Neo4j and Cypher Cheat Sheet

Neo4j and Cypher Cheat Sheet

A practical quick-reference guide for Neo4j graph database operations and the Cypher query language. Commands are organized by category with realistic examples.

Creating Data

Create Nodes

```
-- Create a single node
CREATE (p:Person {name: 'Alice', age: 30})

-- Create multiple nodes in one query
CREATE (p1:Person {name: 'Alice'}),
       (p2:Person {name: 'Bob'}),
       (p3:Person {name: 'Charlie'})

-- Create node with multiple labels
CREATE (p:Person:Employee {name: 'Alice', department: 'Engineering'})

-- Create node and return it
CREATE (p:Person {name: 'Alice'})
RETURN p
```

Create Relationships

```
-- Create a relationship between existing nodes
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
CREATE (a)-[:KNOWS]->(b)

-- Create relationship with properties
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
CREATE (a)-[:KNOWS {since: 2020, strength: 'strong'}]->(b)

-- Create nodes and relationship in one query
CREATE (a:Person {name: 'Alice'})-[:WORKS_FOR {role: 'Engineer'}]->(c:Company {name: 'TechCor

-- Create multiple relationships
MATCH (a:Person {name: 'Alice'})
```

```
CREATE (a)-[:LIVES_IN]->(:City {name: 'Seattle'}),
       (a)-[:WORKS_FOR]->(:Company {name: 'TechCorp'})
```

MERGE (Create If Not Exists)

```
-- Create node only if it does not exist
MERGE (p:Person {name: 'Alice'})

-- MERGE with ON CREATE and ON MATCH
MERGE (p:Person {name: 'Alice'})
ON CREATE SET p.created = timestamp(), p.age = 30
ON MATCH SET p.lastSeen = timestamp()

-- MERGE relationship
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
MERGE (a)-[r:KNOWS]->(b)
ON CREATE SET r.since = 2020
```

Matching Patterns

Basic MATCH

```
-- Find all nodes with a label
MATCH (p:Person)
RETURN p

-- Find nodes with property filter
MATCH (p:Person {name: 'Alice'})
RETURN p

-- Find with multiple conditions
MATCH (p:Person)
WHERE p.age > 25 AND p.city = 'Seattle'
RETURN p.name, p.age

-- Limit results
MATCH (p:Person)
RETURN p
LIMIT 10
```

Relationship Patterns

```
-- Outgoing relationship
MATCH (a:Person {name: 'Alice'})-[:KNOWS]->(b)
RETURN b.name

-- Incoming relationship
MATCH (a)<-[:WORKS_FOR]-(e:Person)
RETURN a.name AS company, e.name AS employee

-- Any direction
MATCH (a:Person)-[:KNOWS]-(b:Person)
RETURN a.name, b.name
```

```
-- Multiple hops (2 hops away)
MATCH (a:Person {name: 'Alice'})-[:KNOWS*2]->(b)
RETURN DISTINCT b.name

-- Variable length paths (1 to 3 hops)
MATCH (a:Person {name: 'Alice'})-[:KNOWS*1..3]->(b)
RETURN DISTINCT b.name

-- Any length path
MATCH (a:Person {name: 'Alice'})-[:KNOWS*]->(b)
RETURN DISTINCT b.name
```

Optional Patterns

```
-- LEFT JOIN equivalent (return null if no match)
MATCH (p:Person)
OPTIONAL MATCH (p)-[:WORKS_FOR]->(c:Company)
RETURN p.name, c.name AS company

-- Optional relationship with condition
MATCH (p:Person)
OPTIONAL MATCH (p)-[r:WORKS_FOR]->(c:Company)
WHERE r.role = 'Engineer'
RETURN p.name, c.name
```

Updating Data

SET (Add/Update Properties)

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

-- Add multiple properties
MATCH (p:Person {name: 'Alice'})
SET p.age = 31, p.city = 'Portland', p.updated = timestamp()

-- Replace all properties
MATCH (p:Person {name: 'Alice'})
SET p = {name: 'Alice', age: 31, city: 'Portland'}

-- Add to existing properties
MATCH (p:Person {name: 'Alice'})
SET p += {age: 31, city: 'Portland'}

-- Add label
MATCH (p:Person {name: 'Alice'})
SET p:Employee
```

REMOVE (Delete Properties or Labels)

```
-- Remove a property
MATCH (p:Person {name: 'Alice'})
REMOVE p.tempProperty

-- Remove multiple properties
MATCH (p:Person {name: 'Alice'})
REMOVE p.tempProperty, p.draft

-- Remove a label
MATCH (p:Person {name: 'Alice'})
REMOVE p:Employee

-- Remove multiple labels
MATCH (p:Person {name: 'Alice'})
REMOVE p:Employee:Contractor
```

DELETE (Delete Nodes and Relationships)

```
-- Delete a node (must have no relationships)
MATCH (p:Person {name: 'Alice'})
DELETE p

-- Delete a relationship
MATCH (a:Person {name: 'Alice'})-[r:KNOWS]->(b:Person {name: 'Bob'})
DELETE r

-- Delete node and all its relationships
MATCH (p:Person {name: 'Alice'})
DETACH DELETE p

-- Delete all nodes of a type (dangerous!)
MATCH (p:Person)
DETACH DELETE p
```

Relationships

Creating Relationships

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

-- With properties
MATCH (a:Person {name: 'Alice'}), (b:Company {name: 'TechCorp'})
CREATE (a)-[:WORKS_FOR {since: 2020, role: 'Engineer'}]->(b)

-- Multiple relationship types (use | for OR in patterns)
MATCH (a:Person)-[r:KNOWS|FRIEND]->(b:Person)
RETURN type(r), a.name, b.name
```

