Chapter 1

Neo4j: A Graph Database Management System

Contributed by Brian DuSell

1.1 Background

Behind most every web or mobile application today is some form of *database management system*: a system responsible for efficiently sifting through a large repository of data in order to retrieve information relevant to a user ("querying") and adding information to the repository in a way that makes changes visible to all parties as soon as possible ("updating"). Most beginning web developers are probably familiar with the concept of a *relational* database management system (RDBMS), which offers a principled way of organizing information into rows and columns of "tables." Despite the widespread use of RDBMSs, the rigidity of their table model is not necessarily a natural fit for representing graph-structured data. In this chapter, we will discuss an example of an alternative database paradigm: Neo4j, a *graph* database management system where vertices and edges are first-class citizens.

In the familiar RDBMS paradigm, *rows* serve as records of individual objects, and *columns* describe attributes of those objects. For instance, a table of "movies" might have columns for a movie's "title" and "release date," as well as a column containing an integer ID that is unique to each movie. The columns' data types and constraints are typically declared in advance as part of the database's *schema*. One RDBMS table naturally corresponds to one type of object.

The strength of RDBMSs lies in their ability to model relationships among tables. Note that the number of rows in a table may grow and shrink arbitrarily throughout the lifetime of an RDBMS, whereas the number of columns does not change dynamically and is not suitable for lists of variable size. How, then, would one represent a list of actors that appeared in each movie? The RDBMS solution is to define a new table of "actors" as well as a third table to model the instances when an actor appeared in a movie. This "acted-in" table would contain two columns: one which contains the unique ID of a row in the "movies" table, and another which contains the unique ID of a row in the "actors" table. Recovering the list of actors that appeared in the movie *The Matrix* would involve searching for the ID of the movie with the matching title and then performing a *join* over rows in the "acted-in" and "actors" tables with matching IDs.

What if our task is to count the degree of separation between two actors, à la the six degrees of Kevin Bacon? The problem is that this query requires following a chain of relationships of variable depth, something that would require multiple, iterative queries in a traditional RDBMS. Neo4j was

designed with this kind of graph-oriented query in mind; its front-end Cypher Query Language readily supports multi-hop queries, and its execution engine is built to handle them efficiently.

As a graph database management system, Neo4j aims to be more expressive, flexible, and scalable when it comes to graph-structured data. Whereas an RDBMS requires anticipating every type of relationship in advance when designing the database schema, Neo4j does not enforce this restriction, instead placing importance on supporting heterogeneous relationships. New types of vertices and edges may be inserted into the database at any time, without the need to redefine a schema. Neo4j assumes that the addition of new kinds of objects and relationships will be commonplace, so that an application's data model may evolve incrementally.

Two versions of Neo4j are publicly available: a free-of-cost, open-source Community Edition; and a closed-source, premium Enterprise Edition which supports additional features such as massively parallel computation. The Community Edition is licensed under GPLv3; the Enterprise Edition is licensed under a commercial Neo4j license. Both versions may be obtained via the Neo4j website.[1] The source code for the Community Edition is available on GitHub.[2] Drivers are available for major programming languages, including Java, Python, JavaScript.

1.2 Expressing Graphs

Graphs in Neo4j follow the *property graph model*. Under this model, graphs consist of a set of *nodes* (vertices) and *relationships* (edges) among nodes. Relationships are directed and connect one node to another node. Every relationship is labeled with exactly one *relationship type*. Multiple types of relationship may connect the same pair of nodes (i.e. "multi-edges" are supported). Nodes may be labeled with one or more *labels* that identify the type of object that the node represents. Both nodes and relationships may be labeled with any number of *properties*, which are key-value pairs that map names to arbitrary data. Neo4j property values have their own type system which includes strings, numbers, spatial points, and dates and times.

Figure 1.1 illustrates a conceptual schema for a movie database (note that Neo4j has no formal concept of a "schema" that must be declared ahead of time; such a diagram would serve simply as a form of documentation). This example database utilizes four types of node: *Movie*, *Genre*, *Person*, and *Keyword*. The database also uses six types of relationship describing the relationships among these objects. For instance, the presence of an `ACTED_IN` relationship connecting a *Person* node to a *Movie* node indicates the appearance of the actor in that movie. The same relationship may optionally be annotated with a "role" attribute describing the nature of the actor's appearance in the movie (e.g. "lead actor," "supporting actor," "stunt double," etc.). Nodes may also be annotated with properties; the *Movie* node, for example, is labeled with its title, summary, and rating. Note that keywords and genres are both represented as nodes in order to accommodate movies with multiple keywords and genres.

