Document Store
Document-Oriented Database

Chananel Perel - 2024

Graph DB

A graph database stores nodes and relationships instead of tables or documents. Data is stored just like you might sketch ideas on a whiteboard. Your data is stored without restricting it to a pre-defined model, allowing a very flexible way of thinking about and using it.

Graph DB - Why?

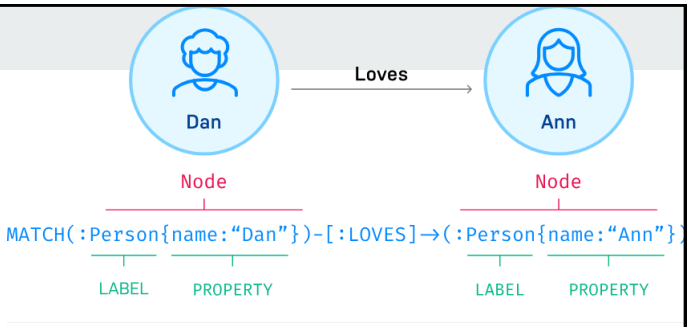
We live in a connected world, and understanding most domains requires processing rich sets of connections to understand what's really happening. Often, we find that the connections between items are as important as the items themselves.
How else do people do this today? While existing relational databases can store these relationships, they navigate them with expensive JOIN operations or cross-lookups, often tied to a rigid schema. It turns out that "relational" databases handle relationships poorly. In a graph database, there are no JOINs or lookups. Relationships are stored natively alongside the data elements (the nodes) in a much more flexible format. Everything about the system is optimized for traversing through data quickly; millions of connections per second, per core.

Graph databases address big challenges many of us tackle daily. Modern data problems often involve many-to-many relationships with heterogeneous data that sets up needs to:

- Navigate deep hierarchies.
- Find hidden connections between distant items.
- Discover inter-relationships between items.

Whether it's a social network, payment networks, or road network you'll find that everything is an interconnected graph of relationships. And when you want to ask questions about the real world, many questions are about the relationships rather than about the individual data elements.

Graph DB - Neo4j



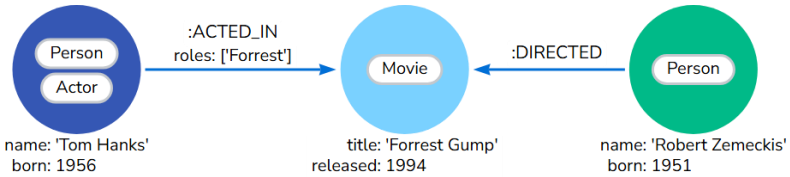
Nodes are the entities in the graph.

- Nodes can be tagged with labels, representing their different roles in your domain (for example, Person).
- Nodes can hold any number of key-value pairs, or properties (for example, name).

Relationships provide directed, named connections between two node entities (for example, Person LOVES Person).

- Relationships always have a direction, a type, a start node, and an end node, and they can have properties, just like nodes.
- Nodes can have any number or type of relationships without sacrificing performance.
- Although relationships are always directed, they can be navigated efficiently in any direction.

Graph DB - Neo4j



In mathematics, graph theory is the study of graphs.

In graph theory:

- **Nodes** are also referred to as **vertices** or **points**.
- **Relationships** are also referred to as **edges**, **links**, or **lines**.

https://neo4j.com/docs/getting-started/appendix/graphdb-concepts/

Graph D

§ Naming conventions

Node labels, relationship types, and properties (the key part) are case sensitive, meaning, for example, that the property `name` is different from the property `Name`.

The following naming conventions are recommended:

