Neo4j Performance Improvement: A Structured Approach for Base and Incremental Loads

Authors: Arpit Jain, Pragati Sharma, Raghav Sharma

Executive Summary

This whitepaper explores the challenges of importing large datasets into Neo4j, particularly when relying on methods like one-by-one ingestion or CSV loading, which can be time-intensive. It focuses on strategies to streamline the loading process for both base and incremental loads. Additionally, the paper addresses the performance bottlenecks that occur when data is already present in the database, further slowing down the load process. By incorporating structured pre-processing techniques—such as data cleansing, deterministic and probabilistic matching, and leveraging Neo4j's import tools—the proposed approach significantly accelerates data ingestion. This results in faster and more efficient loading of large datasets into the graph database, especially when using a base load or a hybrid approach for incremental loads in the Neo4j community edition.

Introduction

Neo4j is a powerful graph database known for its ability to represent relationships between data points, especially when working with large and complex datasets. However, as data volumes grow, traditional methods like one-by-one ingestion or CSV loading can significantly slow down the import process, especially when data is already present in the database. This whitepaper addresses the challenges of efficiently loading large datasets into Neo4j community edition, both for base and incremental loads. It outlines strategies to optimize data ingestion, transformation, and relationship creation, with a particular focus on identity resolution to improve load times and ensure consistent performance.

Need for this Approach

Organizations are increasingly adopting graph databases like Neo4j to manage complex relationships between data points. While graph databases offer flexibility and powerful relationship modeling, large-scale data ingestion—especially when using traditional methods like one-by-one ingestion or CSV loading—can result in significant delays and inefficiencies. As data volumes grow, maintaining performance during both base and incremental data loads becomes a challenge, particularly when data already exists in the database.

Our experience underscores these challenges. Before implementing the approach outlined in this whitepaper, we encountered indefinite times for loading incremental data into the graph database. This issue was especially problematic when dealing with large datasets, leading to delays, data inconsistencies, and operational inefficiencies.

The key challenges we faced—and which this approach addresses—include:

1. Handling Large Datasets

As data volumes increase, managing both base and incremental loading efficiently becomes a major bottleneck. We were dealing with datasets in the millions and traditional loading methods for incremental data were highly inefficient. The load time was indefinite, which hindered our ability to update the graph in real-time or at regular intervals.
By leveraging structured pre-processing and Neo4j's bulk import tools, we drastically reduced the time needed for incremental data loads. For example, after optimization, it now takes only about 45 minutes to merge 1000 new persons into a prepopulated database with 15 lakh nodes, compared to the previous indefinite load times.

2. Operational Efficiency

Regular updates to the graph database are essential for many organizations, making incremental data loading a critical process. However, traditional incremental load methods were slow and resource-intensive, hampering our ability to scale operations.
With the new approach, we optimized incremental loading through structured pre-processing, use of a backup database for merging, and efficient Cypher queries for relationship creation. As a result, the incremental load process became significantly faster, allowing more frequent updates to the database without impacting performance.

3. Lack of Efficient Merge Operations

Merging new data, particularly when creating relationships between nodes or maintaining referential integrity, can be a slow and complicated process, especially with large datasets. Prior to implementing our approach, merge operations in Neo4j were excessively time-consuming.
After optimization, we streamlined merge operations by leveraging Neo4j's bulk import tools, Cypher procedures, and APOC for batch processing. This greatly improved merge efficiency, enabling faster updates to the graph.

Solution

Our Research for IDR (Identity Resolution)

Identity resolution is the process of determining whether customers, sourced from different or the same platforms, represent the same individual. In our approach, we used Neo4j to store the final output, creating a separate **Person** node for each user. When customers are determined to be **probabilistically matched**, a **match relationship** is created between their person nodes. For **deterministic matches**, the individual **Person** nodes are merged into a single node to consolidate the data. Neo4j is ideal for this approach due to its ability to efficiently manage complex relationships and perform fast graph traversals, making it perfect for handling large-scale identity resolution tasks.

Base Load Process

1) Cleansing: The first step in the base load process involves data cleansing, where raw data files are cleaned and prepared for import into Neo4j. This step is essential for ensuring data integrity and consistency. The cleansing process includes:

- **Removing Duplicate Records:** Identifying and eliminating any redundant data entries, ensuring each record is unique.
- **Eliminating Special Characters:** Special characters that might cause issues during import, such as non-UTF-8 characters or unexpected symbols, are removed or standardized.
- **Standardizing Data:** This includes standardizing formats (e.g., date formats, phone numbers, address formats) and ensuring that field values follow consistent naming conventions (e.g., "New York" vs. "NYC").

