

# MANUAL COMPLETO NEO4J: GUIA EXTENSO E CIENTÍFICO

---

## Índice Geral

1. [Introdução a Neo4j](#)
  2. [Conceitos Fundamentais](#)
  3. [Sintaxe Cypher](#)
  4. [Cláusulas Principais](#)
  5. [Operadores e Predicados](#)
  6. [Funções Predefinidas](#)
  7. [Consultas Avançadas](#)
  8. [Boas Práticas](#)
- 

## INTRODUÇÃO A NEO4J

### O que é Neo4j?

Neo4j é um **banco de dados de grafos** (graph database) altamente otimizado, open-source e empresarial, desenvolvido para armazenar, gerenciar e consultar dados altamente conectados. Ao contrário dos bancos de dados relacionais tradicionais que usam tabelas e chaves estrangeiras, Neo4j usa uma estrutura de grafos nativa onde:

- **Nós (Nodes)** representam entidades (pessoas, produtos, lugares)
- **Relações (Relationships)** representam conexões entre nós
- **Propriedades (Properties)** armazenam atributos de nós e relações
- **Labels** categorizam nós em tipos específicos

### Características Principais

#### Vantagens de Neo4j:

1. **Performance em Consultas de Relacionamento:** Grafos nativos permitem traversals (travessias) extremamente rápidas
2. **Escalabilidade:** Suporta bilhões de nós e relações
3. **Segurança:** Role-based access control (RBAC), encriptação
4. **ACID Compliance:** Garante transações seguras
5. **Cypher Query Language:** Linguagem declarativa intuitiva e poderosa
6. **Enterprise Features:** Clustering, backup, alta disponibilidade

### Quando Usar Neo4j?

#### ✅ Adequado para:

- Redes sociais e conexões entre usuários
- Sistemas de recomendação

- Análise de fraude e detecção de padrões
- Gestão de conhecimento e ontologias
- Sistemas de viagem e logística
- Análise de impacto de dependências

### ✗ Não adequado para:

- Dados sem relações significativas
- Processamento em lote simples
- Séries temporais tradicionais
- Dados altamente estruturados em tabelas

---

## CONCEITOS FUNDAMENTAIS

### 1. Nós (Nodes)

Um **nó** é uma entidade básica no grafo. Cada nó pode ter:

Estrutura de um Nó:

NODE (Nó)
Labels: :Person :Employee
Propriedades: <ul style="list-style-type: none"><li>• name: "João Silva"</li><li>• age: 35</li><li>• email: "joao@email.com"</li><li>• department: "IT"</li></ul>
ID interno: 123

#### Características:

- Cada nó recebe um ID único interno (não manipulável diretamente)
- Um nó pode ter **zero ou mais labels**
- Um nó pode ter **zero ou mais propriedades**
- Labels são case-sensitive
- Propriedades são pares chave-valor

### 2. Relações (Relationships)

Uma **relação** (ou edge) conecta dois nós e tem sempre:

- Um nó de início
- Um nó de fim
- Um tipo (label)
- Uma direção

Estrutura de uma Relação:

RELATIONSHIP (Relação)
Tipo: :WORKS_FOR
Propriedades: <ul style="list-style-type: none"><li>• since: 2020</li><li>• position: "Senior Developer"</li><li>• salary: 5000</li></ul>
Direção: (Person) -[:WORKS_FOR]-> (Company) ID interno: 456

Tipos de Relações:

- **Direcionadas:** (A) -[:REL]-> (B) [de A para B]
- **Bidirecionais:** (A) -[:REL]- (B) [ignora direção]
- **Reversas:** (A) <-[:REL]- (B) [de B para A]

3. Propriedades (Properties)

Propriedades armazenam dados específicos. Neo4j suporta os seguintes tipos:

Tipo	Descrição	Exemplo
String	Texto Unicode	"João Silva"
Integer	Número inteiro (64-bit)	42, -100
Float	Número decimal (64-bit)	3.14, -2.5
Boolean	Verdadeiro/Falso	true, false
Date	Data sem hora	date('2025-12-12')
DateTime	Data com hora e timezone	datetime('2025-12-12T15:30:00Z')
LocalDateTime	Data com hora sem timezone	localdatetime('2025-12-12T15:30:00')
Time	Hora com timezone	time('15:30:00+01:00')
Duration	Período de tempo	duration('P1Y2M10DT2H30M')
Point	Coordenada geoespacial	point({x:1.0, y:2.0})
List	Coleção de valores	[1, 2, 3] ou ['a', 'b', 'c']
Null	Valor ausente	null
Map	Estrutura chave-valor	{name: 'João', age: 30}

Exemplos de Propriedades:

```
-- Nó com várias propriedades
CREATE (p:Person {
  name: 'João Silva',
  age: 35,
  email: 'joao@email.com',
  active: true,
  joinDate: date('2020-01-15'),
  tags: ['developer', 'java', 'python'],
  metadata: {city: 'Lisboa', country: 'Portugal'}}
})
```

## 4. Labels

Labels são **categorias ou tipos** que um nó pode ter:

```
-- Um nó pode ter múltiplos labels
(p:Person:Developer:Manager)

-- Labels agrupam nós logicamente
CREATE (p:Person:Developer {name: 'Alice', language: 'Python'})
CREATE (c:Company:Technology {name: 'TechCorp'})
```

### Usos de Labels:

- Categorizar diferentes tipos de entidades
- Melhorar performance (indexação)
- Simplificar queries (filtrar por tipo)
- Estruturar o domínio da aplicação

## 5. Paths (Caminhos)

Um **path** (caminho) é uma sequência conectada de nós e relações:

```
Exemplo de Path:
(Person) -[:KNOWS]-> (Person) -[:WORKS_FOR]-> (Company) -[:LOCATED_IN]->
(City)

Path representando:
Alice -[conhece]-> Bob -[trabalha em]-> TechCorp -[localizada em]-> Lisboa
```

---

# SINTAXE CYPHER

## Introdução a Cypher

**Cypher** é a linguagem de consulta declarativa de Neo4j, similar a SQL mas otimizada para grafos. A sintaxe usa **ASCII-Art** para representar visualmente os padrões de grafos.

