Advanced Java Programming Course

NEO4J

Faculty of Information Technologies
Industrial University of Ho Chi Minh City

Session objectives

- ✓ Understand Graph Databases: Nodes, relationships, and properties.
- ✓ Install, Set Up Neo4j, and Learn Cypher
- ✓ Build and Manage Graphs
- ✓ Query and Explore Data
- ✓ Explore Advanced Features: Learn about indexing, full-text search, and Neo4j Bloom for data visualization.

Introduction

Introduction to Graph Databases

- What is a Graph Database?
  - A graph database is designed to store and manage data whose relationships are as important as the data itself.
  - It uses graph structures: nodes, relationships, and properties to represent and store data.
- Why Graph Databases?
  - Unlike traditional relational databases, graph databases excel at handling highly connected data and complex queries.
  - Ideal for real-time analytics, recommendation engines, social networks, fraud detection, and more.

Core Concepts of a Graph Database

- Nodes
  - Entities or objects in the graph *(e.g., people, places, products).*
- Relationships
  - Connections between nodes that define how they are related *(e.g., "FRIENDS_WITH", "LIKES", "BOUGHT").*
- Properties
  - Key-value pairs attached to nodes and relationships *(e.g., a Person node may have properties like name and age).*

Graph vs. Relational Databases

- Relational Databases (RDBMS):
  - Use tables and rows to store data.
  - Relationships are represented by foreign keys.
  - JOIN operations are used to combine data from different tables.
- Graph Databases:
  - Use graphs *(nodes, relationships, and properties)* to store data.
  - Relationships are first-class citizens and can be traversed efficiently without JOINs.

Popular Graph Databases

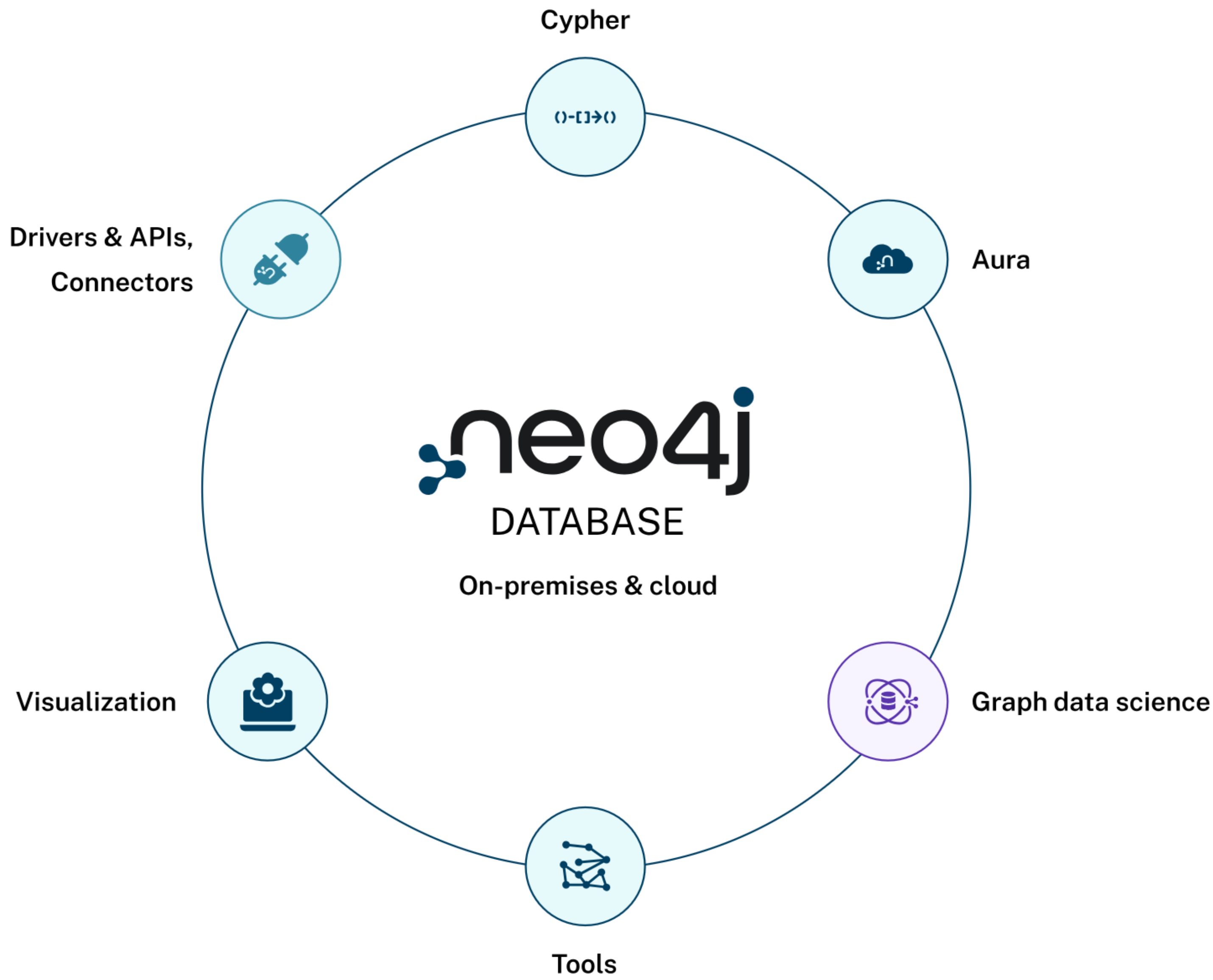
Some of the most popular graph databases currently available:

