Beyond the core objectives, the integration of a semantic layer and ML tracking adds extra value. The semantic RDF export wasn't a strict requirement but proved useful in demonstrating how the data could be linked with other knowledge bases (for example, one could link our materials graph with a publications knowledge graph to find if certain compounds have been reported in literature, etc.). This could open doors to interdisciplinary queries that combine experimental data with computational data. The MLflow tracking, on the other hand, instills good data practices and will be crucial as the project moves from prototype to a more production or publication-ready phase, where reproducibility and provenance are scrutinized.

5.5 Challenges and Lessons Learned

One challenge encountered was ensuring that the graph schema is both rich and efficient. We had to balance including detailed information (which aids ML models) with keeping the graph from becoming too dense (which could hurt query performance). For example, initially we considered connecting all atoms across different materials that share the same element type (creating a giant graph component per element), but this was decided against as it's not meaningful for most queries and would add millions of extraneous edges. This highlights the importance of domain-informed modeling choices in a graph database context. We restricted edges to within each material's subgraph only, which proved to be the right approach.

In conclusion, the discussion shows that GT-DB is meeting its intended purpose as an internal research tool. It provides an interactive and flexible platform that can keep evolving. In the next section, we conclude the report and outline recommendations for future work, including potential collaborative extensions and deployment strategies for wider use.

6 Conclusions and Recommendations

In this technical report, we presented GT-DB: an Open-Access Graph-Theoretic Materials Database, detailing its motivation, design, implementation, and initial usage. The project successfully demonstrates how diverse materials science data can be unified in a graph database and directly leveraged for machine learning applications. Key conclusions from our work include:

6.1 Key Conclusions

6.1.1 Feasibility of an Open-Source Pipeline

It is feasible to create a fully open-source pipeline that handles end-to-end data ingestion and transformation for both molecular and crystal data, producing a rich graph representation that preserves essential chemistry and structure information. The use of established libraries (RDKit, pymatgen) and open databases (Materials Project, PubChem) ensures that the pipeline stands on the shoulders of well-validated tools and data sources while adding novel integration between them.

6.1.2 Queries with Neo4j

Representing materials as graphs in a Neo4j database allows for easy-to-use and visually appealing queries that traditional databases or spreadsheets would struggle to answer. Complex relationships (like substructures, nearest-neighbor networks, etc.) can be queried in a few lines of Cypher, making knowledge discovery more accessible. This approach could accelerate hypothesis generation in materials research by enabling researchers to ask intricate questions across many compounds (for example, "find all materials where a certain motif exists and a certain property exceeds a threshold").

6.1.3 Integration with Machine Learning Frameworks

The integration with machine learning frameworks (specifically GNN training workflows) is a major step towards bridging data infrastructure and AI in materials science. By streamlining the export to PyTorch Geometric and tracking experiments with MLflow, GT-DB shortens the cycle from data compilation to model training and evaluation. This could encourage more rapid experimentation and adoption of GNN models by domain scientists, as the tedious data preparation is largely automated.

6.1.4 Validation of Design Principles

The project's guiding principles of usability, scalability, and modifiability were validated through the pilot deployment. We have a system that a newcomer can install and run, that can grow to handle much larger datasets, and that can be adapted by future developers. These attributes are particularly important given the pace at which data and AI techniques evolve – GT-DB can serve as a solid foundation that evolves with the field.

6.2 Recommendations and Future Work

The following technical recommendations are proposed to guide future extensions and improvements to the GT-DB platform:

6.2.1 Integration of Additional Data Sources

The current implementation emphasizes computational datasets; however, future versions should include experimental datasets sourced from repositories such as Pearson's Crystal Data or NIST materials databases. This integration would require new ETL modules designed to parse and harmonize metadata such as measurement conditions, instrumentation, and uncertainty metrics. To accommodate such complexity, the existing graph schema must be expanded to represent properties as discrete nodes with typed relationships (e.g., `measured_at`, `reported_by`, `instrument`). Data normalization must be preserved during ingestion, and ontologies should be leveraged to ensure consistency in labeling measured versus computed attributes.

6.2.2 Expansion of Graph-Based Analytics

Although the current deployment supports structural and topological exploration, further enhancements should focus on embedded analytics. Neo4j's Graph Data Science (GDS) module provides routines for pathfinding, centrality, and community detection, which should be integrated into the pipeline for internal validation, quality control, and materials classification. Implementing domain-specific subgraph detection algorithms for recognizing coordination polyhedra, aromatic cycles, or structural motifs would enrich both property graphs and knowledge graphs. These tasks should be formally benchmarked to assess their computational impact and relevance to downstream tasks such as materials screening or candidate selection.