The goal of this step is to produce a clean dataset that is free of inconsistencies, which is crucial for smooth ingestion and for maintaining the quality of the graph once data is loaded into Neo4j.

2) Pre-Ingestion Preparation: After data cleansing, the next step is pre-ingestion preparation. In this phase, a job is run to generate the necessary CSV files for nodes and relationships. The pre-ingestion job does the following:

- It takes the processed data from the cleansing step and organizes it into CSV files, which are the format required by Neo4j's import tools.
- Separate CSV files are created for each type of entity (nodes), such as person.csv, address.csv, phone.csv, etc., and for each relationship type (e.g., relationship.csv).

These CSV files are structured to ensure that Neo4j can correctly interpret and ingest the data, where each row represents a data entry and columns represent various attributes or relationships between entities.

This step prepares the data in a format that can be efficiently imported into the Neo4j database.

3) Neo4j Import: The final step in the base load process involves using Neo4j's neo4j-admin tool to import the prepared CSV files into the Neo4j database. This step performs the actual data ingestion. Here's a breakdown of this process:

- **Command Execution:** The neo4j-admin database import command is executed, which is responsible for ingesting the data into the database. This tool is optimized for bulk import and handles large datasets efficiently.
- **Node Import:** CSV files for each node type (e.g., person.csv, address.csv, phone.csv) are ingested. Each row in the CSV represents a specific data entity (like a person, phone, or address) in the graph database.
- **Relationship Import:** CSV files for relationships (e.g., relationship.csv, relationship_name.csv, relationship_match.csv) are ingested. These files represent the connections between the nodes (e.g., a person "lives at" an address, or a person "has a phone number").

The neo4j-admin database import command looks something like this:

bin/neo4j-admin database import full --overwrite-destination neo4j --nodes=import/person.csv --nodes=import/touch.csv --nodes=import/persist.csv --nodes=import/name.csv --

nodes=import/address.csv --nodes=import/phone.csv --nodes=import/email.csv --relationships=import/relationship.csv --relationships=import/relationship_add.csv --relationships=import/relationship_name.csv --relationships=import/relationship_match.csv --relationships=import/relationship_ph.csv --relationships=import/relationship_touch.csv --verbose

Incremental Load Process

1) **Data Cleansing:**
As with the base load, the same data cleansing steps are applied to ensure consistency and remove any duplicates, special characters, or inconsistencies in the incoming incremental data.

2) **Deterministic Matching:**
Exact matches are identified using predefined attributes (e.g., name, email, address) to resolve identities and prevent duplication.

3) **Probabilistic Matching:**
Similar, but non-exact matches are detected based on attribute similarity, helping to merge records that likely refer to the same entity.

4) **Pre-Ingestion Preparation:**
In this step, files are prepared for ingestion into Neo4j using the neo4j-admin tool:

a) The output data is split into three categories based on the type of load:
- **New Records:** Data that does not yet exist in the database and needs to be added.
- **Merge Records:** Existing data that needs to be updated or merged with new information.
- **Probabilistic records:** Person nodes to be linked to showcase probabilistic matching

b) **Node Files:** Separate CSV files are created for each attribute (e.g., person, email, phone, address, store address, touchpoint).

c) **Relationship Files:** Relationship CSV files are generated to link person nodes to their corresponding attributes (e.g., person-email, person-phone), and also to create probabilistic links between person nodes identified as matches.

5) **Backup Database Ingestion:**
The new and probabilistic records (excluding match relationships) are ingested into a backup database using Neo4j's neo4j-admin import tool.

6) **Cypher Extract:**
A Cypher extract is generated from the backup database using the following command:
CALL apoc.export.cypher.all('exported_data1.cypher')

7) **Merge Data into Main Database:**

The Cypher file (exported_data1.cypher) is copied to the main Neo4j import folder and executed with after which we get all the new records in the main database:
bin/cypher-shell -u neo4j -p password -d neo4j -f import/exported_data1.cypher

8) Match Relationships:

Create Index: An index is created on the persistentID field for the Person node to improve match relationship performance:

```
CREATE INDEX index_name FOR (n:Person) ON (n.persistentID);
```