- **Neo4j**: A native graph database that is known for its high performance and scalability.
- **Amazon Neptune:** A managed graph database service from Amazon Web Services (AWS) that supports both property graph and RDF data models.
- **JanusGraph:** A scalable graph database that supports a variety of storage backends, including Cassandra, HBase, and Elasticsearch.
- **OrientDB:** A multi-model database that supports both graph and document data models.
- **ArangoDB:** Another multi-model database that supports graph, document, and key/value data models.

Introduction to Neo4j

- What is Neo4j?
  - Neo4j is a powerful, open-source graph database management system.
  - Neo4j is a highly scalable graph database that leverages graph theory to store, query, and process data.
  - It uses nodes, relationships, and properties to model data.
- Why use Neo4j?
  - Efficient handling of complex relationships between data points.
  - Graph-based query language *(Cypher)* makes it easier to work with data in a graph structure.

What is Neo4j?



Getting Started with Neo4j

- Installation
  - Neo4j locally or use the Neo4j Aura cloud service.
  - Available on different platforms: Windows, macOS, and Linux.
- Graph Data Model
  - Nodes: Entities (e.g., Person, Product).
  - Relationships: Connections between entities (e.g., "LIKES", "FRIENDS_WITH").
  - Properties: Data attached to nodes and relationships.

Key Concepts of Neo4j

- Nodes & Relationships
  - Nodes represent entities.
  - Relationships connect nodes and define how they relate to each other.
- Labels and Types
  - Labels classify nodes (e.g., :Person, :Movie).
  - Relationship types define the kind of relationship (e.g., :LIKES, :FRIENDS_WITH).
- Properties
  - Properties are key-value pairs attached to nodes or relationships (e.g., a Person node may have name and age properties).

Cypher Query Language

- What is Cypher?
  - Cypher is the query language for Neo4j.
  - It's designed to work seamlessly with graph data models.
- Basic Cypher Syntax
  - Match nodes and relationships using the MATCH keyword.
  - Example query: MATCH (n:Person) WHERE n.name = 'Alice' RETURN n.
- Using Clauses:
  - MATCH: Find patterns in the graph.
  - WHERE: Filter data.
  - RETURN: Return data.
  - CREATE: Add new nodes or relationships.

