Graph Databases & High-Performance Python

From Data Models to HPC Triangle Analytics

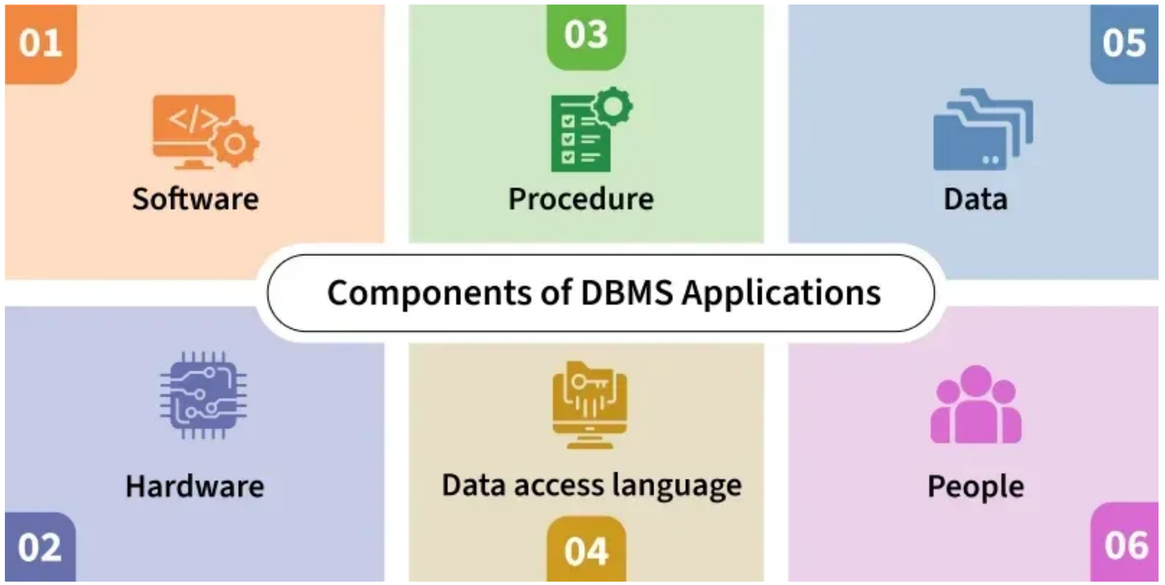
Daria Kriuchkova

Agenda

- Graph Databases: Core Idea
- Relational vs Graph Models
- From ORM to OGM
- Python Objects to Graph Structures
- Limits of OGMs for Analytics
- HPC with Python on Graph Data
- Triangle Counting in G(n,p)

What is a Database?

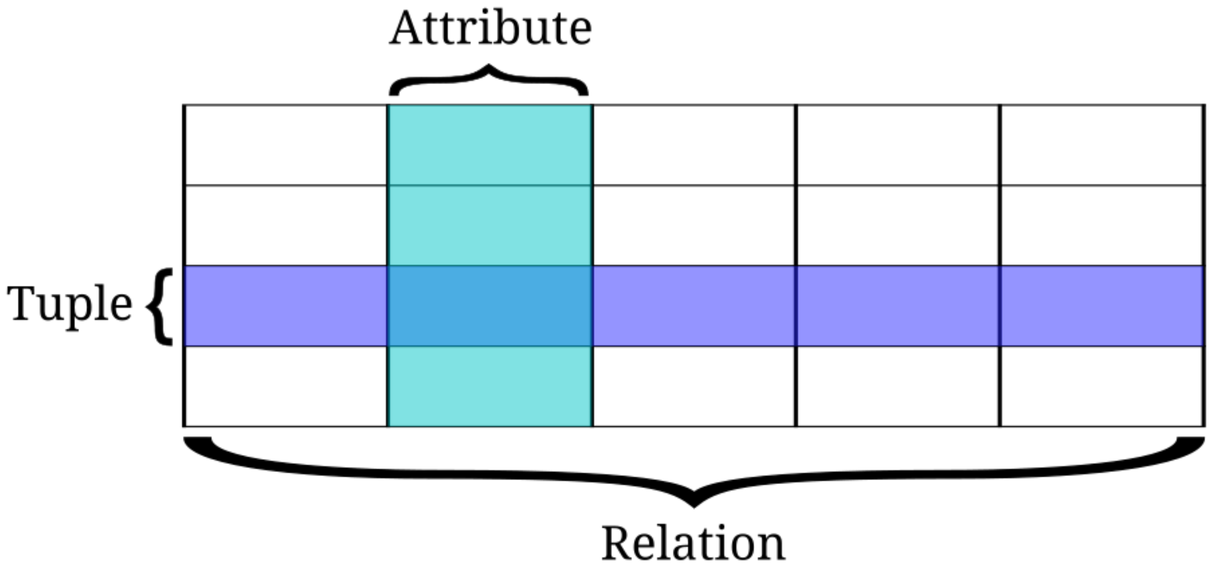
A **database** is an organised collection of data or a type of data store based on the use of a database management system (DBMS), the software that interacts with end users, applications, and the database itself to capture and analyse the data.