Traversing Relationships

```
-- Outgoing (arrow pointing right)
MATCH (a:Person)-[:KNOWS]->(b)
RETURN a.name, b.name

-- Incoming (arrow pointing left)
MATCH (a:Person)<-[:KNOWS]-(b)
RETURN a.name, b.name

-- Either direction (no arrow head)
MATCH (a:Person)-[:KNOWS]-(b)
RETURN a.name, b.name

-- Named relationship for accessing properties
MATCH (a:Person)-[r:KNOWS]->(b)
RETURN r.since, a.name, b.name

-- Variable length (min 1, max 5 hops)
MATCH (a:Person)-[:KNOWS*1..5]->(b)
RETURN a.name, b.name

-- Exact length (exactly 2 hops)
MATCH (a:Person)-[:KNOWS*2]->(b)
RETURN a.name, b.name
```

Filtering

WHERE Clause

```
-- Basic comparison
MATCH (p:Person)
WHERE p.age > 30
RETURN p.name

-- Multiple conditions
MATCH (p:Person)
WHERE p.age > 25 AND p.city = 'Seattle'
RETURN p.name

-- OR condition
MATCH (p:Person)
WHERE p.city = 'Seattle' OR p.city = 'Portland'
RETURN p.name

-- String matching
MATCH (p:Person)
WHERE p.name STARTS WITH 'A'
RETURN p.name

MATCH (p:Person)
WHERE p.name ENDS WITH 'son'
RETURN p.name

MATCH (p:Person)
WHERE p.name CONTAINS 'ali'
RETURN p.name
```

```
-- Regular expression
MATCH (p:Person)
WHERE p.name =~ 'A.*'
RETURN p.name
```

IN and Range

```
-- IN list
MATCH (p:Person)
WHERE p.city IN ['Seattle', 'Portland', 'San Francisco']
RETURN p.name

-- Numeric range
MATCH (p:Person)
WHERE p.age >= 25 AND p.age <= 35
RETURN p.name

-- Using IN with numbers
MATCH (p:Person)
WHERE p.age IN [25, 30, 35, 40]
RETURN p.name
```

EXISTS and Pattern Matching

```
-- Check if property exists
MATCH (p:Person)
WHERE EXISTS(p.email)
RETURN p.name

-- Check if relationship exists
MATCH (p:Person)
WHERE EXISTS((p)-[:WORKS_FOR]->())
RETURN p.name

-- NOT EXISTS
MATCH (p:Person)
WHERE NOT EXISTS((p)-[:WORKS_FOR]->())
RETURN p.name

-- Pattern in WHERE
MATCH (p:Person)
WHERE (p)-[:LIVES_IN]->(:City {name: 'Seattle'})
RETURN p.name
```

NULL Handling

```
-- Check for null
MATCH (p:Person)
WHERE p.email IS NULL
RETURN p.name

-- Check for not null
MATCH (p:Person)
WHERE p.email IS NOT NULL
```

```
RETURN p.name

-- Coalesce (return first non-null)
MATCH (p:Person)
RETURN p.name, COALESCE(p.nickname, p.name, 'Unknown') AS displayName

-- Default value with CASE
MATCH (p:Person)
RETURN p.name,
       CASE WHEN p.age IS NULL THEN 'Unknown'
            ELSE toString(p.age)
       END AS age
```

Aggregation

COUNT

```
-- Count all nodes
MATCH (p:Person)
RETURN COUNT(p)

-- Count with grouping
MATCH (p:Person)-[:LIVES_IN]->(c:City)
RETURN c.name, COUNT(p) AS population

-- Count distinct values
MATCH (p:Person)
RETURN COUNT(DISTINCT p.city) AS cities

-- Count relationships
MATCH (p:Person)-[r:KNOWS]->()
RETURN p.name, COUNT(r) AS friends
```

SUM, AVG, MIN, MAX

```
-- Sum
MATCH (p:Person)
RETURN SUM(p.salary) AS totalSalary

-- Average
MATCH (p:Person)
RETURN AVG(p.age) AS averageAge

-- Min and Max
MATCH (p:Person)
RETURN MIN(p.age) AS youngest, MAX(p.age) AS oldest

-- With grouping
MATCH (p:Person)-[:WORKS_FOR]->(c:Company)
RETURN c.name, AVG(p.salary) AS avgSalary, COUNT(p) AS employees
```

COLLECT and DISTINCT

```
-- Collect into a list
MATCH (p:Person)-[:WORKS_FOR]->(c:Company)
RETURN c.name, COLLECT(p.name) AS employees

-- Distinct values
MATCH (p:Person)
RETURN DISTINCT p.city

-- Collect distinct
MATCH (p:Person)-[:WORKS_FOR]->(c:Company)
RETURN c.name, COLLECT(DISTINCT p.department) AS departments

-- Nested collect
MATCH (c:Company)<-[:WORKS_FOR]-(p:Person)-[:HAS_SKILL]->(s:Skill)
RETURN c.name, COLLECT(DISTINCT {person: p.name, skill: s.name}) AS skills
```