Table 1. Naming conventions

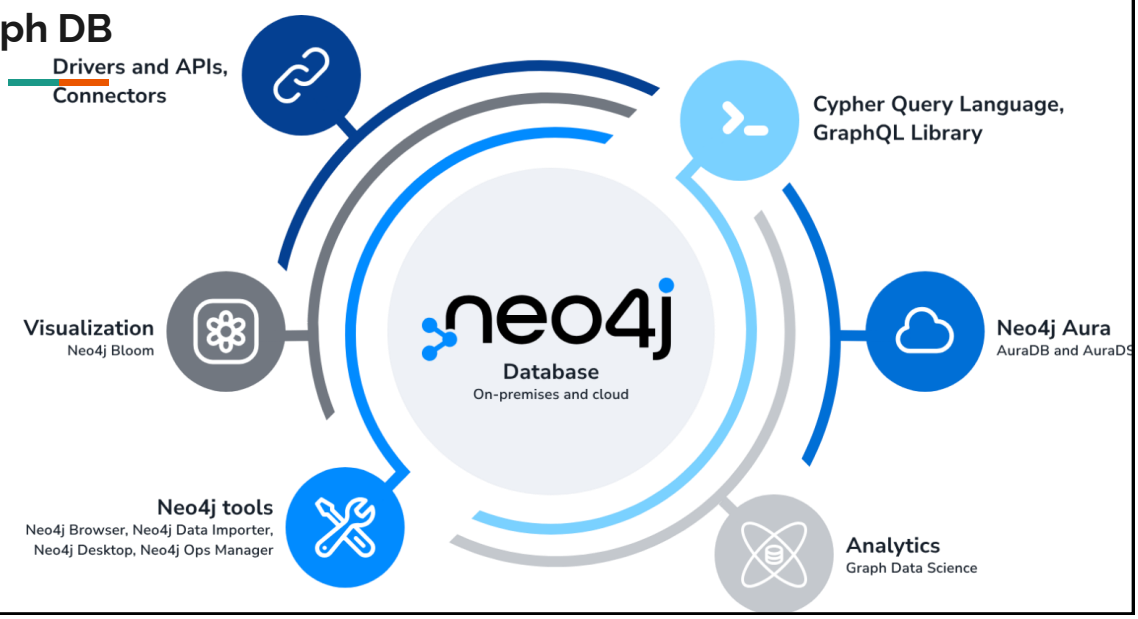
Graph entity	Recommended style	Example
Node label	Camel case, beginning with an upper-case character	`:VehicleOwner` rather than `:vehicle_owner`
Relationship type	Upper case, using underscore to separate words	`:OWNS_VEHICLE` rather than `:ownsVehicle`
Property	Lower camel case, beginning with a lower-case character	`firstName` rather than `first_name`

Graph DB - Neo4j

With a native graph database at the core, Neo4j stores and manages data in a natural, connected state. The graph database takes a property graph approach, which is beneficial for both traversal performance and operations runtime.

Neo4j started as a graph database and has evolved into a rich ecosystem with numerous tools, applications, and libraries. This ecosystem allows you to seamlessly integrate graph technologies with your working environment.

Graph DB



Graph DB - Cypher query language

Cypher is an open data query language, based on the openCypher initiative. It is the most established and intuitive query language used for property graphs. Cypher can be characterized by following:

- easy to learn
- visual and logical
- secure, reliable, and data-rich
- open and flexible

See introduction to Cypher to familiarize yourself with the fundamentals of Cypher.
For detailed information, refer to the Cypher manual.

https://neo4j.com/docs/getting-started/cypher-intro/
https://neo4j.com/product/cypher-graph-query-language/?ref=product
https://neo4j.com/docs/cypher-manual/current/
https://neo4j.com/docs/getting-started/appendix/example-data/#neo4j-sandbox (Browser | WorkSpace)
https://graphacademy.neo4j.com/categories/cypher/
https://neo4j.com/docs/cypher-cheat-sheet/5/auradb-free

Graph DB - openCypher

https://opencypher.org/
openCypher is an open source implementation of Cypher® - the most widely adopted, fully-specified, and open query language for property graph databases. Cypher was developed by Neo4j®.
Cypher provides an intuitive way to work with property graphs today and is the best on-ramp to the graph query language (GQL) standard being developed by ISO.

SQL ... and now GQL
The international committees that develop the SQL standard have voted to initiate GQL (Graph Query Language) as a new database query language.

Graph DB - Cypher query fundamentals

In DB list you'll see how many nodes you have and all their labels, the number of relationships and their types, and also a list of all the property keys present in the database.

A node can have multiple labels and the list shows all labels in the database.
Relationships on the other hand, can have only one type, but multiple relationships can exist between the same nodes.

The list of property keys include both node properties and relationship properties.
You can click any of the labels, types, or property keys to run a query that returns a sample of nodes and/or relationships that has that label, type, or property.

Graph DB - queries with graph results

Central to the Query UI is the Cypher editor on top where you write and run your queries and the list of result frames.
When you run any query, a result frame shows the query and its results, pushing previous ones down.
You can edit the query within the frame and run it again without creating a new result frame.

The frame has two important views: Graph and Table. (Plan is for showing query plans and RAW for the raw results).
The default view is graph if your query returns graph elements like nodes, relationships or paths.

Relationships are only shown if you return a path, or name and return the relationships as well as the nodes.
On the right side of the graph view you can see the properties panel with information about the entities in your graph.
You can click on any of them to change their styling, i.e. color, size, and caption, clicking on one element shows its attributes.