Pic source: geeksforgeeks.org

Relational DBs

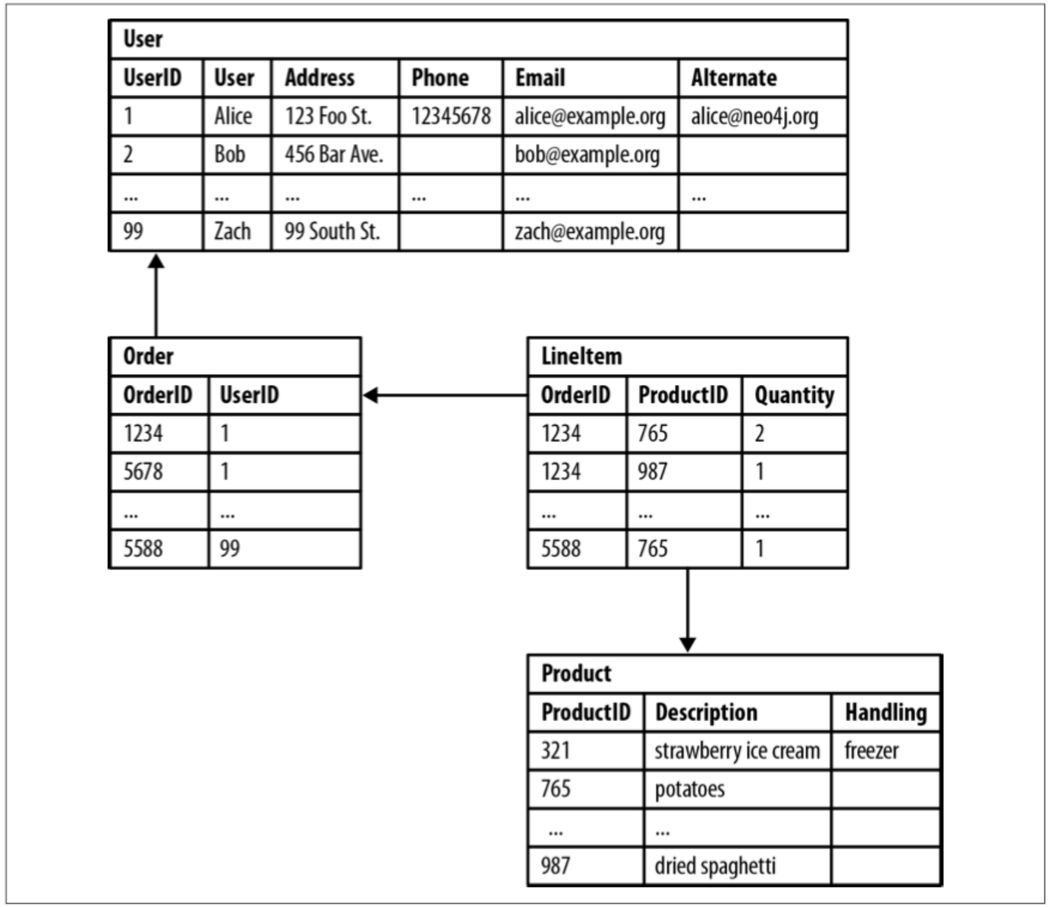
RDBMS



DBMS engine:

- Implements **tables, rows, schemas**, and relational constraints.
- The query optimiser builds efficient execution plans for **joins**

Relational Databases Lack Relationships



User

UserID	User	Address	Phone	Email	Alternate
1	Alice	123 Foo St.	12345678	alice@example.org	alice@neo4j.org
2	Bob	456 Bar Ave.		bob@example.org	
...	...	...	...	...	...
99	Zach	99 South St.		zach@example.org	

Order

OrderID	UserID
1234	1
5678	1
...	...
5588	99

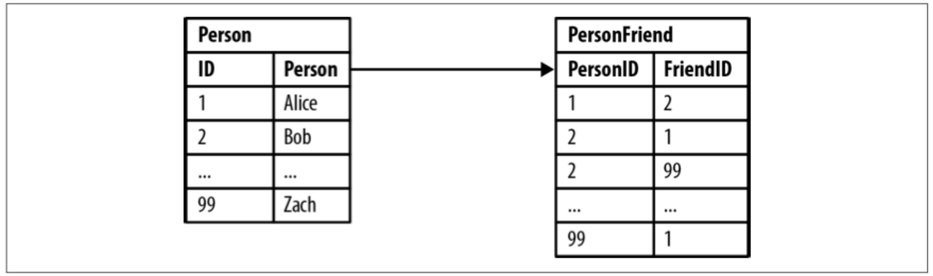
LineItem

OrderID	ProductID	Quantity
1234	765	2
1234	987	1
...	...	...
5588	765	1

Product

ProductID	Description	Handling
321	strawberry ice cream	freezer
765	potatoes	
...	...	
987	dried spaghetti	

Figure 2-1. Semantic relationships are hidden in a relational database



Person

ID	Person
1	Alice
2	Bob
...	...
99	Zach

PersonFriend

PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

Figure 2-2. Modeling friends and friends-of-friends in a relational database

```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
  ON PersonFriend.FriendID = p1.ID
JOIN Person p2
  ON PersonFriend.PersonID = p2.ID
WHERE p2.Person = 'Bob'
```

Pic source: Graph Databases: The Definitive Guide (O'Reilly)