Interacting with Neo4j: Tools & Interfaces

- Neo4j Browser
  - The web-based interface to interact with Neo4j databases.
  - Allows running Cypher queries, viewing results, and visualizing graph data.
- Neo4j Desktop
  - A local development environment for managing Neo4j databases.
  - Includes built-in tools to run queries and visualize the data.
- Neo4j Aura (Cloud)
  - Cloud-based version of Neo4j, offering easy access without local installation.
  - Offers both free and paid plans depending on your needs.

Advantages of Neo4j Graph Database

- Efficient Relationship Handling
  - Directly stores relationships, making traversals fast and efficient.
- Flexible Data Model
  - The schema-less nature of Neo4j allows for flexibility in how data is represented and related.
- Powerful Query Language (Cypher)
  - Cypher is intuitive and optimized for querying graph structures.

Key Features of Neo4j

- ACID Compliant
  - Neo4j ensures data integrity through atomicity, consistency, isolation, and durability.
- High Performance
  - Optimized for fast traversals and real-time analytics.
- Scalability
  - Supports horizontal scaling, clustering, and replication for enterprise use cases.

When to Use a Graph Database

- Use Cases for Graph Databases:
  - Social Networks: Representing relationships between users.
  - Recommendation Engines: Recommending products or content based on relationships.
  - Fraud Detection: Identifying suspicious patterns in financial transactions.
  - Network Analysis: Analyzing communication patterns, infrastructure, etc.
- Example: Social Network
  - Nodes: Users
  - Relationships: "FRIENDS_WITH", "LIKES", "POSTED"
  - Properties: Name, Age, Interests, etc.

Best Practices for Working with Neo4j

- Designing the Graph Model
  - Keep your model simple, and design relationships based on how your data interacts.
- Optimizing Queries
  - Use indexes to speed up query performance, especially on high-cardinality properties.
- Scaling Neo4j
  - Consider using Neo4j clusters for horizontal scaling in production environments.

Getting Started with Neo4j

- Install Neo4j:
  - Download and install Neo4j locally, or use Neo4j Aura (cloud).
- Neo4j Browser
  - A web-based interface to query and visualize graphs.
- Neo4j Desktop
  - A local development environment for working with Neo4j databases.
- Neo4j Aura
  - A fully managed cloud service that provides an easy-to-use platform for building and deploying graph-based applications.

Interacting with Neo4j

- Running Cypher Queries:
  - Create data in Neo4j
    - Example: CREATE (a:Person {name: 'Alice', age: 30})
  - Basic query syntax to retrieve data
    - Example: MATCH (n:Person) WHERE n.name = 'Alice' RETURN n
  - Use the Neo4j Browser to run Cypher queries and view the graph visually.

Basic Cypher Syntax

- Match Clause:
  - The MATCH keyword is used to find patterns in the graph.
- Return Clause:
  - The RETURN keyword specifies the data to return after matching the pattern.
- Example: Find all persons named 'Alice'

```
MATCH (n:Person {name: 'Alice'})
RETURN n
```

Basic Data Types in Neo4j

- Supported Basic Data Types:
  - String: Text data (e.g., "Alice", "Hello World").
  - Integer: Whole numbers (e.g., 42, 1000).
  - Float: Decimal numbers (e.g., 3.14, 9.99).
  - Boolean: True or false (e.g., true, false).
  - Null: Absence of value.
- Data Types for Graph Modeling:
  - These types can be used in node properties, relationship properties, and even in query results.

