Graph Data Models and Neo4j: An Introduction

Briefing Document: Introduction to Graph Data Model and Neo4j

Date: October 26, 2023

Prepared For: Interested Parties

Subject: Review of Graph Data Models and the Neo4j Graph Database System

This briefing document summarizes the key concepts and information presented in the provided excerpts from "Introduction to Graph Data Model.pdf" and "Neo4j.pdf". It covers the fundamentals of graph data models, their applications, common graph algorithms, and an introduction to the Neo4j graph database system, including its query language (Cypher) and setup using Docker Compose.

1. Graph Data Model Fundamentals

The foundation of a graph database is the **graph data model**. This model is structured around:

- **Nodes (Vertices):** These represent entities and are uniquely identified. They can contain **properties**, which are key-value pairs that describe the node (e.g., a person node might have properties like "name" and "occupation"). According to the source, "Composed of a set of node (vertex) objects... Labels are used to mark a node as part of a group - Properties are attributes (think KV pairs) and can exist on nodes..."
- **Edges (Relationships):** These connect nodes and represent the relationships between them. Like nodes, edges can also have **properties** (e.g., a "KNOWS" relationship between two people might have a "since" property indicating when they met). Importantly, the source states, "...relationship (edge) objects... Properties are attributes (think KV pairs) and can exist on... relationships - Edges not connected to nodes are not permitted."
- **Labels:** Nodes can have labels to categorize them into groups (e.g., "person," "car"). The example provided shows "2 Labels: - person - car".
- **Relationship Types:** Edges have types that define the nature of the connection between nodes (e.g., "Drives," "Owns," "Lives_with," "Married_to"). The example lists "4 relationship types: - Drives - Owns - Lives_with - Married_to".

Paths are a fundamental concept in graph databases, defined as "an ordered sequence of nodes connected by edges in which no nodes or edges are repeated." The example illustrates valid and invalid paths.

1.1. Flavors of Graphs

Graphs can exhibit various characteristics:

- **Connected vs. Disconnected:** A connected graph has a path between any two nodes, while a disconnected graph does not.
- **Weighted vs. Unweighted:** In a weighted graph, edges have an associated weight property, which is crucial for certain algorithms.

Directed vs. Undirected: Directed graphs have edges that define a start and end node, indicating a one-way relationship. Undirected graphs have edges where the relationship is bidirectional. The Neo4j excerpt later notes, "Note: Relationships are directed in neo4j."

•

Acyclic vs. Cyclic: An acyclic graph contains no cycles (a path that starts and ends at the same node without repeating edges).

•

Sparse vs. Dense: This refers to the number of edges relative to the number of nodes.

•

Trees: A specific type of connected, acyclic graph.

1.2. Where Graphs Show Up

Graph data models are applicable in numerous domains:

•

Social Networks: Modeling users and their connections (e.g., "things like Instagram"). This also extends to modeling social interactions in psychology and sociology.

•

The Web: The internet can be viewed as a large graph of web pages (nodes) connected by hyperlinks (edges).

•

Chemical and Biological Data: Representing interactions and relationships in systems biology, genetics, and chemistry.

2. Types of Graph Algorithms

The introduction highlights several important categories of graph algorithms:

2.1. Pathfinding

•

The most common operation is finding the **shortest path** between two nodes, measured by the fewest edges or the lowest weight. The excerpt notes, "- Pathfinding - finding the shortest path between two nodes, if one exists, is probably the most common operation - 'shortest' means fewest edges or lowest weight".

•

Average Shortest Path can be used to assess network efficiency and resilience.

•

Other pathfinding algorithms include **minimum spanning tree**, **cycle detection**, and **max/min flow**.

•

Two fundamental search algorithms mentioned are **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**.

2.2. Centrality & Community Detection

•

Centrality algorithms aim to identify the most "important" nodes within a network. An example given is identifying "Social Network Influencers?".

•

Community Detection focuses on identifying clusters or partitions of nodes that are more strongly connected within the group than to nodes outside it.

2.3. Famous Graph Algorithms

- **Dijkstra’s Algorithm:** A single-source shortest path algorithm suitable for graphs with positively weighted edges.
- **A Algorithm:** Similar to Dijkstra's but incorporates a heuristic to guide the traversal, potentially improving efficiency.
- **PageRank:** An algorithm that measures the importance of nodes based on the number and importance of incoming relationships. The source explains it as measuring importance "based on the number of incoming relationships and the importance of the nodes from those incoming relationships".

3. Neo4j: A Graph Database System

Neo4j is presented as "A Graph Database System that supports both transactional and analytical processing of graph-based data." It is categorized as a relatively new class of NoSQL databases and is considered "schema optional" (though a schema can be imposed). Key features mentioned include:

- Support for various types of indexing.
- ACID (Atomicity, Consistency, Isolation, Durability) compliance, ensuring data integrity.
- Support for distributed computing.
- Similar systems include Microsoft CosmoDB and Amazon Neptune.