Ordering and Pagination

ORDER BY

```
-- Ascending (default)
MATCH (p:Person)
RETURN p.name, p.age
ORDER BY p.age

-- Descending
MATCH (p:Person)
RETURN p.name, p.age
ORDER BY p.age DESC

-- Multiple columns
MATCH (p:Person)
RETURN p.name, p.city, p.age
ORDER BY p.city, p.age DESC

-- Order by aggregated value
MATCH (p:Person)-[:KNOWS]->(f)
RETURN p.name, COUNT(f) AS friendCount
ORDER BY friendCount DESC
```

SKIP and LIMIT

```
-- Limit results
MATCH (p:Person)
RETURN p.name
ORDER BY p.name
LIMIT 10

-- Skip results (offset)
MATCH (p:Person)
RETURN p.name
ORDER BY p.name
SKIP 10
```

```
-- Pagination (page 2 with 10 per page)
MATCH (p:Person)
RETURN p.name
ORDER BY p.name
SKIP 10 LIMIT 10
```

WITH Clause

Chaining Queries

```
-- Filter after aggregation
MATCH (p:Person)-[:WORKS_FOR]->(c:Company)
WITH c, COUNT(p) AS employeeCount
WHERE employeeCount > 5
RETURN c.name, employeeCount

-- Transform data between stages
MATCH (p:Person)
WITH p.name AS personName, p.age AS personAge
WHERE personAge > 30
RETURN personName, personAge

-- Multiple aggregations
MATCH (p:Person)-[:WORKS_FOR]->(c:Company)
WITH c, COLLECT(p.name) AS employees, AVG(p.salary) AS avgSalary
WHERE avgSalary > 50000
RETURN c.name, employees, avgSalary

-- Calculate then filter
MATCH (p:Person)
WITH p, p.salary * 1.1 AS newSalary
WHERE newSalary > 100000
RETURN p.name, newSalary
```

Renaming Variables

```
-- Rename for clarity
MATCH (p:Person)-[:WORKS_FOR]->(comp:Company)
WITH p AS employee, comp AS employer
RETURN employee.name, employer.name

-- Preserve only needed variables
MATCH (p:Person)-[:WORKS_FOR]->(c:Company)
WITH p.name AS name, c.name AS company
RETURN name, company
```

Indexes and Constraints

CREATE INDEX

```
-- Single property index
CREATE INDEX person_name IF NOT EXISTS
```

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

-- Composite index
CREATE INDEX person_city_age IF NOT EXISTS
FOR (p:Person) ON (p.city, p.age)

-- Full-text index
CREATE FULLTEXT INDEX person_fulltext IF NOT EXISTS
FOR (p:Person) ON EACH [p.name, p.bio]

-- Show indexes
SHOW INDEXES

-- Drop index
DROP INDEX person_name IF EXISTS
```

CREATE CONSTRAINT

```
-- Unique constraint
CREATE CONSTRAINT person_email_unique IF NOT EXISTS
FOR (p:Person) REQUIRE p.email IS UNIQUE

-- Existence constraint (property must exist)
CREATE CONSTRAINT person_name_exists IF NOT EXISTS
FOR (p:Person) REQUIRE p.name IS NOT NULL

-- Node key constraint (composite unique + not null)
CREATE CONSTRAINT person_key IF NOT EXISTS
FOR (p:Person) REQUIRE (p.firstName, p.lastName) IS NODE KEY

-- Relationship existence constraint
CREATE CONSTRAINT works_since_exists IF NOT EXISTS
FOR ()-[r:WORKS_FOR]-()
REQUIRE r.since IS NOT NULL

-- Show constraints
SHOW CONSTRAINTS

-- Drop constraint
DROP CONSTRAINT person_email_unique IF EXISTS
```

Procedures

Calling Procedures

```
-- Call built-in procedure
CALL db.labels()

-- Call with arguments
CALL db.schema.visualization()

-- Call APOC procedure (if installed)
CALL apoc.help('search')

-- Call and filter results
```

```
CALL db.indexes()
YIELD name, state
WHERE state = 'ONLINE'
RETURN name, state

-- Call with YIELD and WHERE
CALL dbms.security.listUsers()
YIELD username, roles
WHERE 'admin' IN roles
RETURN username
```

Listing Available Procedures

```
-- List all procedures
SHOW PROCEDURES

-- List functions
SHOW FUNCTIONS

-- Filter by category
SHOW PROCEDURES YIELD name, category
WHERE category = 'ADMIN'
RETURN name
```

Database Management

Database Operations

```
-- List databases
SHOW DATABASES

-- Show current database
SHOW DEFAULT DATABASE

-- Create database (Enterprise only)
CREATE DATABASE myDatabase IF NOT EXISTS

-- Start database
START DATABASE myDatabase

-- Stop database
STOP DATABASE myDatabase

-- Drop database (Enterprise only)
DROP DATABASE myDatabase IF EXISTS

-- Switch database context
:use myDatabase
```

Database Statistics

```
-- Get node count
MATCH (n)
```

```
RETURN count(n) AS nodeCount

-- Get counts by label
MATCH (n)
RETURN labels(n) AS labels, count(*) AS count

-- Get relationship count
MATCH ()-[r]->()
RETURN count(r) AS relationshipCount

-- Get counts by relationship type
MATCH ()-[r]->()
RETURN type(r) AS type, count(*) AS count

-- Property existence stats
CALL db.stats.retrieve('GRAPH COUNTS')
YIELD data
RETURN data
```

User Management

User Operations

```
-- Create user
CREATE USER alice SET PASSWORD 'password123' CHANGE NOT REQUIRED

-- Create user with password change required
CREATE USER bob SET PASSWORD 'tempPassword' CHANGE REQUIRED