Date and Time Data Types

- Date: Represents a calendar date (e.g., 2025-01-01).

```
CREATE (e:Event {eventDate: DATE('2025-01-01')})
```

- DateTime: Represents both date and time (e.g., 2025-01-01T12:30:00).

```
CREATE (e:Event {eventDateTime: DATETIME('2025-01-01T12:30:00’)})
```

- Querying Date/DateTime Properties:

```
MATCH (e:Event)
WHERE e.eventDate = DATE('2025-01-01')
RETURN e
```

List and Map Data Types

- List Data Type: A list is an ordered collection of values.

```
CREATE (p:Person {friends: ['Alice', 'Bob', 'Charlie']})
```

  - Example : Query person's friends:

```
MATCH (p:Person)
WHERE 'Alice' IN p.friends
RETURN p.name
```

- Map Data Type: A map is a collection of key-value pairs.

Cypher Functions

- Cypher Built-in Functions:
  - Mathematical Functions: abs(), rand(), log()…
  - String Functions: toUpper(), substring(), trim()…
  - Date/Time Functions: DATE(), DATETIME()…
  - Aggregation Functions: count(), avg(), sum()…
- Example:
  - DATE(): Returns the current date.

    ```
    RETURN DATE()  // Result: '2025-01-12'
    ```
  - DATETIME(): Returns the current date and time.

    ```
    RETURN DATETIME()  // Result: '2025-01-12T12:30:00'
    ```

Creating Nodes and Relationships

- Creating Nodes:
  - Use the CREATE keyword to create nodes.
  - Example: Create a node with a Person label and name property

```
CREATE (a:Person {name: 'Alice', age: 30})
```

- Creating Relationships:
  - Use CREATE to create relationships between nodes.
  - Example: Create a FRIENDS_WITH relationship between two persons:

```
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
CREATE (a)-[:FRIENDS_WITH]->(b)
```

Querying with MATCH

- MATCH Clause:
  - The MATCH clause is used to search for patterns in the graph.
  - Example: Find all friends of Alice

```
MATCH (a:Person {name: 'Alice'})-[:FRIENDS_WITH]-  >(b:Person)
RETURN b
```

- Pattern Matching:
  - Use parentheses () for nodes and -[]-> for relationships.

```
MATCH (a)-[:FRIENDS_WITH]->(b) RETURN a.name, b.name
```

Filtering Data with WHERE

- Using WHERE Clause:
  - The WHERE clause is used to filter the results.
  - Example: Find persons older than 25

```
MATCH (p:Person)
WHERE p.age > 25
RETURN p.name, p.age
```

- AND/OR in WHERE:
  - Combine multiple conditions using AND or OR.
  - Example: Find persons named 'Alice' or aged above 30:

```
MATCH (p:Person)
WHERE p.name = 'Alice' OR p.age > 30
RETURN p.name, p.age
```

Updating Data in Neo4j

- SET Clause:
  - The SET clause is used to update properties of nodes or relationships.
  - Example: Update the age of Alice

```
MATCH (a:Person {name: 'Alice'})
SET a.age = 31
```

- Adding Properties:
  - You can add new properties to nodes or relationships.
  - Example: Add a location property to a person

```
MATCH (a:Person {name: 'Alice'})
SET a.location = 'New York'
```

Deleting Nodes and Relationships

- Deleting Relationships:
  - Use DELETE to remove relationships.
  - Example: Delete a relationship between Alice and Bob

```
MATCH (a:Person {name: 'Alice'})-[r:FRIENDS_WITH]->(b:Person {name: 'Bob'})
DELETE r
```

- Deleting Nodes:
  - Nodes can be deleted, but only if they do not have any relationships.
  - Example: Delete a node

```
MATCH (a:Person {name: 'Alice'})
DELETE a
```

Aggregation and Grouping

- Aggregation:
  - Use aggregation functions like COUNT(), SUM(), AVG(), etc.
  - Example: Count the number of persons:

```
MATCH (p:Person)
RETURN COUNT(p)
```

- Grouping Results:
  - Use the WITH keyword to group results.
  - Example: Count the number of friends each person has

```
MATCH (p:Person)-[:FRIENDS_WITH]->(f:Person)
WITH p, COUNT(f) AS num_friends
RETURN p.name, num_friends
```

Using Indexes for Performance

- Create indexes on properties that are frequently queried to improve performance.
  - Example: Create an index on the name property of Person nodes

    ```
    CREATE INDEX FOR (p:Person) ON (p.name)
    ```
- Cypher automatically uses indexes for faster lookups on indexed properties.