Graph DB - queries tabular results

If your query only returns scalar values (like strings or numbers), the result defaults to the table view and the graph view is not available.

The query uses variables, c and p, for the category and the product as you will want to refer to them later.
The RAW option shows the submitted query, the Neo4j server version and address, the result, and a summary.

If you prefix your query with EXPLAIN or PROFILE you can see a plan option for a visual query plan, which helps later with optimizations. The difference between using EXPLAIN and PROFILE is that EXPLAIN provides estimates of the query steps where PROFILE provides the exact steps and number of rows retrieved for the query. Providing you are simply querying the graph and not updating anything, it is fine to execute the query multiple times using PROFILE. In fact, as part of query tuning, you should execute the query at least twice as the first execution involves the generation of the execution plan which is then cached. That is, the first PROFILE of a query will always be more expensive than subsequent queries.

Graph DB - Queries 1

```
MATCH (p:Person)
WHERE p.name = 'Tom Hanks' OR p.name = 'Rita Wilson'
RETURN p.name, p.born

MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name = 'Tom Hanks'
RETURN m.title

MATCH (p)-[:ACTED_IN]->(m)
WHERE p:Person AND m:Movie AND m.title='The Matrix'
RETURN p.name

MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE 2000 <= m.released <= 2003
RETURN p.name, m.title, m.released

MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name='Jack Nicholson' AND m.tagline IS NOT NULL
RETURN m.title, m.tagline
```

Graph DB - Queries 2

```
MATCH (p:Person)
WHERE p.born IN [1965, 1970, 1975]
RETURN p.name, p.born

MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE  'Neo' IN r.roles AND m.title='The Matrix'
RETURN p.name, r.roles


Properties:
MATCH (p:Person)
RETURN p.name, keys(p)

MATCH (p:Person)
WHERE  p.born.year > 1960
    AND p:Actor
    AND p:Director
RETURN p.name, p.born, labels(p)
```

Graph DB - Queries 3

```
Schema:
CALL db.propertyKeys()
CALL db.schema.visualization()
CALL db.schema.nodeTypeProperties()
CALL db.schema.relTypeProperties()


MATCH (p:Person)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(p)
WHERE  p.born.year > 1960
RETURN p.name, p.born, labels(p), m.title

MATCH (p:Person)-[r]->(m:Movie)
WHERE  p.name = 'Tom Hanks'
RETURN m.title AS movie, type(r) AS relationshipType

MATCH (m:Movie)
WHERE "Israel" IN m.countries
RETURN m.title, m.languages, m.countries

MATCH (p:Actor | Person)
RETURN DISTINCT labels(p)

MATCH (p)
WHERE p:Actor and p:Person
RETURN DISTINCT labels(p)
```

Graph DB - Strings

```
MATCH (m:Movie)
WHERE  m.title STARTS WITH 'Toy Story'
RETURN m.title, m.released

MATCH (m:Movie)
WHERE  m.title CONTAINS 'River'
RETURN m.title, m.released

MATCH (p:Person)-[:ACTED_IN]->()
WHERE toLower(p.name) STARTS WITH 'michael'
RETURN p.name

MATCH (p:Person)
WHERE toUpper(p.name) ENDS WITH 'DEMILLE'
RETURN p.name
```

Graph DB - Query Patterns

```
MATCH (p:Person)-[:WROTE]->(m:Movie)
//WHERE NOT exists( (p)-[:DIRECTED]->(m) )
RETURN p.name, m.title

MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE  p.name = 'Tom Hanks'
AND exists {(p)-[:DIRECTED]->(m)}
RETURN p.name, labels(p), m.title

Same - much more efficient:
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(p)
WHERE  p.name = 'Tom Hanks'
RETURN  m.title

For NOT - must do this:
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE  p.name = 'Tom Hanks'
AND NOT exists {(p)-[:DIRECTED]->(m)}
RETURN  m.title
```

Graph DB - Multiple MATCH Clauses

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
WHERE m.year > 2000
MATCH (m)<-[:DIRECTED]-(d:Person)
RETURN a.name, m.title, d.name

Same:
MATCH (a:Person)-[:ACTED_IN]->(m:Movie),
      (m)<-[:DIRECTED]-(d:Person)
WHERE m.year > 2000
RETURN a.name, m.title, d.name

Same - much more efficient:
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(d:Person)
WHERE m.year > 2000
RETURN a.name, m.title, d.name
```

Graph DB - Optional MATCH

