Lab 2: Import Data to Neo4j

Building Your Knowledge Graph

Concepts Introduced

Neo4j Spark Connector - A library that enables bidirectional data transfer between Spark DataFrames and Neo4j graphs.

Key Concepts:

- **Nodes** - Entities in the graph (Customer, Account, Stock)
- **Relationships** - Connections between nodes (HAS_ACCOUNT, AT_BANK)
- **Properties** - Key-value attributes on nodes and relationships
- **Labels** - Categories that classify nodes (like table names)

Graph vs Relational:

- Relationships are first-class citizens, not foreign keys
- Traversing connections is O(1), not O(n) JOINs
- Schema-flexible: add properties without migrations

The Neo4j Spark Connector

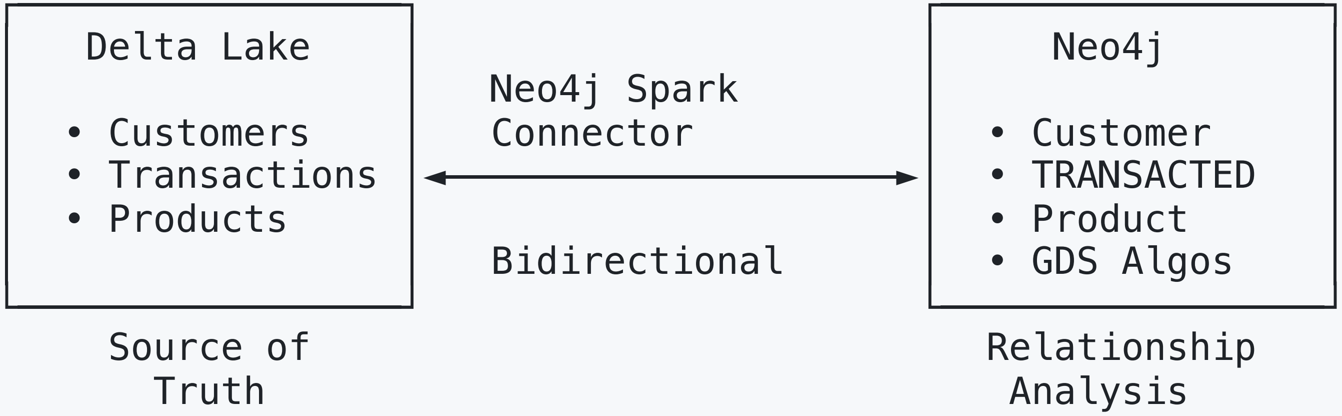
Bridges two data models:

Databricks (Tables)	Neo4j (Graph)
Rows become	Nodes
Foreign keys become	Relationships
Columns become	Properties

Why use both?

- Tables excel at aggregations and analytics
- Graphs excel at relationship traversals and pattern matching

The Bridge: Neo4j Spark Connector



Delta Lake

- Customers
- Transactions
- Products

Source of Truth

Neo4j Spark Connector

Bidirectional

Neo4j

- Customer
- TRANSACTED
- Product
- GDS Algos

Relationship Analysis

Data flows both ways through familiar DataFrame APIs.

What the Neo4j Spark Connector Enables

- **Read** from Neo4j into Spark DataFrames
- **Write** from DataFrames into Neo4j
- **Stream** for continuous synchronization
- **Query** GDS algorithms and get results as DataFrames

No custom ETL scripts. Native Spark integration.

The Key Insight: Graphs and Tables Are Interchangeable

Graphs and tables map bidirectionally:

```
Graph:                          Tables:

(Person)-[:HAS]->(Job)    ↔️     Person: [id, name, age]
                                Job: [id, title, salary]
                                HAS: [person_id, job_id, since]
```

- **Nodes** ↔️ rows (one table per label)
- **Relationships** ↔️ rows with source and target columns
- **Properties** ↔️ columns

This means you can both **export graphs to tables** and **import tables into graphs**.

What We're Creating

```
Financial Knowledge Graph

(Customer)-[:HAS_ACCOUNT]->(Account)-[:AT_BANK]->(Bank)
                              |
                     [:HAS_POSITION]
                              |
                              v
                         (Position)-[:OF_SECURITY]->(Stock)-[:OF_COMPANY]->(Company)
```

764 nodes across 7 types | **814 relationships** across 7 types

Key Steps

1. **Configure** - Connect Spark to Neo4j via secrets
2. **Create constraints** - Indexes before data load
3. **Write nodes** - DataFrames become graph nodes
4. **Write relationships** - Foreign keys become edges
5. **Validate** - Verify counts match expectations

Best Practice: Use `coalesce(1)` for relationship writes to prevent deadlocks.

Basic Node Write

```
df.write \
    .format("org.neo4j.spark.DataSource") \
    .option("url", "neo4j://localhost:7687") \
    .option("authentication.type", "basic") \
    .option("authentication.basic.username", "neo4j") \
    .option("authentication.basic.password", "password") \
    .option("labels", ":Customer") \
    .option("node.keys", "customer_id") \
    .mode("Overwrite") \
    .save()
```

Key options: `labels` defines node labels, `node.keys` defines uniqueness.

Save modes: `Overwrite` uses MERGE (upsert), `Append` uses CREATE (always inserts).

How MERGE Works

The write generates Cypher that merges rows in batches:

```
UNWIND $batch AS row
MERGE (n:Customer {customer_id: row.customer_id})
SET n.name = row.name, n.email = row.email, n.created_at = row.created_at
```

`UNWIND` unpacks the list, processing all rows in a single transaction.

Basic Relationship Write

```
df.write.format("org.neo4j.spark.DataSource") \
    .mode("Append") \
    .option("relationship", "HAS_ACCOUNT") \
    .option("relationship.save.strategy", "keys") \
    .option("relationship.source.save.mode", "Match") \
    .option("relationship.source.labels", ":Customer") \
    .option("relationship.source.node.keys", "customer_id:customer_id") \
    .option("relationship.target.save.mode", "Match") \
    .option("relationship.target.labels", ":Account") \
    .option("relationship.target.node.keys", "account_id:account_id") \
    .save()
```

Creates: `(:Customer {customer_id})-[:HAS_ACCOUNT]->(:Account {account_id})`

Key Settings:

- **`relationship`** — The relationship type to create (e.g., `HAS_ACCOUNT`)
- **`relationship.save.strategy: "keys"`** — Match existing nodes by key properties
- **`relationship.source/target.save.mode: "Match"`** — Find existing nodes (don't create new ones)
- **`relationship.source/target.node.keys`** — Map DataFrame columns to node keys (`df_col:node_prop`)

Next: Export graph data back to Delta Lake for AI agents