Subqueries in Cypher

- A subquery is a query nested within another query in Cypher.
- Subqueries allow you to break down complex queries into smaller, more manageable parts.
- To reuse parts of queries and improve readability and maintainability.
- Syntax of Subqueries
  - A subquery is enclosed in parentheses and can be used anywhere a regular part of a query is expected.
  - Subqueries can be used within the RETURN, WHERE, or SET clauses.

Subqueries in Cypher

- Example in the RETURN Clause:
  - Retrieve a person and their friends

```
MATCH (p:Person)
RETURN p.name, [ (p)-[:FRIENDS_WITH]->(f:Person) |
f.name ] AS friends
```

- Example in the WHERE Clause:
  - Find persons who have at least 3 friends

```
MATCH (p:Person)
WHERE SIZE([(p)-[:FRIENDS_WITH]->() | 1]) >= 3
RETURN p.name
```

Subqueries in Cypher

- Using a Subquery to Filter:
  - Example: Find persons who are friends with Alice

```
MATCH (p:Person)
WHERE EXISTS { MATCH (a:Person {name: 'Alice'})-[:FRIENDS_WITH]->(p) }
RETURN p.name
```

Constraints in Neo4j

- Constraints are rules that enforce data integrity.
- Uniqueness Constraint
  - Ensures a property on nodes or relationships is unique across the graph.
  - Example: Enforce unique name for Person nodes

```
CREATE CONSTRAINT uq_person_name for (n:Person) REQUIRE n.name is UNIQUE;
```

- Existence Constraint
  - Example: Ensure every Person node has an age property

```
CREATE CONSTRAINT exists_person_age FOR (n:Person)
REQUIRE n.age IS NOT NULL;
```

Modifying the Schema

- Dropping Constraints:
  - Drop the uniqueness constraint on Person nodes:

    ```
    DROP CREATE CONSTRAINT uq_person_name
    ```

- Dropping Indexes:
  - Example: Drop the index on name for Person nodes:

    ```
    DROP INDEX FOR (p:Person) ON (p.name)
    ```

Schema Best Practices

- Keep the schema simple and flexible.
- Use labels and relationship types that reflect the real-world entities and connections.
- Apply constraints to enforce data integrity.
- Create indexes for properties that are frequently queried to improve performance.
- Avoid Overcomplicating the Schema
  - Too many constraints or indexes may slow down writes. Focus on optimizing read performance.

Loading CSV in Cypher

- What is CSV Import?
  - CSV *(Comma-Separated Values)* files are a common format for storing tabular data.
  - Neo4j allows importing CSV data directly into the graph database using Cypher queries.
- Why Use CSV in Neo4j?
  - Easy to work with external datasets.
  - Helps integrate data from external sources into your graph model.

Basic Syntax for Loading CSV

- Loading CSV File: The basic syntax to load CSV data in Cypher:

```
LOAD CSV WITH HEADERS FROM 'file:///path/to/file.csv'
AS row
```

- Example: Loading a list of people from CSV

```
LOAD CSV WITH HEADERS FROM 'file:///people.csv' AS row
CREATE (p:Person {name: row.name, age: toInteger(row.age)})
```

Handling Data Types

- Type Conversion: You may need to convert data types when loading CSV.
  - Example: Converting a string to an integer

```
CREATE (n:Person {name: row.name, age: toInteger(row.age)})
```

  - Other Type Conversions

Create Relationships While Loading

- You can also create relationships between nodes during CSV import

```
LOAD CSV WITH HEADERS FROM 'file:///relationships.csv'
AS row
MATCH (a:Person {name: row.name1}), (b:Person {name:
row.name2})
CREATE (a)-[:KNOWS]->(b)
```