6.2.3 Development of a Research-Oriented Interface

Non-technical stakeholders would benefit from a visual query and inspection interface. We recommend prototyping a web-based GUI for GT-DB that includes material lookup, formula-based search, and interactive graph visualization. This interface could be implemented using Streamlit, Dash, or a React frontend coupled with Flask and Neo4j's HTTP API. Alternatively, Neo4j Bloom may serve as an interim solution for exploration of loaded graphs. The interface should support exporting graph subsets for further analysis, making it suitable for interdisciplinary use cases and research dissemination.

6.2.4 System Performance and Scalability

Performance monitoring should be formalized for both graph ingestion and query throughput. The system should be profiled to detect memory bottlenecks during graph traversal, I/O lag during batch ingestion, and latency in semantic queries. Indexing strategies should be revisited for heavily queried properties such as `symbol`, `atomic_number`, or `material_id`.

For large-scale deployments, Neo4j cache configuration and heap sizing parameters should be tuned, and alternative backends such as TigerGraph or JanusGraph could be benchmarked to evaluate performance under increased data volume.

6.2.5 Public Release and Community Engagement

A curated version of GT-DB should be prepared for public release. This includes redacted datasets, environment files, reproducible containers, and limited-access credentials to a preloaded Neo4j instance hosted via cloud infrastructure. Documentation should accompany the release in the form of a version-controlled README, usage notebook, and dataset schema reference. Persistent identifiers such as DOIs (issued via Zenodo or similar platforms) should be registered for all public exports. Community participation should be enabled through GitHub issue tracking, contribution guidelines, and roadmap publication to encourage open development.

6.2.6 Institutional and Research Integration

GT-DB has demonstrated the effectiveness of graph-native modeling for materials data pipelines. Continued internal development should prioritize integration with ongoing institutional research efforts in AI-driven materials design, electronic structure prediction, and laboratory automation. Collaboration with faculty and student research groups will be essential for tailoring new modules and scaling validation efforts. The system should serve not only as a backend for predictive modeling but also as a tool for hypothesis generation through queryable data exploration.

The GT-DB initiative represents a significant step toward restructuring materials data workflows; its graph-based architecture offers extensibility, expressiveness, and computational leverage across research domains. This foundation provides a viable platform for advanced informatics pipelines and offers a model for future projects aiming to modernize materials data infrastructure.

Acknowledgments

The author would like to thank the Embry-Riddle Aeronautical University College of Engineering EECS department. Special appreciation goes to the project advisor, Professor M. Ilhan Akbas, and the faculty committee for their guidance and insightful feedback throughout the development of GT-DB. The author also acknowledges the developers of open-source tools (RDKit, pymatgen, Neo4j, PyTorch Geometric, and others) that made this work possible. Finally, gratitude is extended to friends and family for their encouragement during the project. This research report is submitted in fulfillment of the requirements for the Spring 2025 Undergraduate Research Project at ERAU.

GitHub Repository

For full source code and documentation of the GT-DB project, see the project's GitHub page at `https://github.com/AllenMProject/GRP25/`.

References

[1] A. Jain et al., "Commentary: The Materials Project: A materials genome approach to accelerating materials innovation," *APL Materials*, vol. 1, no. 1, p. 011002, 2013. doi: 10.1063/1.4812323.

[2] National Center for Biotechnology Information (NCBI), "PubChem: an open chemistry database at NIH," *PubChem Database*, [Online]. Available: `https://pubchem.ncbi.nlm.nih.gov/`.

[3] RDKit Developers, "RDKit: Open-Source Cheminformatics Software," [Online]. Available: `https://www.rdkit.org/`.

[4] S. P. Ong et al., "Python Materials Genomics (pymatgen): A robust, open-source Python library for materials analysis," *Computational Materials Science*, vol. 68, pp. 314–319, 2013.

[5] M. Fey and J. E. Lenssen, "Fast Graph Representation Learning with PyTorch Geometric," *arXiv preprint* arXiv:1903.02428, 2019.

[6] Neo4j, Inc., "Neo4j Graph Database Platform Overview," [Online]. Available: `https://neo4j.com/`.

[7] Databricks, "MLflow: An open source platform for the machine learning lifecycle," [Online]. Available: `https://mlflow.org/`.

[8] V. Venugopal and E. Olivetti, "MatKG: An autonomously generated knowledge graph in material science," *Scientific Data*, vol. 11, Article 217, 2024.