```
MATCH (m:Movie) WHERE m.title = "Kiss Me Deadly"
MATCH (m)-[:IN_GENRE]->(g:Genre)<-[:IN_GENRE]-(rec:Movie)
MATCH (m)<-[:ACTED_IN]-(a:Actor)-[:ACTED_IN]->(rec)
RETURN rec.title, a.name

Optional (like left join):
MATCH (m:Movie) WHERE m.title = "Kiss Me Deadly"
MATCH (m)-[:IN_GENRE]->(g:Genre)<-[:IN_GENRE]-(rec:Movie)
OPTIONAL MATCH (m)<-[:ACTED_IN]-(a:Actor)-[:ACTED_IN]->(rec)
RETURN rec.title, a.name
```

Graph DB - Order

```
MATCH (p:Person)
WHERE p.born.year = 1980
RETURN p.name AS name,
       p.born AS birthDate
ORDER BY p.born

MATCH (p:Person)
WHERE p.born IS NOT NULL
RETURN p.name AS name, p.born AS birthDate
ORDER BY p.born DESC
```

Graph DB - Limit

```
MATCH (m:Movie)
WHERE m.released IS NOT NULL
RETURN m.title AS title,
m.released AS releaseDate
ORDER BY m.released DESC LIMIT 100

MATCH (p:Person)
WHERE p.born.year = 1980
RETURN  p.name as name,
p.born AS birthDate
ORDER BY p.born SKIP 40 LIMIT 10

MATCH (p:Person)-[:DIRECTED | ACTED_IN]->(m:Movie)
WHERE p.name = 'Tom Hanks'
RETURN DISTINCT m.title, m.released
ORDER BY m.title

MATCH (m:Movie)
RETURN DISTINCT m.year
ORDER BY m.year
```

Graph DB - Aggregating Data

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
// WHERE a.name = 'Tom Hanks'
RETURN // a.name AS actorName,
count(*) AS numMovies
```

Aggregation in Cypher is different from aggregation in SQL. In Cypher, you need not specify a grouping key. As soon as an aggregation function like count() is used, all non-aggregated result columns become grouping keys. The grouping is implicitly done, based upon the fields in the RETURN clause.

Create list:

```
MATCH (p:Person)
RETURN p.name, [p.born, p.died] AS lifeTime
LIMIT 10
```

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
RETURN a.name AS actor,
count(*) AS total,
collect(m.title) AS movies
ORDER BY total DESC LIMIT 10
```

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
WHERE m.year = 1920
RETURN  collect(DISTINCT m.title) AS movies,
collect( a.name) AS actors
```

Graph DB - Aggregating 2

There are more aggregating functions:

```
min()
max()
avg()
stddev()
sum()
```

count() versus size():

You can either use count() to count the number of rows, or alternatively, you can return the size of the collected results. The size() function returns the number of elements in a list.

```
MATCH (actor:Person)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(director:Person)
RETURN actor.name, director.name,
size(collect(m)) AS collaborations,
collect(m.title) AS movies

...
```

Graph DB - Graph Traversal

The anchor for a query will be based upon the fewest number of nodes that need to be retrieved into memory.
The next step if the query specifies a path is to follow the path.

No path:
```
MATCH (person:Person)-[]->(movie)
WHERE person.name = 'Walt Disney'
RETURN person, movie
```
With path:
```
MATCH p = ((person:Person)-[]->(movie))
WHERE person.name = 'Walt Disney'
RETURN p
```

useful functions that can be used to analyze paths:
- length(p) returns the length of a path.
- nodes(p) returns a list containing the nodes for a path.
- relationships(p) returns a list containing the relationships for a path.

Graph DB - Graph Traversal

Shortest path:
```
MATCH p = shortestPath((p1:Person)-[*]-(p2:Person))
WHERE p1.name = "Eminem"
AND p2.name = "Charlton Heston"
RETURN  p
```
Limit relationship:
```
MATCH p = shortestPath((p1:Person)-[:ACTED_IN*]-(p2:Person))
WHERE p1.name = "Eminem"
AND p2.name = "Charlton Heston"
RETURN  p
```

Length 2:
```
MATCH (p:Person {name: 'Eminem'})-[:ACTED_IN*2]-(others:Person)
RETURN  others.name
```

```
MATCH q=((p:Person {name: 'Eminem'})-[a:ACTED_IN*2]-(others:Person))
RETURN q
```

One to four:
```
MATCH (p:Person {name: 'Eminem'})-[:ACTED_IN*1..4]-(others:Person)
RETURN  others.name
```