Graph Databases Relationships

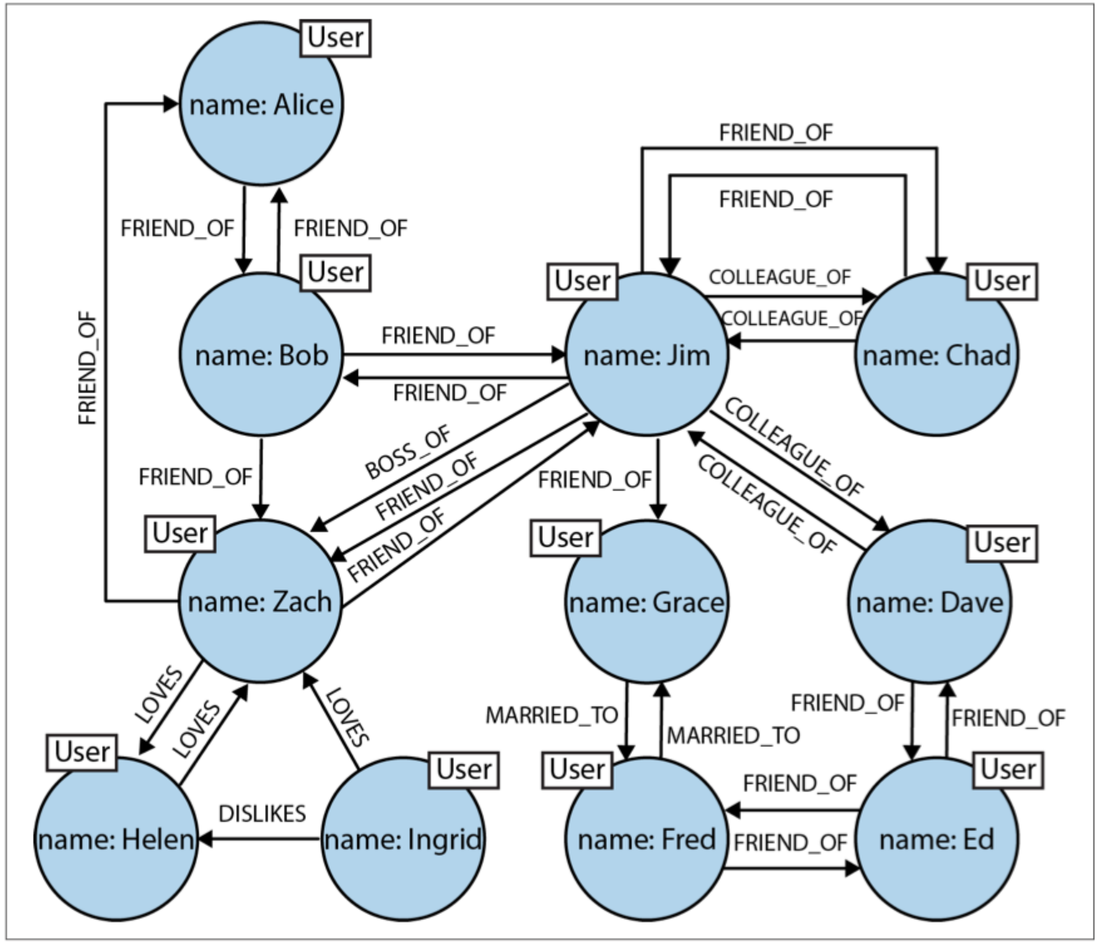
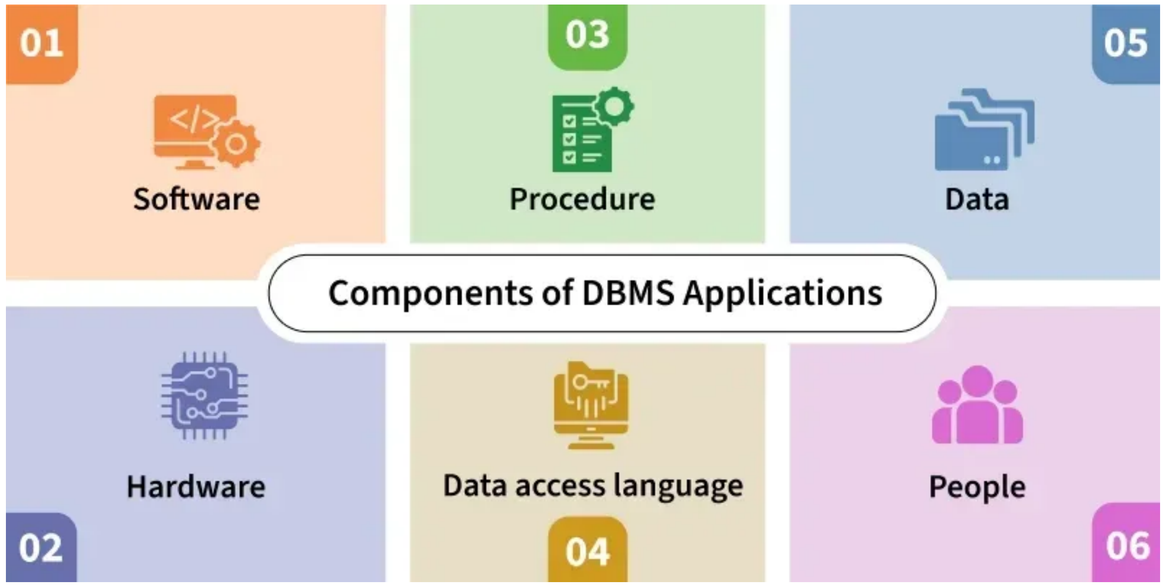
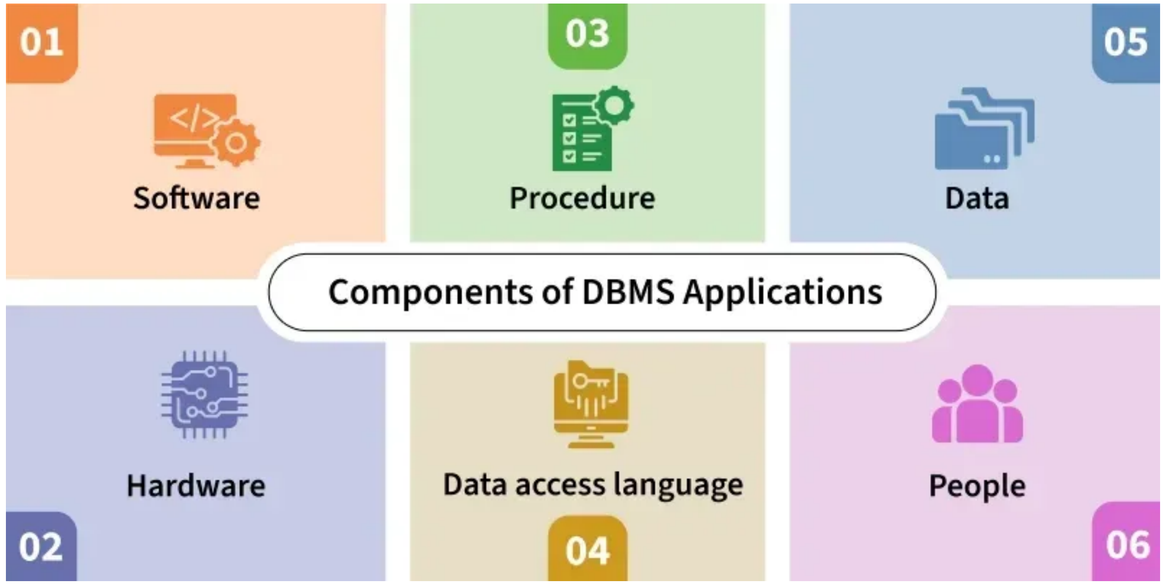