-- Alter user password
ALTER USER alice SET PASSWORD 'newPassword456'

-- Alter user status
ALTER USER alice SET STATUS SUSPENDED
ALTER USER bob SET STATUS ACTIVE

-- Drop user
DROP USER alice IF EXISTS

-- List users
SHOW USERS
```

Role Management

```
-- Create role
CREATE ROLE analyst

-- Grant role to user
GRANT ROLE analyst TO alice

-- Revoke role from user
REVOKE ROLE analyst FROM alice

-- List roles
SHOW ROLES
```

```
-- Show user roles
SHOW POPULATED ROLES
```

Privileges

```
-- Grant read access
GRANT READ {name, age} ON GRAPH * NODES Person TO analyst

-- Grant traverse (can traverse but not read data)
GRANT TRAVERSE ON GRAPH * TO analyst

-- Grant write access
GRANT WRITE ON GRAPH * TO analyst

-- Grant all privileges
GRANT ALL ON DATABASE * TO admin

-- Revoke privileges
REVOKE READ ON GRAPH * FROM analyst
```

Import and Export

LOAD CSV

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

-- Load CSV without headers
LOAD CSV FROM 'file:///data.csv' AS row
CREATE (p:Person {name: row[0], age: toInteger(row[1])})

-- With field terminator
LOAD CSV WITH HEADERS FROM 'file:///data.tsv' AS row
FIELDTERMINATOR '\t'
CREATE (p:Person {name: row.name})

-- Batch processing with periodic commit
USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM 'file:///people.csv' AS row
CREATE (p:Person {name: row.name})

-- Create relationships from CSV
LOAD CSV WITH HEADERS FROM 'file:///works_for.csv' AS row
MATCH (p:Person {name: row.personName})
MATCH (c:Company {name: row.companyName})
CREATE (p)-[:WORKS_FOR {role: row.role}]->(c)
```

APOC Load (if APOC is installed)

```
-- Load JSON
CALL apoc.load.json('file:///data.json')
```

```
YIELD value
CREATE (p:Person SET p = value)

-- Load JSON from URL
CALL apoc.load.json('https://api.example.com/users')
YIELD value
CREATE (p:Person {name: value.name})

-- Export to JSON
CALL apoc.export.json.query(
  'MATCH (p:Person) RETURN p.name',
  'people.json',
  {stream: true}
)

-- Export to CSV
CALL apoc.export.csv.query(
  'MATCH (p:Person) RETURN p.name, p.age',
  'people.csv',
  {stream: true}
)
```

Path Functions

Shortest Path

```
-- Find shortest path
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
MATCH path = shortestPath((a)-[:KNOWS*]-(b))
RETURN path

-- Shortest path with max length
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
CALL apoc.algo.shortestPath(a, b, 'KNOWS')
YIELD path
RETURN path

-- All shortest paths
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
MATCH path = allShortestPaths((a)-[:KNOWS*]-(b))
RETURN path
```

Path Operations

```
-- Return path and its length
MATCH path = (a:Person {name: 'Alice'})-[:KNOWS*1..5]-(b)
RETURN path, length(path) AS hops

-- Get nodes from path
MATCH path = (a)-[:KNOWS*]-(b)
RETURN [node IN nodes(path) | node.name] AS names

-- Get relationships from path
MATCH path = (a)-[:KNOWS*]-(b)
RETURN [rel IN relationships(path) | type(rel)] AS relTypes
```

```
-- Get start and end nodes
MATCH path = (a)-[:KNOWS*]-(b)
RETURN startNode(path).name AS start, endNode(path).name AS end
```

Common Functions

String Functions

```
-- Concatenation
RETURN toString(42) + ' items' AS result

-- toUpper / toLower
MATCH (p:Person)
RETURN toUpper(p.name) AS nameUpper, toLower(p.name) AS nameLower

-- Substring
RETURN substring('Hello World', 0, 5) AS result  -- 'Hello'

-- Trim
RETURN trim('  hello  ') AS result  -- 'hello'

-- Replace
RETURN replace('hello world', 'world', 'Neo4j') AS result

-- Split
RETURN split('a,b,c', ',') AS result  -- ['a', 'b', 'c']

-- Left / Right
RETURN left('Hello', 3) AS result  -- 'Hel'
RETURN right('Hello', 3) AS result  -- 'llo'

-- String matching
RETURN ltrim('  hello') AS result  -- 'hello' (left trim)
RETURN rtrim('hello  ') AS result  -- 'hello' (right trim)
```

Numeric Functions

```
-- Absolute value
RETURN abs(-42) AS result  -- 42

-- Rounding
RETURN round(3.7) AS result  -- 4
RETURN floor(3.7) AS result  -- 3
RETURN ceil(3.2) AS result   -- 4

-- Random
RETURN rand() AS random  -- 0.0 to 1.0
RETURN toInteger(rand() * 100) AS randomInt  -- 0 to 99

-- Sign
RETURN sign(-42) AS result  -- -1
RETURN sign(42) AS result   -- 1
RETURN sign(0) AS result    -- 0
```

```
-- Square root
RETURN sqrt(16) AS result  -- 4.0

-- Power
RETURN 2^10 AS result      -- 1024
RETURN pow(2, 10) AS result  -- 1024.0
```