Optimizing CSV Import

- Using USING PERIODIC COMMIT for Large Files:
  - For large CSV files, use USING PERIODIC COMMIT to commit transactions periodically.
  - This improves performance and avoids running out of memory for large datasets.

```
LOAD CSV WITH HEADERS FROM 'file:///bigfile.csv' AS row
USING PERIODIC COMMIT 1000
CREATE (p:Person {name: row.name, age: toInteger(row.age)})
```

Best Practices for Loading CSV

- Check for Duplicates:

  Before importing, check for existing nodes to avoid duplicates.

  ```
  MERGE (p:Person {name: row.name})
  ```

- Validate Data:

  Ensure data integrity before loading by filtering invalid rows.

  ```
  LOAD CSV WITH HEADERS FROM 'file:///people.csv' AS row
  WHERE row.age IS NOT NULL
  CREATE (p:Person {name: row.name, age: toInteger(row.age)})
  ```

Neo4j for Java developers

- Neo4j provides drivers which allow you to make a connection to the database and develop applications which create, read, update, and delete information from the graph.
- Neo4j Java Driver
  https://neo4j.com/docs/getting-started/languages-guides/java/java-intro/
  https://mvnrepository.com/artifact/org.neo4j.driver/neo4j-java-driver

Neo4j for Java developers

- Install the Neo4j Java Driver
  - Add dependencies for Maven or Gradle.
- Build a Connection String
  - Format: bolt://<host>:<port>.
- Create an Instance of the Driver
  - Use GraphDatabase.driver() to establish a connection.
- Verify Driver Connection
  - Ensure successful connection with simple query execution or exception handling.

    ```
    driver.verifyConnectivity();
    ```

Neo4j for Java developers

- Java Code for Connecting to Neo4j

```
Driver driver = GraphDatabase.driver("bolt://localhost:7687",
AuthTokens.basic("neo4j", "password"));
try (Session session = driver.session()) {
    // Perform queries
}
```

- Running Queries with Neo4j in Java

```
session.run("MATCH (n) RETURN n LIMIT 25");
```

- Closing the Connection

```
driver.close();
```

Interacting with Neo4j

- Open a Session and execute a Unit of Work within a Transaction.
- Execute Read and Write queries through the Driver.
- Consume the results returned from Neo4j.
- Handle potential errors thrown by the Driver.

Interacting with Neo4j - Session

- Open a new Session

```
var session = driver.session();
```

- Using a session with try-with-resources: The session will be automatically closed when the try block scope ends.

```
try (var session = driver.session()) {
   // Do something with the session
} // session will automatically close when exiting the scope
```

Interacting with Neo4j - Session

- Opening a Session with Additional Configuration: Configure the database and access mode (READ/WRITE) when working with multiple databases.

```
var sessionConfig = SessionConfig.builder()
        .withDefaultAccessMode(AccessMode.WRITE)
        .withDatabase("dbName")
        .build();

try (var session = driver.session(sessionConfig)) {
   // Do something with the session
}
```

Interacting with Neo4j - Transactions

- A transaction is a unit of work performed on the database, ensuring it is treated in a coherent and reliable way.
- ACID Transactions:
  - Atomicity: The transaction is indivisible.
  - Consistency: The database must transition from one valid state to another.
  - Isolation: Transactions do not affect each other.
  - Durability: Once committed, changes are permanent.

Interacting with Neo4j - Transactions

- Types of Transactions:
  - Auto-commit Transactions: A single unit of work that is executed immediately on the DBMS and acknowledged right away.
  - Read Transactions: Use when reading data from Neo4j.
  - Write Transactions: Use when writing data to the database.