Figure 2-5. Easily modeling friends, colleagues, workers, and (unrequited) lovers in a graph


01
Software
03
Procedure
05
Data
Components of DBMS Applications
Hardware
02
Data access language
04
People
06

The database engine is built around **graph-native storage and processing**. It supports ACID transactions, persistence, recovery, and high availability, but is optimized for **connected data**, not tables.



The database engine is built around **graph-native storage and processing**. It supports ACID transactions, persistence, recovery, and high availability, but is optimized for **connected data**, not tables.

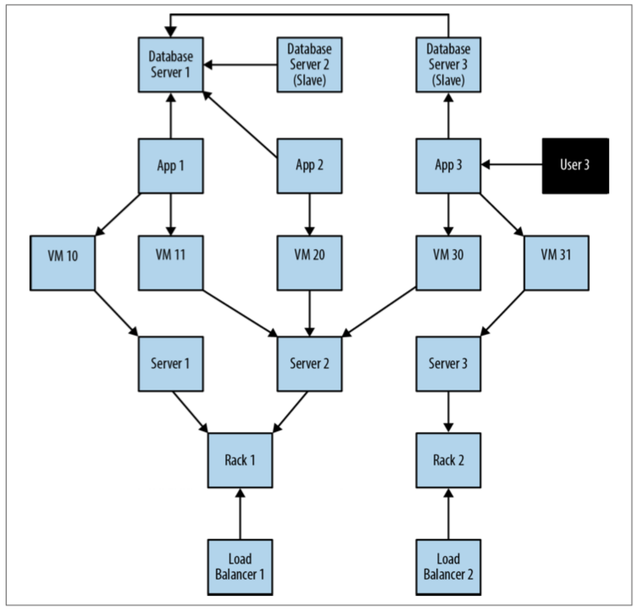
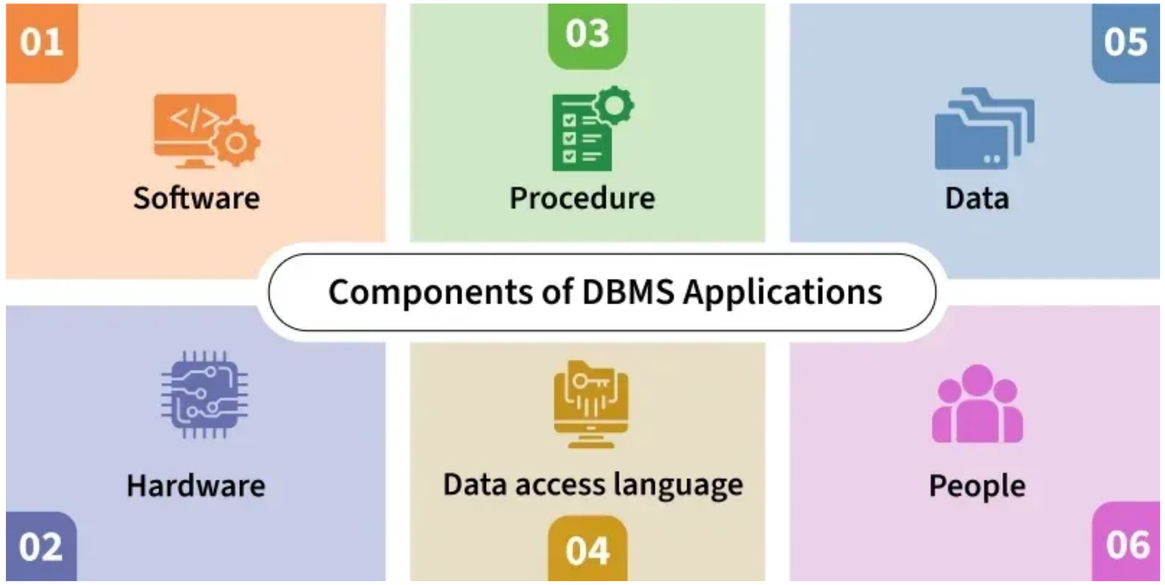
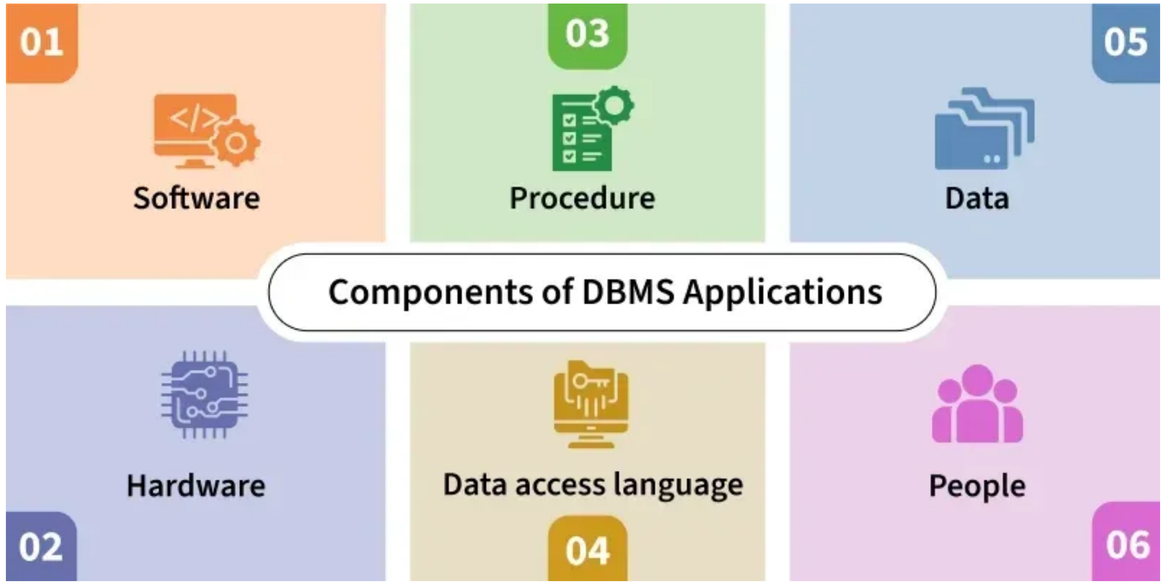