List Functions

```
-- Size of list
RETURN size([1, 2, 3, 4, 5]) AS result  -- 5

-- Head and last
RETURN head([1, 2, 3]) AS result   -- 1
RETURN last([1, 2, 3]) AS result   -- 3

-- Range
RETURN range(0, 10) AS result      -- [0,1,2,3,4,5,6,7,8,9,10]
RETURN range(0, 10, 2) AS result   -- [0,2,4,6,8,10]

-- List comprehension
MATCH (p:Person)
RETURN [x IN p.skills WHERE x <> 'legacy' | toUpper(x)] AS skills

-- Reduce
RETURN reduce(total = 0, x IN [1,2,3,4,5] | total + x) AS sum  -- 15

-- Unwind (expand list to rows)
UNWIND [1, 2, 3] AS x
RETURN x * 2 AS doubled

-- Flatten nested lists
RETURN [[1,2], [3,4]] AS nested, flatten([[1,2], [3,4]]) AS flat
```

Temporal Functions

```
-- Current date and time
RETURN date() AS today
RETURN datetime() AS now
RETURN time() AS currentTime
RETURN timestamp() AS unixTimestamp

-- Create specific date
RETURN date('2026-02-16') AS specificDate
RETURN date({year: 2026, month: 2, day: 16}) AS specificDate

-- Date arithmetic
MATCH (p:Person)
WHERE date(p.birthDate) > date('2000-01-01')
RETURN p.name

-- Date components
WITH datetime() AS dt
RETURN dt.year, dt.month, dt.day, dt.hour, dt.minute

-- Duration
```

```
RETURN duration.between(date('2020-01-01'), date('2026-02-16')) AS diff
RETURN duration.inDays(date('2020-01-01'), date('2026-02-16')).days AS days
```

Type Conversion

```
-- To string
RETURN toString(42) AS result
RETURN toString(true) AS result

-- To integer
RETURN toInteger('42') AS result
RETURN toInteger(3.7) AS result  -- 3

-- To float
RETURN toFloat('3.14') AS result

-- To boolean
RETURN toBoolean('true') AS result
RETURN toBoolean(1) AS result

-- Check type
RETURN apoc.meta.type(42) AS result        -- 'Long'
RETURN apoc.meta.type('hello') AS result   -- 'String'
RETURN apoc.meta.type([1,2,3]) AS result   -- 'List'
```

Coalesce and Null Handling

```
-- COALESCE: return first non-null
RETURN COALESCE(null, null, 'hello') AS result  -- 'hello'

-- NullIf: return null if values match
RETURN nullif('hello', 'hello') AS result  -- null
RETURN nullif('hello', 'world') AS result  -- 'hello'

-- CASE expression
MATCH (p:Person)
RETURN p.name,
       CASE
         WHEN p.age < 18 THEN 'Minor'
         WHEN p.age < 65 THEN 'Adult'
         ELSE 'Senior'
       END AS category
```

Graph Data Science (GDS)

The Neo4j Graph Data Science library provides algorithms for analyzing graph structures. Most algorithms support multiple execution modes.

Setup and Configuration

```
-- Check GDS library version
CALL gds.version()
```

```
-- List all available algorithms
CALL gds.list()

-- Check if GDS is properly installed
RETURN gds.version() AS gdsVersion
```

Graph Projection

```
-- Project a named graph (native projection)
CALL gds.graph.project(
  'myGraph',
  'Person',
  'KNOWS'
)

-- Project with multiple node labels and relationship types
CALL gds.graph.project(
  'socialNetwork',
  ['Person', 'Company'],
  ['KNOWS', 'WORKS_FOR']
)

-- Project with relationship properties
CALL gds.graph.project(
  'weightedGraph',
  'Person',
  {
    KNOWS: {
      properties: ['weight', 'since']
    }
  }
)

-- Project using Cypher query (Cypher projection)
CALL gds.graph.project.cypher(
  'cypherGraph',
  'MATCH (p:Person) RETURN id(p) AS id',
  'MATCH (p:Person)-[:KNOWS]->(q:Person) RETURN id(p) AS source, id(q) AS target'
)

-- List all projected graphs
CALL gds.graph.list()

-- Get details of a specific graph
CALL gds.graph.list('myGraph')
YIELD graphName, nodeCount, relationshipCount

-- Drop a projected graph
CALL gds.graph.drop('myGraph')

-- Drop graph without error if it does not exist
CALL gds.graph.drop('myGraph', false)
```

Centrality Algorithms

Centrality algorithms identify important nodes in a network.

```
-- PageRank: Measures node importance based on incoming relationships
CALL gds.pageRank.stream('myGraph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC
LIMIT 10

-- PageRank with write mode (saves score to database)
CALL gds.pageRank.write('myGraph', {
  writeProperty: 'pagerank'
})

-- PageRank with custom damping factor
CALL gds.pageRank.stream('myGraph', {
  dampingFactor: 0.85,
  maxIterations: 20
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score

-- Betweenness Centrality: Measures how often a node lies on shortest paths
CALL gds.betweenness.stream('myGraph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC

-- Betweenness with sampled approximation (faster for large graphs)
CALL gds.betweenness.stream('myGraph', {
  samplingSize: 1000
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score

-- Degree Centrality: Counts the number of relationships per node
CALL gds.degree.stream('myGraph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC

-- Degree centrality for specific relationship direction
CALL gds.degree.stream('myGraph', {
  orientation: 'REVERSE'
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score

-- Closeness Centrality: Measures average distance to all other nodes
CALL gds.closeness.stream('myGraph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC

-- Closeness with Wasserman-Faust variant (for disconnected graphs)
CALL gds.closeness.stream('myGraph', {
  useWassermanFaust: true
})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
```