Interacting with Neo4j - Transactions

- Auto-commit Transactions:

```
var query = "MATCH () RETURN count(*) AS count";
var params = Values.parameters();
// Run a query in an auto-commit transaction
var res = session.run(query, params)
.single()
.get("count")
.asLong();
```

Interacting with Neo4j - Transactions

- Read Transactions:

```
var res = session.readTransaction(tx -> {
return tx.run("MATCH (p:Person)-[:FRIEND]-> (f:Person)
      WHERE p.name = $name
      RETURN f.name AS friend
      LIMIT 10",
      Values.parameters("name", "Arthur")).
      list(r -> r.get("friend").asString());
});
```

Interacting with Neo4j - Transactions

- Write Transactions:

```
session.writeTransaction(tx -> {
return tx.run("MATCH (p:Person {name: $name1}),
      (f:Person {name: $name2}) " +
      "CREATE (p)-[:FRIEND]->(f)",
      Values.parameters("name1", "Arthur",
                  "name2", "Michael"))
      .consume();
});
```

Interacting with Neo4j - Transactions

- Manually Creating Transactions

```
var tx = session.beginTransaction();
try {
    tx.run(query, params);
    tx.commit();  // Commit the transaction
} catch (Exception e) {
    tx.rollback();  // Rollback the transaction if something goes
wrong
}
```

Processing Results

- The Neo4j Java Driver provides you with three APIs for consuming results:
  - Synchronous API
  - Async API
  - Reactive API

Synchronous API

- The most common and straightforward method for consuming results.

```
try (var session = driver.session()) {
    var res = session.readTransaction(tx -> tx.run(
        "MATCH (p:Person) RETURN p.name AS name LIMIT 10").list());
    res.stream()
        .map(row -> row.get("name"))
        .forEach(System.out::println);
} catch (Exception e) {
    e.printStackTrace();
}
```

Async API

- Asynchronous version of the API for more efficient resource usage.

```
var session = driver.asyncSession();
session.readTransactionAsync(tx -> tx.runAsync(
        "MATCH (p:Person) RETURN p.name AS name LIMIT 10")
    .thenApplyAsync(res -> res.listAsync(row -> row.get("name")))
    .thenAcceptAsync(System.out::println)
    .exceptionallyAsync(e -> {
        e.printStackTrace();
        return null;
    });
```

Reactive API

- Reactive API uses a reactive framework for asynchronous processing.

```
Flux.usingWhen(Mono.fromSupplier(driver::rxSession),
    session -> session.readTransaction(tx -> {
        var rxResult = tx.run(
            "MATCH (p:Person) RETURN p.name AS name LIMIT 10");
        return Flux
            .from(rxResult.records())
            .map(r -> r.get("name").asString())
            .doOnNext(System.out::println)
            .then(Mono.from(rxResult.consume()));
    }), RxSession::close);
```

The Result Object

- Result Object contains the records and additional meta data about the query execution.
- Accessing Results:
  - Iterator<Record>
  - stream(), list(), single()
- Record row = res.single();
- List<Record> rows = res.list();
- Stream<Record> rowStream = res.stream();

Working with Records

- Accessing Record Values:
  - Use the alias specified in the RETURN clause of the Cypher statement.
  - Or use column index (not recommended).

```
row.keys();  // Column names
row.containsKey("movie");  // Check existence
row.get("movieCount").asInt(0);  // Get numeric value
row.get("isDirector").asBoolean();  // Get boolean value
```

Result Summary

- Additional metadata about the query:
  - Number of nodes and relationships created, updated, or deleted.
  - Database and server information.
  - Query plan details.

```
ResultSummary summary = res.consume();
summary.resultAvailableAfter(TimeUnit.MILLISECONDS);
summary.resultConsumedAfter(TimeUnit.MILLISECONDS);
```

Result Counters

- Counters for statistics about the query:
  - Number of nodes, relationships created, deleted, and updated.
  - Added indexes and constraints.

```
SummaryCounters counters = summary.counters();
counters.nodesCreated();
counters.labelsAdded();
counters.relationshipsDeleted();
counters.indexesAdded();
```