Figure 3-2. Simplified snapshot of application deployment within a data center



Optimised for traversal-heavy workloads.

The database engine is built around **graph-native storage and processing**. It supports ACID transactions, persistence, recovery, and high availability, but is optimized for **connected data**, not tables.



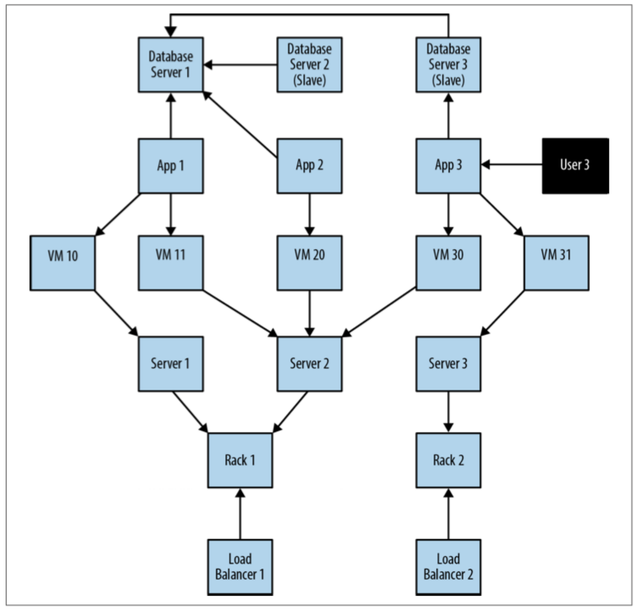
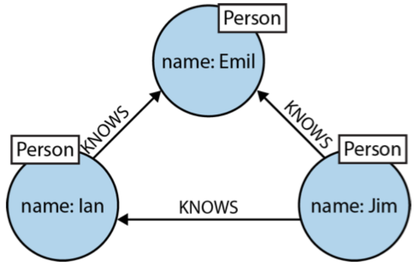


Figure 3-2. Simplified snapshot of application deployment within a data center

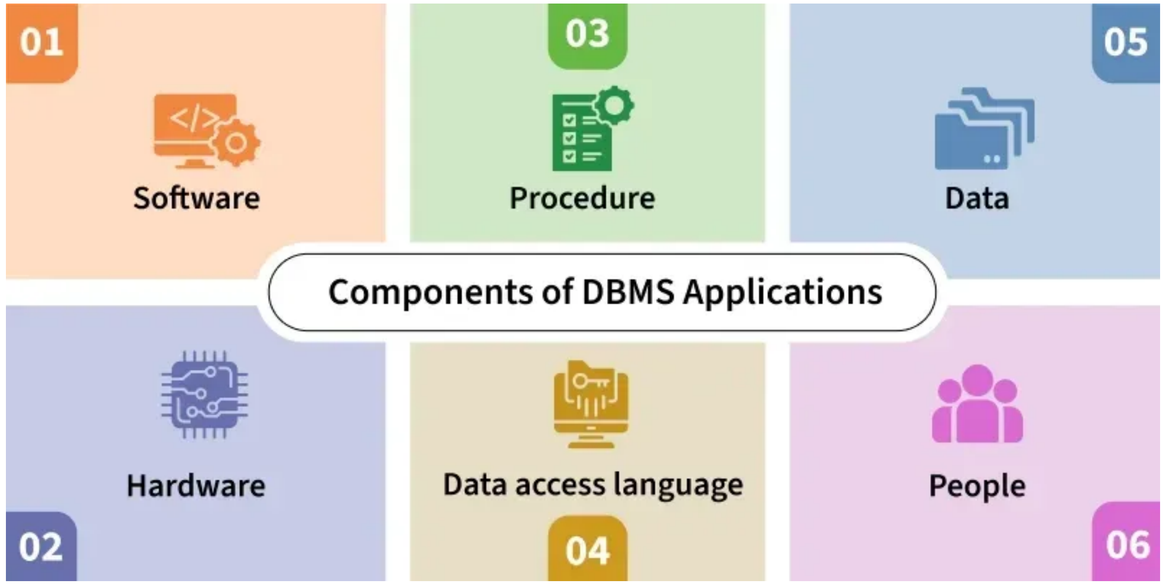
Optimised for traversal-heavy workloads.