Community Detection

Community detection algorithms find clusters of densely connected nodes.

```
-- Louvain: Detects communities by maximizing modularity
CALL gds.louvain.stream('myGraph')
YIELD nodeId, communityId
RETURN gds.util.asNode(nodeId).name AS name, communityId
ORDER BY communityId

-- Louvain with write mode
CALL gds.louvain.write('myGraph', {
  writeProperty: 'community'
})

-- Louvain with max iterations and hierarchy levels
CALL gds.louvain.stream('myGraph', {
  maxIterations: 20,
  includeIntermediateCommunities: true
})
YIELD nodeId, communityId, intermediateCommunityIds
RETURN gds.util.asNode(nodeId).name AS name, communityId

-- Label Propagation: Spreads labels through the network
CALL gds.labelPropagation.stream('myGraph')
YIELD nodeId, communityId
RETURN gds.util.asNode(nodeId).name AS name, communityId
ORDER BY communityId

-- Label Propagation with seed property
CALL gds.labelPropagation.stream('myGraph', {
  seedProperty: 'initialCommunity'
})
YIELD nodeId, communityId
RETURN gds.util.asNode(nodeId).name AS name, communityId

-- Weakly Connected Components (WCC): Finds disconnected subgraphs
CALL gds.wcc.stream('myGraph')
YIELD nodeId, componentId
RETURN gds.util.asNode(nodeId).name AS name, componentId
ORDER BY componentId

-- WCC with write mode
CALL gds.wcc.write('myGraph', {
  writeProperty: 'component'
})

-- Get count of nodes per component
CALL gds.wcc.stream('myGraph')
YIELD componentId
RETURN componentId, count(*) AS componentSize
ORDER BY componentSize DESC

-- Strongly Connected Components (SCC): Mutually reachable nodes
CALL gds.scc.stream('myGraph')
YIELD nodeId, componentId
RETURN gds.util.asNode(nodeId).name AS name, componentId
ORDER BY componentId
```

```
-- Triangle Count: Counts triangles each node participates in
CALL gds.triangleCount.stream('myGraph')
YIELD nodeId, triangleCount
RETURN gds.util.asNode(nodeId).name AS name, triangleCount
ORDER BY triangleCount DESC

-- Local Clustering Coefficient: How connected a node's neighbors are
CALL gds.localClusteringCoefficient.stream('myGraph')
YIELD nodeId, localClusteringCoefficient
RETURN gds.util.asNode(nodeId).name AS name, localClusteringCoefficient
ORDER BY localClusteringCoefficient DESC
```

Path Finding Algorithms

Path finding algorithms find optimal routes between nodes.

```
-- Dijkstra Shortest Path: Finds shortest path by weight
MATCH (source:Person {name: 'Alice'}), (target:Person {name: 'Bob'})
CALL gds.shortestPath.dijkstra.stream('myGraph', {
  sourceNode: source,
  targetNode: target,
  relationshipWeightProperty: 'weight'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
  [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS nodeNames,
  costs,
  totalCost

-- Dijkstra with multiple targets
MATCH (source:Person {name: 'Alice'})
CALL gds.shortestPath.dijkstra.stream('myGraph', {
  sourceNode: source,
  relationshipWeightProperty: 'weight'
})
YIELD nodeIds, totalCost
RETURN totalCost, size(nodeIds) AS pathLength
ORDER BY totalCost
LIMIT 5

-- A* (A-Star) Shortest Path: Uses heuristic for faster pathfinding
MATCH (source:Person {name: 'Alice'}), (target:Person {name: 'Bob'})
CALL gds.shortestPath.astar.stream('myGraph', {
  sourceNode: source,
  targetNode: target,
  relationshipWeightProperty: 'weight',
  latitudeProperty: 'lat',
  longitudeProperty: 'lon'
})
YIELD nodeIds, totalCost
RETURN [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS path, totalCost

-- Yen's K-Shortest Paths: Finds top K shortest paths
MATCH (source:Person {name: 'Alice'}), (target:Person {name: 'Bob'})
CALL gds.allShortestPaths.yens.stream('myGraph', {
  sourceNode: source,
  targetNode: target,
```

```
  k: 3,
  relationshipWeightProperty: 'weight'
})
YIELD index, nodeIds, totalCost
RETURN index, [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS path, totalCost
ORDER BY index

-- Breadth-First Search (BFS): Level-by-level traversal
MATCH (source:Person {name: 'Alice'})
CALL gds.bfs.stream('myGraph', {
  sourceNode: source
})
YIELD nodeIds
RETURN [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS visitedNodes

-- BFS with target node
MATCH (source:Person {name: 'Alice'}), (target:Person {name: 'Bob'})
CALL gds.bfs.stream('myGraph', {
  sourceNode: source,
  targetNodes: [target]
})
YIELD nodeIds
RETURN [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS path

-- Depth-First Search (DFS): Explores deeply before backtracking
MATCH (source:Person {name: 'Alice'})
CALL gds.dfs.stream('myGraph', {
  sourceNode: source
})
YIELD nodeIds
RETURN [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS visitedNodes

-- All Shortest Paths (unweighted)
MATCH (source:Person {name: 'Alice'}), (target:Person {name: 'Bob'})
CALL gds.allShortestPaths.stream('myGraph', {
  sourceNode: source,
  targetNode: target
})
YIELD nodeIds
RETURN [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS path
```