1.3 Syntax

Neo4j uses a dedicated front-end language called the Cypher Query Language ("Cypher"). Cypher includes commands for creating, querying, updating, and deleting graph-structured data. By design, Cypher syntax superficially resembles that of SQL, the standard query language for RDBMSs. This allows programmers already familiar with SQL to pick up Cypher by example with relative

[1] `https://neo4j.com/subscriptions/`
[2] `https://github.com/neo4j/neo4j`

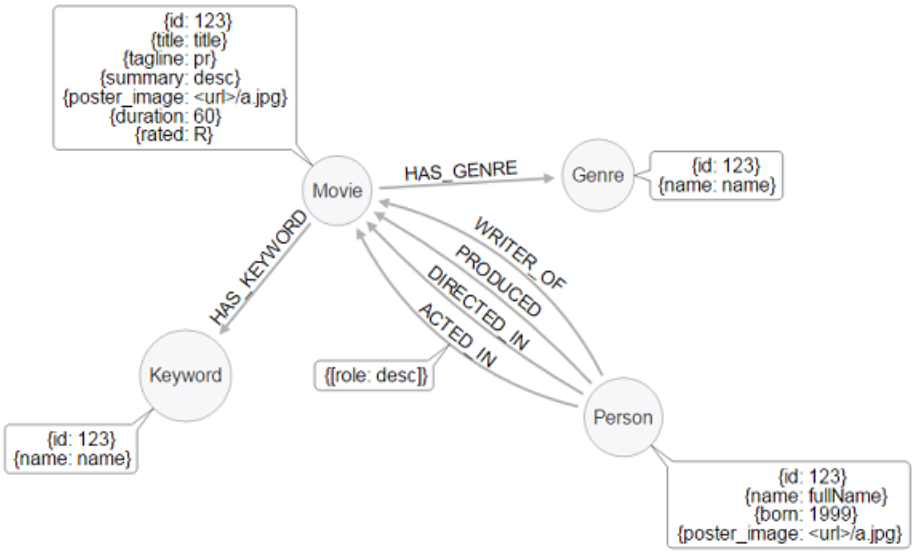

Figure 1.1: Conceptual diagram of a property graph for a movie database. Image source: `https://neo4j.com/blog/flask-react-js-developers-neo4j-template/`.