Queries execute as graph traversals and pattern matching, not joins.
Execution cost depends on the local subgraph explored, not total graph size.
Example: “friends-of-friends” runs in milliseconds even on large graphs.

The database engine is built around **graph-native storage and processing**. It supports ACID transactions, persistence, recovery, and high availability, but is optimized for **connected data**, not tables.



A graph database stores data as a labeled property graph



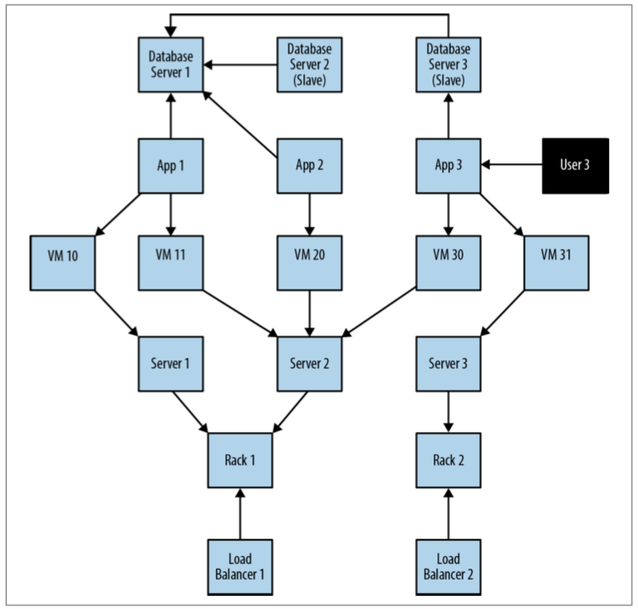


Figure 3-2. Simplified snapshot of application deployment within a data center

Optimised for traversal-heavy workloads.

Queries execute as graph traversals and pattern matching, not joins.
Execution cost depends on the local subgraph explored, not total graph size.
Example: “friends-of-friends” runs in milliseconds even on large graphs.

Graph Databases — Architectural Overview

storage model, query language, execution model, scalability

Neo4j	Memgraph	ArangoDB	JanusGraph
native graph storage (nodes & relationships first-class)	**in-memory graph**	**multi-model (graph + document + key-value)**	**distributed (on top of Cassandra / HBase / etc.)**
Cypher	openCypher	AQL	Gremlin
disk-based, optimized traversals + GDS (C++/native)	real-time, streaming-friendly	general-purpose engine	cluster-oriented
mature ecosystem, analytics + applications	fast subgraph extraction, tight Python workflows	hybrid data, not a “pure” graph system	very large graphs, high operational complexity

Object-Relational Mapping

**solves “impedance mismatch”
and converts Python objects into
a database structure.**

Object-Relational Mapping

solves “impedance mismatch” and converts Python objects into a database structure.

RDBMS: Converts Python operations into SQL under the hood.

Python classes ↔ SQL tables

Object attributes ↔ columns

Object relationships ↔ foreign keys

```
class Person(Base):
    __tablename__ = "persons"
    id = Column(Integer, primary_key=True)
    name = Column(String)

class Friendship(Base):
    __tablename__ = "friendships"
    person_id = Column(ForeignKey("persons.id"), primary_key=True)
    friend_id = Column(ForeignKey("persons.id"), primary_key=True)
```

SQLAlchemy/Django

Object-Graph Mapping

solves “impedance mismatch” and converts Python objects into a database structure.

Python classes ↔ graph nodes

Object attributes ↔ node properties

Python references ↔ graph edges

RDBMS: Converts Python operations into SQL under the hood.

Python classes ↔ SQL tables

Object attributes ↔ columns

Object relationships ↔ foreign keys

```
class Person(Base):
    __tablename__ = "persons"
    id = Column(Integer, primary_key=True)
    name = Column(String)

class Friendship(Base):
    __tablename__ = "friendships"
    person_id = Column(ForeignKey("persons.id"), primary_key=True)
    friend_id = Column(ForeignKey("persons.id"), primary_key=True)
```

SQLAlchemy/Django

Object-Graph Mapping

solves “impedance mismatch” and converts Python objects into a database structure.

Python classes ↔ graph nodes
Object attributes ↔ node properties
Python references ↔ graph edges

```python
from neomodel import StructuredNode, StringProperty, RelationshipTo

class Person(StructuredNode):
    name = StringProperty()
    friends = RelationshipTo("Person", "FRIENDS_WITH")

alice = Person(name="Alice").save()
bob   = Person(name="Bob").save()
alice.friends.connect(bob)

friends_of_alice = alice.friends.all()
```

Neomodel/Neo4

RDBMS: Converts Python operations into SQL under the hood.