Batch Relationship Creation: Use probabilistic match file to create match relationships between nodes.The APOC periodic.iterate function is used to create match relationships in batches:

```
CALL apoc.periodic.iterate(
  'LOAD CSV WITH HEADERS FROM "file:///relationship_match.csv" AS row RETURN row',
  'MATCH (a:Person {persistentID: row.`:START_ID`}), (b:Person {persistentID: row.`:END_ID`})
   WITH a, b, row,
   [key IN keys(row) WHERE key <> ":START_ID" AND key <> ":END_ID" AND key <> ":TYPE" |
[key, row[key]]] AS entries
   CALL apoc.create.relationship(a, row.`:TYPE`, apoc.map.fromPairs(entries), b) YIELD rel RETURN
rel',
  {batchSize: 1000}
);
```

9) Extract for Merge Type Persistent IDs (will be used for updates):

Extract persistent IDs for merge type records using this query:

```
CALL apoc.export.csv.query(
  'MATCH (p:Person)
   WHERE p.persistentId IN $persistentIDs
   OPTIONAL MATCH (p)-[r]->(o)
   WHERE type(r) <> "matches"
   RETURN p.persistentID, p, r, o',
  'output.csv',
  {params: {persistentIDs: []}} // Replace with your list of persistentIDs
);
```

Merge Records Processing:

Use APOC procedures to ingest updates in batches. Removing those properties value of which is N.A from nodes and edges which got created during time of ingestion for merge type files.

```
CALL apoc.periodic.iterate(
  "MATCH (n)
   WHERE any(key IN keys(n) WHERE n[key] = 'N.A')",
  "SET n += apoc.map.removeKeys(n, [key IN keys(n) WHERE n[key] = 'N.A'])",
  {batchSize: 1000, parallel: false}
);
CALL apoc.periodic.iterate(
  "MATCH ()-[r:matches]->()
   WHERE any(key IN keys(r) WHERE r[key] = 'N.A')",
  "SET r += apoc.map.removeKeys(r, [key IN keys(r) WHERE r[key] = 'N.A'])",
  {batchSize: 1000, parallel: false}
);
```

Configuration Changes

Modify the neo4j.conf to allow unrestricted APOC procedures and file export functionality:

```
dbms.security.procedures.unrestricted=jwt.security.*,apoc.*

dbms.security.procedures.allowlist=apoc.*,gds.*

apoc.export.file.enabled=true

apoc.export.file.directory=/dbdump

dbms.security.procedures.allowFileUrls=true
```

Analysis of Time Taken

Metric	Earlier Value	Current Value	Notes
Total Time Taken in Base Load	4 days	2-3 minutes	Base load time was significantly reduced from 4 days to 2-3 minutes.
Total Persons Ingested in Base Load	18 lakh (1.8 million)	18 lakh (1.8 million)	Number of persons ingested remains the same for base load.
Total Nodes Ingested in Base Load	75 lakh (7.5 million)	75 lakh (7.5 million)	Number of nodes ingested remains the same for base load.

Metric	Earlier Value	Current Value	Notes
Time Taken to Merge 1000 Person Nodes	Not applicable	~45 minutes	Time to merge 1000 nodes now takes ~45 minutes, earlier not defined.
Total Time Taken for Complete Incremental Ingestion	8 days	~5.5 hours	Incremental ingestion time reduced from 8 days to ~5.5 hours.
Performance Improvement (Base Load)	4 days	2-3 minutes	Base load time drastically improved from 4 days to 2-3 minutes.
Performance Improvement (Incremental Load)	8 days	~5.5 hours	Incremental load time significantly improved, reducing from 8 days to ~5.5 hours.

Key Insights:

- Base Load Performance Improvement: Previously, it took 4 days to complete the base load for 18 lakh persons. With the new approach, this has been reduced to just 2-3 minutes, significantly improving the efficiency.
- Incremental Load Performance Improvement: The time to complete the incremental load for the same dataset (18 lakh persons) was previously taking 8 days. Now, it takes only ~5.5 hours, resulting in a major reduction in processing time.
- Merging Performance: Merging 1000 nodes (persons) that were processed during incremental load now takes around 45 minutes, much faster compared to earlier where it was an indefinite and inefficient process.

This table highlights the tremendous improvement in performance for both base and incremental loads. The optimization has reduced processing time by more than 99% for base load and around 30% for incremental load.

Applications

This methodology is well-suited for organizations looking to:

- **Integrate and manage large-scale datasets** from multiple sources into Neo4j.
- **Scale graph databases** efficiently by optimizing both base and incremental loads for large datasets.

Conclusion

By adopting these best practices, organizations can significantly enhance the performance of their Neo4j community databases, especially for large-scale data ingestion and identity resolution tasks. With optimized import processes, efficient use of Neo4j tools, and batch processing, this approach allows for faster and more scalable data management, making Neo4j an even more powerful tool for handling complex, relational data.