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE m.title CONTAINS 'Matrix'
WITH m.title AS movie_title, COUNT(*) AS number_of_actors
RETURN movie_title, number_of_actors
ORDER BY number_of_actors DESC
LIMIT 5;
```

Figure 1.2: Example Cypher query on the movie database. This query returns the number of actors that acted in each "Matrix" movie, sorted from most to least.

ease. Crucially, Cypher uses straightforward, intuitive syntax for expressing graph patterns in a way that is lacking in SQL.

Figure 1.2 shows a simple example of a Cypher query on the movie database described previously. In Cypher, the `MATCH` command is used for reading data and is analogous to SQL `SELECT`. The `MATCH` keyword introduces a clause that contains a graph pattern in *pattern syntax*. Pattern syntax is the workhorse of any Cypher query; it describes a configuration of nodes, relationships, labels, and properties using intuitive "ASCII art" syntax. Because of its dual nature of both describing and defining graph structures, the same pattern syntax is used for both statements that query existing data (`MATCH`) and statements that create new data (`CREATE`).

The pattern in a `MATCH` clause declaratively describes all instances of nodes and relationships to which the query pertains. Patterns may contain any number of nodes and edges and are not restricted to the simple single-edge pattern show in Figure 1.2. The pattern can bind matched structures to variable names that can be referenced later in the query. Unlike SQL, the flow of execution and variable scope in a Cypher query conceptually proceeds from the top of the query to the bottom, rather than from bottom to top.

In pattern syntax, nodes are enclosed in `()` characters and begin with an optional variable name, followed by any number of node labels preceded with `:` symbols. Edges are represented

using ASCII arrows (`-->`) between two nodes. For convenience, the arrow may point in either direction or may be undirected by omitting the arrowhead. A relationship type may optionally be inserted into the middle of the relationship, enclosed in `[]` characters. Like nodes, the body of the relationship type may optionally include a variable name and any number of relationship type labels followed by `:` characters. Relationships may also include alternations separated by `|` characters; for example, the pattern `(p:Person)-[:ACTED_IN|:DIRECTED_IN]->(m:Movie)` would match all instances of acting in or directing a movie.

Within the bodies of both nodes and relationships, after the optional variable and labels, matching properties may also be specified, enclosed in {} symbols. For instance, the pattern `(n:Person name: 'Keanu Reeves')` would match all people with the name "Keanu Reeves." Finally, paths of variable length may be specified in `MATCH` patterns using the `*` character. For example, `(n)-[*1..5]->(m)` matches all paths connecting two nodes that are between one and five relationships apart. For performance purposes, it is recommended always to specify an upper limit on path length in order to avoid unexpectedly long graph traversals.

Following the `MATCH` clause is an optional `WHERE` clause which further restricts the set of data to which the query pertains. It is analogous to the same keyword in SQL. The body of the `WHERE` clause is a predicate, represented as a Boolean expression, which only allows data to pass through when the expression evaluates to true. It may reference variables and properties of variables declared in the `MATCH` clause.

An optional `WITH` clause serves the dual purpose of assigning expressions to intermediate variables and computing aggregate functions such as `COUNT`. It structurally separates different parts of the query. Although not shown, after the `WITH` clause, additional `MATCH-WHERE` clauses may be stacked on top of each other in order to filter results further.

The `RETURN` clause simply determines the values that are returned to the client that invoked the query. In Figure 1.2, the query returns the title of each Matrix movie as well as the number of actors in it. `ORDER BY` and `LIMIT` clauses, which determine the ordering and maximum number of records returned, are analogous to their roles in SQL.

Although we have covered the most salient features of Cypher's query syntax, Cypher also includes an extensive set of commands for creating, updating, and deleting data. A more comprehensive reference for the Cypher Query Language may be found at the Neo4j website.[3]

1.4 Key Graph Primitives

One of the defining features of Neo4j is its treatment of nodes and relationships as first-class citizens. Neo4j operates with a minimal set of primitive entities, yet is powerful enough to express real-world graph-structured data. We briefly review the key graph primitives described in Section 1.2.

Node A graph vertex. Labeled with at least on node label which identifies its type, and labeled with any number of properties. Node labels and properties may be used in queries.

Relationship A directed graph edge connecting one node to another. Labeled with exactly one relationship type. May be labeled with any number of properties. Multiple relationships of different types may connect the same two nodes. Relationship types and properties may be used in queries.

Property A key-value pair associated with either a node or relationship. The key must be a string. The value may have a variety of data types supported by Neo4j.

[3] `https://neo4j.com/docs/cypher-refcard/current/`

Cypher queries also utilize a first-class citizen path data type.

1.5 Execution Model

The execution of a Neo4j query begins with execution planning, whereby a Cypher query is translated into a tree-like structure called an *execution plan*. Each node in the tree represents a low-level record-wise operator such as `Sort` or `Limit`.[4] Each operator accepts a list of records as input and produces a list of records as output. Child operators pipe their results into their parent operators. Some operators, like `Limit`, execute lazily in that they do not require child operators to produce the entirety of their output. The leaves of the execution plan tree correspond to operations which interact directly with the data in the database. The execution planner should make efficient use of indexes in order to speed up queries.

The execution engine of Neo4j differs from that of traditional RDBMSs in that it does not rely on global indexes. Instead, it takes a graph-centric approach known as "graph native processing" or "index-free adjacency."[5] Index-free adjacency speeds up graph queries by storing each node directly with its adjacent nodes and relationships, an arrangement that serves as a kind of micro-index for nearby nodes. This node-centric locality means that graph traversals be executed quickly without the need for global indexes whose computational cost would grow proportionally to the size of the entire graph.

Like RDBMSs, Neo4j also adopts the ACID [1] consistency model for write operations.

The Enterprise Edition of Neo4j includes a "Causal Clustering" feature which supports massively parallel graph queries. Causal Clusters make use of a multi-machine architecture that replicates graph data to *replica servers* and keeps synchronized data on a small set of *core servers*, as shown in Figure 1.3. The purpose of the core servers is to safeguard and synchronize data, whereas the purpose of replica servers is to scale out graph workloads (including Cypher queries). Core servers replicate all transactions using a special protocol called Raft, which requires a consensus of core servers before accepting writes. However, the number of core servers must be limited in order to keep write latency reasonable. Read replicas serve as caches for the data on the core servers and contain their own Neo4j execution engines that can execute parts of queries in parallel. Core servers asynchronously replicate data to the replica servers. With fault-tolerance in mind, core servers can also become replica servers if they experience too many failures. Within this architecture, write operations from the client are exclusively handled by the core servers, whereas replica servers handle read-only queries.

An open-source library of common graph algorithms called `algo`[6] is also available, containing implementations of algorithms such such as A* search and Jaccard similarity. These algorithms are exposed as callable Cypher procedures and are hand-written in Java to maximize local CPU and I/O utilization.

1.6 Examples

Let us continue the running movie database example introduced previously. The following query will return all actors with the name "Tom Hanks."

[4] See `https://neo4j.com/docs/developer-manual/current/cypher/execution-plans/` for a comprehensive list of operators.

[5] A more detailed introduction to this topic may be found at `https://neo4j.com/blog/native-vs-non-native-graph-technology/`.

[6] `https://github.com/neo4j-contrib/neo4j-graph-algorithms`

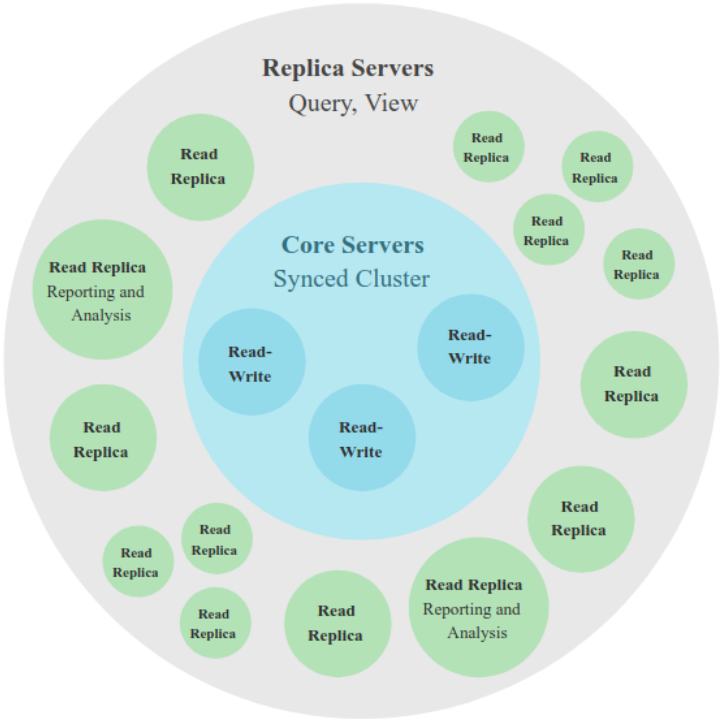

Figure 1.3: Architecture diagram of a Neo4j Causal Cluster. Image source: `https://neo4j.com/docs/operations-manual/current/clustering/introduction/`.

Figure 1.4: Consistency model of a Neo4j Causal Cluster. Image source: `https://neo4j.com/docs/operations-manual/current/clustering/introduction/`.