Python classes ↔ SQL tables
Object attributes ↔ columns
Object relationships ↔ foreign keys

```python
class Person(Base):
    __tablename__ = "persons"
    id = Column(Integer, primary_key=True)
    name = Column(String)

class Friendship(Base):
    __tablename__ = "friendships"
    person_id = Column(ForeignKey("persons.id"), primary_key=True)
    friend_id = Column(ForeignKey("persons.id"), primary_key=True)
```

SQLAlchemy/Django

Object-Graph Mapping

solves “impedance mismatch” and converts Python objects into a database structure.

Python classes ↔ graph nodes
Object attributes ↔ node properties
Python references ↔ graph edges

```python
from neomodel import StructuredNode, StringProperty, RelationshipTo

class Person(StructuredNode):
    name = StringProperty()
    friends = RelationshipTo("Person", "FRIENDS_WITH")

alice = Person(name="Alice").save()
bob   = Person(name="Bob").save()
alice.friends.connect(bob)

friends_of_alice = alice.friends.all()
```

Neomodel/Neo4j

- Relationships are **explicit objects**, not foreign-key fields.
- Traversal is done by walking edges, not joining tables.
- The graph model is much closer to the object model → *less mismatch*.

RDBMS: Converts Python operations into SQL under the hood.

Python classes ↔ SQL tables
Object attributes ↔ columns
Object relationships ↔ foreign keys

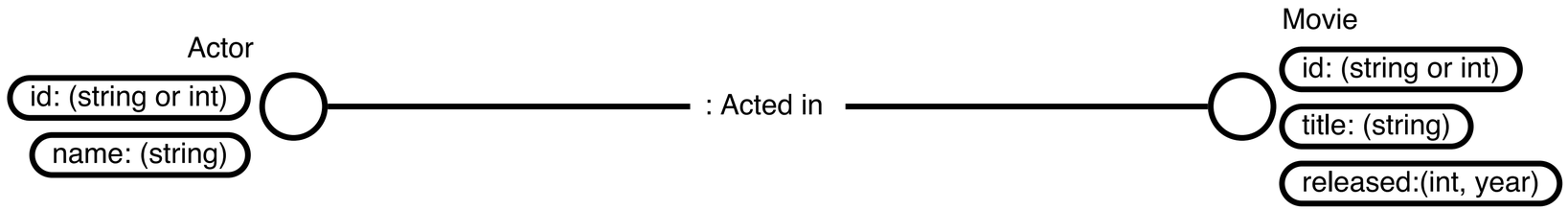
```python
class Person(Base):
    __tablename__ = "persons"
    id = Column(Integer, primary_key=True)
    name = Column(String)

class Friendship(Base):
    __tablename__ = "friendships"
    person_id = Column(ForeignKey("persons.id"), primary_key=True)
    friend_id = Column(ForeignKey("persons.id"), primary_key=True)
```

SQLAlchemy/Django

How Python Maps Objects to Graph Structures

Example Project: A Minimal Python Object–Graph Mapper



Pic Done: arrows.app
GitHub project: nodes2022

How Python Maps Objects to Graph Structures

Class Definition → Graph Schema

```python
class Movie:
    def __init__(self, title: str, released: Optional[int] = None):
        self.title = title
        self.released = released
```

- **Node** → Movie (entity)
- **Properties** → title, released

```python
class NodeBase(type):
    def __new__(mcs, name, bases, attrs):
        properties = {}
        for attr_name, attr in attrs.items():
            if isinstance(attr, Property):
                properties[attr_name] = attr
        attrs["_properties"] = properties
        kls = type.__new__(mcs, name, bases, attrs)
        return kls

    def __init__(cls, name, bases, attrs):
        super().__init__(name, bases, attrs)
```

Python receives the **raw class body** as a dictionary.

Declarative schema: only attributes declared as Property are treated as graph properties.

The class gains a **schema description** at creation time

A new class is created that already *knows*:

- what properties it has
- how it can be serialised into query parameters
- how queries can be generated

GitHub project: nodes2022

How Python Maps Objects to Graph Structures

Python class → Metadata → Cypher

Collects values from **kwargs and assigns them to declared properties

```python
class Node(metaclass=NodeBase):

    def __init__(self, /, **kwargs):
        self.cached_properties = {}
        for fn, prop in self._properties.items():
            val = kwargs.pop(fn, None)
            setattr(self, fn, val)
```

Validates input against the schema and extracts query parameters

```python
    @classmethod
    def _get_params_from_kwargs(cls, **kwargs):
        params = {}
        for prop_name, prop in cls._properties.items():
            params[prop_name] = kwargs.get(prop_name, None)
        for k in kwargs:
            if k not in cls._properties:
                raise AttributeError(f"'{k}' is not a valid property for model {cls}")
        return params
```

GitHub project: nodes2022

How Python Maps Objects to Graph Structures

Python class → Metadata → Cypher

Compiles class metadata into a parameterized Cypher query

```
52      @classmethod
53      def build_create_query(cls, **params):
54          node_alias = "node"
55          label = cls.__name__
56          query = f"CREATE ({node_alias}:{label}) "
57
58          set_query = [
59              f"{node_alias}.{k}=${k}"
60              for k in cls._properties
61          ]
62          set_query = ", ".join(set_query)
63          if set_query:
64              set_query = "SET " + set_query
65
66          params = cls._get_params_from_kwargs(**params)
67
68          query = query + set_query
69          return query, params
```

The graph **node label** is derived directly from the Python class name. No manual mapping

Cypher is **generated dynamically** from Python metadata.