Similarity Algorithms

Similarity algorithms measure how alike nodes are based on their connections.

```
-- Jaccard Similarity: Intersection over union of neighbors
CALL gds.nodeSimilarity.stream('myGraph', {
  topK: 5,
  similarityCutoff: 0.1
})
YIELD node1, node2, similarity
RETURN
  gds.util.asNode(node1).name AS person1,
  gds.util.asNode(node2).name AS person2,
  similarity
ORDER BY similarity DESC

-- Node Similarity with write mode
```

```
CALL gds.nodeSimilarity.write('myGraph', {
  writeRelationshipType: 'SIMILAR',
  writeProperty: 'score',
  topK: 10
})

-- Filter by specific nodes
MATCH (p:Person)
WHERE p.name IN ['Alice', 'Bob', 'Charlie']
WITH collect(p) AS people
CALL gds.nodeSimilarity.stream('myGraph', {
  nodeFilter: people
})
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name, gds.util.asNode(node2).name, similarity

-- Cosine Similarity: Vector-based similarity
CALL gds.nodeSimilarity.cosine.stream('myGraph', {
  nodeProperties: ['feature1', 'feature2', 'feature3']
})
YIELD node1, node2, similarity
RETURN
  gds.util.asNode(node1).name AS person1,
  gds.util.asNode(node2).name AS person2,
  similarity
ORDER BY similarity DESC

-- Pearson Similarity: Correlation-based similarity
CALL gds.nodeSimilarity.pearson.stream('myGraph', {
  nodeProperties: ['rating1', 'rating2']
})
YIELD node1, node2, similarity
RETURN
  gds.util.asNode(node1).name AS item1,
  gds.util.asNode(node2).name AS item2,
  similarity
ORDER BY similarity DESC

-- Euclidean Distance: Geometric distance between nodes
CALL gds.alpha.similarity.euclidean.stream({
  nodeProperties: 'embedding',
  data: [
    {item: 'Alice', properties: [1.0, 2.0, 3.0]},
    {item: 'Bob', properties: [4.0, 5.0, 6.0]}
  ]
})
YIELD item1, item2, similarity
RETURN item1, item2, similarity
```

Link Prediction Algorithms

Link prediction algorithms estimate the likelihood of future connections.

```
-- Adamic Adar: Weights common neighbors by their degree
MATCH (p1:Person {name: 'Alice'})
MATCH (p2:Person {name: 'Bob'})
RETURN gds.alpha.linkprediction.adamicAdar(p1, p2, {
  relationshipType: 'KNOWS',
```

```
  direction: 'BOTH'
}) AS score

-- Common Neighbors: Counts shared neighbors
MATCH (p1:Person {name: 'Alice'})
MATCH (p2:Person {name: 'Bob'})
RETURN gds.alpha.linkprediction.commonNeighbors(p1, p2, {
  relationshipType: 'KNOWS',
  direction: 'BOTH'
}) AS commonNeighborCount

-- Preferential Attachment: Product of neighbor counts
MATCH (p1:Person {name: 'Alice'})
MATCH (p2:Person {name: 'Bob'})
RETURN gds.alpha.linkprediction.preferentialAttachment(p1, p2, {
  relationshipType: 'KNOWS',
  direction: 'BOTH'
}) AS score

-- Resource Allocation: Similar to Adamic Adar with different weighting
MATCH (p1:Person {name: 'Alice'})
MATCH (p2:Person {name: 'Bob'})
RETURN gds.alpha.linkprediction.resourceAllocation(p1, p2, {
  relationshipType: 'KNOWS',
  direction: 'BOTH'
}) AS score

-- Total Neighbors: Total unique neighbors of both nodes
MATCH (p1:Person {name: 'Alice'})
MATCH (p2:Person {name: 'Bob'})
RETURN gds.alpha.linkprediction.totalNeighbors(p1, p2, {
  relationshipType: 'KNOWS',
  direction: 'BOTH'
}) AS totalNeighbors

-- Batch link prediction for all unconnected pairs
MATCH (p1:Person), (p2:Person)
WHERE p1.name < p2.name
AND NOT EXISTS((p1)-[:KNOWS]-(p2))
WITH p1, p2,
     gds.alpha.linkprediction.adamicAdar(p1, p2, {
       relationshipType: 'KNOWS',
       direction: 'BOTH'
     }) AS score
WHERE score > 0
RETURN p1.name, p2.name, score
ORDER BY score DESC
LIMIT 10
```