3.1. Query Language and Plugins

- **Cypher:** Neo4j's graph query language, created in 2011, aims to be an "SQL-equivalent language for graph databases." It is designed to provide a visual way of matching patterns and relationships, exemplified by the syntax `(nodes)-[:CONNECT_TO]->(otherNodes).`
- **APOC (Awesome Procedures on Cypher) Plugin:** An add-on library offering a wide range of procedures and functions to extend Cypher's capabilities.
- **Graph Data Science Plugin:** Provides efficient implementations of common graph algorithms, aligning with the algorithms discussed earlier.

3.2. Neo4j in Docker Compose

Docker Compose is introduced as a tool for managing multi-container Docker applications using a declarative YAML file (`docker-compose.yaml`). Its benefits include:

- Simplified management of multiple services.
- Consistent environment creation, addressing the "well… it works on my machine!" problem.
-

Interaction primarily through the command line.
The example `docker-compose.yaml` file for Neo4j demonstrates how to define a Neo4j service, including:

- `container_name`: Setting the name of the Docker container.
- `image`: Specifying the Neo4j Docker image to use (`neo4j:latest`).
- `ports`: Mapping host ports (7474 for the Neo4j Browser and 7687 for Bolt protocol) to container ports.
- `environment`: Setting environment variables, such as the Neo4j authentication credentials (`NEO4J_AUTH`), and enabling the APOC and Graph Data Science plugins (`NEO4J_PLUGINS=["apoc", "graph-data-science"]`).
- `volumes`: Mounting host directories to container directories for persistent data (`./neo4j_db/data:/data`), logs (`./neo4j_db/logs:/logs`), import files (`./neo4j_db/import:/var/lib/neo4j/import`), and plugins (`./neo4j_db/plugins:/plugins`).

The document emphasizes the importance of not storing secrets directly in the `docker-compose.yaml` file and recommends using **.env files** to manage environment variables for different environments (e.g., `.env.local`, `.env.dev`, `.env.prod`). An example `.env` file shows how to define `NEO4J_PASSWORD`.
Common Docker Compose commands are listed, including `docker compose up`, `docker compose down`, `docker compose start`, `docker compose stop`, and `docker compose build`.
3.3. Interacting with Neo4j
The Neo4j Browser is accessible at `localhost:7474` after the Neo4j instance is running. Users can log in using the configured credentials.
The document provides examples of **inserting data** by creating nodes with properties and labels using the `CREATE` command in Cypher. For example:
CREATE (:User {name: "Alice", birthPlace: "Paris"})

It also demonstrates how to **add edges** (relationships) between existing nodes using `MATCH` to find the nodes and `CREATE` to define the relationship:
MATCH (alice:User {name:”Alice”})
MATCH (bob:User {name: “Bob”})
CREATE (alice)-[:KNOWS {since: “2022-12-01”}]->(bob)

The directionality of relationships in Neo4j is explicitly mentioned: "Note: Relationships are directed in neo4j."

An example of **matching nodes** based on properties using the `MATCH` and `RETURN` commands is provided:

```
MATCH (usr:User {birthPlace: “London”})
RETURN usr.name, usr.birthPlace
```

3.4. Importing Data into Neo4j

The document outlines the process of importing data from CSV files:

1.
Download Dataset and Move to Import Folder: This involves cloning a repository, unzipping the data, and copying the CSV file (e.g., `netflix_titles.csv`) to the `neo4j_db/neo4j_db/import` directory (relative to the `docker-compose.yaml` file).

2.
Basic Data Importing: The `LOAD CSV WITH HEADERS` command is used to read data from the CSV file and create nodes. For example, to import movies:

```
LOAD CSV WITH HEADERS FROM 'file:///netflix_titles.csv' AS line
CREATE(:Movie {
   id: line.show_id,
   title: line.title,
   releaseYear: line.release_year
 } )
```

3.
Loading CSVs - General Syntax: The general syntax for loading CSV files is provided, including the optional `WITH HEADERS` clause and the `FIELDTERMINATOR` option.

4.
Importing with Relationships (Directors Example): The document demonstrates how to import data that involves relationships. This includes:

- Splitting comma-separated values (e.g., directors) into lists using `split()`.
- Using `UNWIND` to iterate over the list of directors.
- Initially creating `Person` nodes for each director (which can lead to duplicates).
- Using `MERGE` instead of `CREATE` to ensure that `Person` nodes are only created if they don't already exist, thus avoiding duplicates.
- Creating `DIRECTED` relationships between `Person` nodes (directors) and `Movie` nodes using `MATCH` to find the respective nodes and `CREATE` to establish the relationship. An example is given:

5.
Gut Check: Finally, a query is provided to verify the imported data and relationships:

```
MATCH (m:Movie {title: "Ray"})<-[:DIRECTED]-(p:Person)
RETURN m, p
```

This query finds the movie with the title "Ray" and the directors who have a `DIRECTED` relationship to it, returning the movie and director nodes.

This briefing document provides a comprehensive overview of the fundamental concepts of graph data models, common graph algorithms, and an introduction to the Neo4j graph database system, including its query language, setup using Docker Compose, and basic data manipulation techniques.