Only properties collected by the metaclass are used:

- schema consistency
- no accidental fields
- safe query construction

Python objects → parameter dict

→ safe, parameterized Cypher queries

GitHub project: nodes2022

From Query Results to Python Objects

Hydration

- Database returns nodes and relationships
- OGM instantiates Python objects from results
- Relationships reconnect objects into an in-memory graph

DB result → {"title": "Matrix", "released": 1999}

movie = Movie(title="Matrix", released=1999)

(Keanu)-[:ACTED_IN]->(Matrix)

keanu.acted_in.append(matrix)

From Graph Traversal to Graph Computation

Abstractions help us model graphs. Computation requires numeric engines.

Graph Databases

Role: Data structure & storage

- nodes, edges, properties
- persistence
- indexing

OGM

Role: Data access & modeling

- Python objects
- schema abstraction
- queries & hydration

HPC / Numeric Stack

Role: Computation

- NumPy
- Numba
- vectorised loops

✅ triangles, centrality, simulations

✅ orders-of-magnitude speedups

HPC with Python on Graph Data

What Breaks and Why It Matters

Problem 1 — CPython is slow for graph workloads

1) Python loops are slow → high per-iteration overhead

2) NetworkX is pure Python → collapses for $n \gtrsim 10^5$

3) GIL → no true parallelism for CPU-bound graph algorithms

4) Poor cache locality:

→ graphs cause random memory access

→ CPU cache misses dominate runtime

HPC with Python on Graph Data

What Breaks and Why It Matters

Problem 1 — CPython is slow for graph workloads

1) Python loops are slow → high per-iteration overhead

2) NetworkX is pure Python → collapses for $n \gtrsim 10^5$

3) GIL → no true parallelism for CPU-bound graph algorithms

4) Poor cache locality:

→ graphs cause random memory access

→ CPU cache misses dominate runtime

Problem 2 — Graph algorithms are computationally heavy

1) Triangle counting → worst case O(n^3)

2) Betweenness centrality → extremely expensive (many shortest paths)

3) PageRank:

→ iterative matrix–vector multiplications

→ convergence takes many rounds

How Python Escapes the Performance Trap

HPC Stack

What we already know

How Python Escapes the Performance Trap

HPC Stack

What we already know

🧱 NumPy — Vectorization

- Operates on arrays, not Python objects
- Inner loops run in C / Fortran
- Better cache locality
- nRemoves Python-level iteratio

How Python Escapes the Performance Trap

HPC Stack

What we already know

🧱 NumPy — Vectorization

- Operates on arrays, not Python objects
- Inner loops run in C / Fortran
- Better cache locality
- Removes Python-level iteratio

⚙️ Numba — JIT Compilation

- Python → LLVM → machine code
- Removes interpreter overhead
- @njit, parallel=True → multi-core CPU
- Explicit control over loops

How Python Escapes the Performance Trap

HPC Stack

Scaling further

How Python Escapes the Performance Trap

HPC Stack

⚡ **Parallel execution**

- parallel=True → loop-level parallelism
- Bypasses GIL for numeric kernels
- Fits graph algorithms with independent iterations

Scaling further

How Python Escapes the Performance Trap

HPC Stack

⚡ **Parallel execution**

- parallel=True → loop-level parallelism
- Bypasses GIL for numeric kernels
- Fits graph algorithms with independent iterations

🧠 **GPU (cuGraph)**

- Thousands of cores
- Massive parallelism for graph primitives
- Same idea: move computation out of Python

Scaling further

Triangle counting in random graphs G(n,p)

Problem Solving

$G \sim G(n,p), \quad X = \#\{\text{triangles in } G\}\,. \quad$ Find $\mathbb{E}[X],\ \mathrm{Var}(X)\,.$

$$\mathbb{E}[X] = \binom{n}{3} p^3$$

$$\mathrm{Var}(X) = \binom{n}{3} p^3(1-p^3) + 3\binom{n}{4} p^5(1-p)$$

Triangle counting in random graphs G(n,p)

Problem Solving

$$G \sim G(n,p), \quad X = \#\{\text{triangles in } G\}. \quad \text{Find } \mathbb{E}[X],\ \mathrm{Var}(X).$$

Why it matters

- Fundamental graph metric
- Computationally heavy
- Quickly breaks naïve Python implementations

What I used

- NumPy → graph as numeric data
- Numba JIT → fast triangle counting
- CPU execution (no database, no OGM)

Computation

Sources

1. Introduction of DBMS — GeeksforGeeksn: https://www.geeksforgeeks.org/dbms/introduction-of-dbms-database-management-system-set-1/
2. Overview of Database Types — DataCamp Blog: https://www.datacamp.com/blog/types-of-databases-overview
3. SQL vs NoSQL Databases — DataCamp Blog: https://www.datacamp.com/blog/sql-vs-nosql-databases
4. Relational Database — Wikipedia: https://en.wikipedia.org/wiki/Relational_database
5. Graph Traversal — Wikipedia: https://en.wikipedia.org/wiki/Graph_traversal
6. Getting Started with Graph Database — Neo4j Documentation: https://neo4j.com/docs/getting-started/graph-database/
7. Graph Databases: The Definitive Guide (O'Reilly) — https://graphdatabases.com
8. Building a Neo4j Python OGM (Nodes 2022) — Neo4j Conference Video - https://neo4j.com/videos/086-building-a-neo4j-python-ogm-nodes2022-estelle-scifo/
9. Object–Relational Mapping — Wikipedia https://en.wikipedia.org/wiki/Object%E2%80%93relational_mapping
10. Nodes 2022 GitHub Repository — Neo4j Python OGM example code - https://github.com/stellasia/nodes2022
11. A Comprehensive Guide to Cypher Map Projection — Medium (Neo4j) - https://medium.com/neo4j/a-comprehensive-guide-to-cypher-map-projection-2622b879e886

Thank you