```
MATCH (tom {name: "Tom Hanks"}) RETURN tom
```

The result of this query is a single record:

"tom"
{"name":"Tom Hanks","born":1956}
{"name":"Tom Hanks","born":1956}

The next query will return all movies released in the 1990's:

```
MATCH (nineties:Movie)
WHERE nineties.released >= 1990 AND nineties.released < 2000
RETURN nineties.title
```

The (truncated) results of this query might be:

"nineties.title"
"The Matrix"
"The Devil's Advocate"
"A Few Good Men"

This query will return all movies and actors within four hops of actor Kevin Bacon:

```
MATCH (bacon:Person {name:"Kevin Bacon"})-[*1..4]-(hollywood)
RETURN DISTINCT hollywood
```

"hollywood"
{"name":"Robin Williams","born":1951}
{"name":"Chris Columbus","born":1958}
{"title":"Bicentennial Man","tagline":"One robot's 200 year journey to become an ordinary man.","released":1999}

Finally, this query will find co-co-actors who have not worked with Tom Hanks. The query scores each co-co-actor with a "Strength" corresponding to the number of times the actor or actress appeared with a Tom Hanks co-actor. Note that multiple edges are specified in the `MATCH` clause by separating them with a comma.

```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
      (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors)
WHERE NOT (tom)-[:ACTED_IN]->()<-[:ACTED_IN]-(cocoActors) AND tom <> cocoActors
RETURN cocoActors.name AS Recommended, count(*) AS Strength ORDER BY Strength DESC
```

"Recommended"	"Strength"
"Tom Cruise"	10
"Zach Grenier"	10
"Cuba Gooding Jr."	8
"Keanu Reeves"	8
"Carrie Fisher"	6

A complete working example of a simple web application serving as a front-end to a movie database, implemented in multiple languages, is available on GitHub.[7]

1.7 Conclusion

Graph-structured data has become exceedingly common in an age of social networks and big data analytics. Neo4j offers an alternative to the pervasive relational database paradigm that is better suited to the graph abstraction, treating graph primitives as first-class citizens and supporting rich graph queries through its Cypher Query Language front-end. Massively parallel graph queries open up possibilities for sophisticated inference from graph data for organizations with the appropriate resources. Application designers would be well-advised to consider alternative database paradigms to match their data's most natural representation.

[7] `https://github.com/neo4j-examples`

Bibliography

[1] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15:287–317, 1983.