Node Embeddings

Node embedding algorithms create vector representations of nodes.

```
-- FastRP (Fast Random Projection): Fast embedding generation
CALL gds.fastRP.stream('myGraph', {
  embeddingDimension: 128,
  randomSeed: 42
})
```

```
YIELD nodeId, embedding
RETURN gds.util.asNode(nodeId).name AS name, embedding

-- FastRP with iteration weights
CALL gds.fastRP.stream('myGraph', {
  embeddingDimension: 256,
  iterationWeights: [0.8, 0.2, 0.02]
})
YIELD nodeId, embedding
RETURN gds.util.asNode(nodeId).name AS name, embedding

-- FastRP with write mode
CALL gds.fastRP.write('myGraph', {
  embeddingDimension: 128,
  writeProperty: 'embedding'
})

-- FastRP with relationship weight property
CALL gds.fastRP.stream('myGraph', {
  embeddingDimension: 128,
  relationshipWeightProperty: 'weight'
})
YIELD nodeId, embedding
RETURN gds.util.asNode(nodeId).name AS name, embedding

-- Node2Vec: Embedding based on random walks
CALL gds.node2vec.stream('myGraph', {
  embeddingDimension: 128,
  walkLength: 80,
  walksPerNode: 10,
  returnFactor: 1.0,
  inOutFactor: 1.0
})
YIELD nodeId, embedding
RETURN gds.util.asNode(nodeId).name AS name, embedding

-- Node2Vec with write mode
CALL gds.node2vec.write('myGraph', {
  embeddingDimension: 64,
  writeProperty: 'node2vecEmbedding'
})

-- GraphSAGE: Embedding using neural network sampling
CALL gds.beta.graphSage.stream('myGraph', {
  embeddingDimension: 128,
  sampleSizes: [25, 10],
  aggregator: 'mean'
})
YIELD nodeId, embedding
RETURN gds.util.asNode(nodeId).name AS name, embedding

-- GraphSAGE with write mode
CALL gds.beta.graphSage.write('myGraph', {
  embeddingDimension: 128,
  writeProperty: 'graphSageEmbedding',
  modelName: 'myGraphSageModel'
})

-- Train GraphSAGE model for later use
```

```
CALL gds.beta.graphSage.train('myGraph', {
  embeddingDimension: 128,
  modelName: 'myGraphSageModel',
  sampleSizes: [25, 10]
})
YIELD modelInfo
RETURN modelInfo

-- HashGNN: Hash-based embedding for large graphs
CALL gds.alpha.hashgnn.stream('myGraph', {
  embeddingDimension: 128,
  iterations: 5
})
YIELD nodeId, embedding
RETURN gds.util.asNode(nodeId).name AS name, embedding
```

Graph Projection Modes

GDS algorithms support different execution modes for various use cases.

```
-- STREAM mode: Returns results directly (default for most examples above)
CALL gds.pageRank.stream('myGraph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score

-- WRITE mode: Writes results back to the database
CALL gds.pageRank.write('myGraph', {
  writeProperty: 'pagerank',
  maxIterations: 20
})
YIELD nodePropertiesWritten, ranIterations
RETURN nodePropertiesWritten, ranIterations

-- MUTATE mode: Adds results to the projected graph (not database)
CALL gds.pageRank.mutate('myGraph', {
  mutateProperty: 'pagerank'
})
YIELD nodePropertiesWritten, ranIterations
RETURN nodePropertiesWritten, ranIterations

-- STATS mode: Returns only summary statistics
CALL gds.pageRank.stats('myGraph')
YIELD ranIterations, didConverge, preProcessingMillis, computeMillis

-- ESTIMATE mode: Estimates memory and time before running
CALL gds.pageRank.stats.estimate('myGraph', {
  maxIterations: 20
})
YIELD bytesMin, bytesMax, nodeCount, relationshipCount

-- Estimate memory for write operation
CALL gds.pageRank.write.estimate('myGraph', {
  writeProperty: 'pagerank'
})
YIELD bytesMin, bytesMax, requiredMemory
RETURN requiredMemory
```

Pregel API (Custom Algorithms)

```
-- Run a custom Pregel algorithm
CALL gds.alpha.pregel.stream('myGraph', {
  maxIterations: 10,
  aggregator: 'single',
  defaultValue: 0.0
})
YIELD nodeId, values
RETURN gds.util.asNode(nodeId).name AS name, values

-- Pregel with message parsing
CALL gds.alpha.pregel.write('myGraph', {
  maxIterations: 20,
  writeProperty: 'pregelResult'
})
YIELD nodePropertiesWritten
RETURN nodePropertiesWritten
```

Common GDS Patterns

```
-- Run algorithm on filtered subgraph
CALL gds.graph.project.subgraph(
  'filteredGraph',
  'myGraph',
  'n.age > 25',
  '*'
)

-- Run multiple algorithms and combine results
CALL gds.pageRank.stream('myGraph')
YIELD nodeId, score AS pagerank
WITH nodeId, pagerank
CALL gds.degree.stream('myGraph')
YIELD nodeId AS degreeNodeId, score AS degree
WHERE nodeId = degreeNodeId
RETURN
  gds.util.asNode(nodeId).name AS name,
  pagerank,
  degree

-- Export algorithm results to another graph
CALL gds.pageRank.mutate('myGraph', {
  mutateProperty: 'pagerank'
})
YIELD nodePropertiesWritten
WITH 1 AS _
CALL gds.graph.project(
  'enrichedGraph',
  '*',
  '*',
  {
    nodeProperties: ['pagerank'],
    relationshipProperties: []
  }
```

```
)
RETURN 'Graph created with PageRank' AS result
```

Quick Reference

Node Syntax

```
(n)           -- Any node
(n:Label)     -- Node with label
(n:Label {prop: value})  -- Node with label and property
(n:L1:L2)     -- Node with multiple labels
```

Relationship Syntax

```
-[r]->        -- Outgoing relationship
-[r:TYPE]->   -- Typed outgoing relationship
-[r:TYPE*]->  -- Variable length
-[r:TYPE*2..5]->  -- Min 2, max 5 hops
```

Common Patterns

```
-- Create pattern
CREATE (a)-[:REL]->(b)

-- Match pattern
MATCH (a)-[:REL]->(b)

-- Merge pattern (create if not exists)
MERGE (a)-[:REL]->(b)

-- Return path
MATCH p = (a)-[:REL*]->(b)
RETURN p
```

Next Steps

- Monitoring Cheat Sheet — Prometheus/Grafana for graph workloads
- Docker CLI Cheat Sheet — Containerize Neo4j deployments
- Kubernetes kubectl Cheat Sheet — Deploy Neo4j on Kubernetes