## Convenções de Sintaxe

```
-- MAIÚSCULAS: Palavras-chave Cypher (MATCH, CREATE, RETURN, etc.)
-- variáveis: nomes de variáveis usadas em queries (minúsculas)
-- (n) = nó com variável 'n'
-- [r] = relação com variável 'r'
-- {prop: valor} = propriedades
-- :Label = label de nó ou tipo de relação

-- Bom:
MATCH (person:Person {name: 'João'})
WHERE person.age > 30
RETURN person.name, person.age

-- Ruim (sintaticamente correto mas difícil de ler):
match(person:Person{name:'João'})where person.age>30return
person.name,person.age
```

## Estrutura Geral de uma Query

```
[USE database_name]

[MATCH pattern]

[WHERE conditions]

[WITH variables]

[OPTIONAL MATCH pattern]

[CREATE|MERGE|DELETE|SET]

[RETURN variables]

[ORDER BY expression]

[SKIP n]

[LIMIT n]
```

### Regras Importantes:

1. RETURN é obrigatório (exceto em queries de mutação puras)
2. Cláusulas são processadas em ordem
3. WHERE refina MATCH/OPTIONAL MATCH (não é um filtro pós-query)

#### 4. WITH encadeia queries (pipeline)

---

## CLÁUSULAS PRINCIPAIS

### 1. MATCH

A cláusula **MATCH** encontra padrões no grafo.

#### Sintaxe Básica

```
MATCH (n:Label {property: value})  
RETURN n
```

#### Exemplos

##### Exemplo 1: Buscar um nó específico

```
MATCH (p:Person {name: 'João'})  
RETURN p
```

##### Exemplo 2: Buscar múltiplos nós

```
MATCH (p:Person), (c:Company)  
RETURN p, c  
LIMIT 10
```

##### Exemplo 3: Buscar com relação

```
MATCH (p:Person)-[:WORKS_FOR]->(c:Company)  
RETURN p.name, c.name
```

##### Exemplo 4: Buscar relações em ambas as direções

```
-- Relação em qualquer direção  
MATCH (p1:Person)-[:KNOWS]-(p2:Person)  
RETURN p1.name, p2.name  
  
-- Relação específica  
MATCH (p1:Person)-[:KNOWS]->(p2:Person)  
RETURN p1.name AS person1, p2.name AS person2
```

Exemplo 5: Padrões complexos (paths)

```
-- Caminho de 2 hops
MATCH (a:Person)-[:KNOWS]->(b:Person)-[:KNOWS]->(c:Person)
RETURN a.name, b.name, c.name

-- Padrão com propriedades na relação
MATCH (p1:Person)-[r:KNOWS {since: 2020}]->(p2:Person)
RETURN p1, r, p2
```

Exemplo 6: Buscar nós isolados

```
-- Nós sem relações específicas
MATCH (p:Person)
WHERE NOT (p)-[:KNOWS]->(other:Person)
RETURN p
```

Exemplo 7: Relações com propriedades

```
MATCH (p:Person)-[r:WORKED_FOR]->(c:Company)
RETURN p.name, c.name, r.years, r.salary
```

2. WHERE

A cláusula **WHERE** filtra resultados com condições lógicas.

Sintaxe

```
WHERE condition1 AND condition2 OR condition3
```

Operadores de Comparação

Operador	Descrição	Exemplo
=	Igual	WHERE p.age = 30
<>	Diferente	WHERE p.age <> 30
<	Menor que	WHERE p.age < 30
>	Maior que	WHERE p.age > 30
<=	Menor ou igual	WHERE p.age <= 30
>=	Maior ou igual	WHERE p.age >= 30

## Operadores Lógicos

```
-- AND (e)
WHERE p.age > 25 AND p.age < 35

-- OR (ou)
WHERE p.department = 'IT' OR p.department = 'HR'

-- NOT (não)
WHERE NOT p.active

-- Combinação
WHERE (p.age > 25 AND p.salary > 3000) OR p.department = 'Executive'
```

## Exemplos de WHERE

### Exemplo 1: Filtro simples

```
MATCH (p:Person)
WHERE p.age > 30
RETURN p.name, p.age
```

### Exemplo 2: Múltiplas condições

```
MATCH (p:Person)
WHERE p.age > 25 AND p.department = 'IT' AND p.active = true
RETURN p
```

### Exemplo 3: Verificar existência de propriedade

```
MATCH (p:Person)
WHERE p.email IS NOT NULL
RETURN p.name, p.email
```

### Exemplo 4: Filtro com listas

```
MATCH (p:Person)
WHERE p.department IN ['IT', 'HR', 'Finance']
RETURN p
```

### Exemplo 5: Padrão de string (LIKE)

```
-- Contém substring
MATCH (p:Person)
WHERE p.name CONTAINS 'Silva'
RETURN p

-- Começa com
MATCH (p:Person)
WHERE p.email STARTS WITH 'joao'
RETURN p

-- Termina com
MATCH (p:Person)
WHERE p.email ENDS WITH '@email.com'
RETURN p
```

### Exemplo 6: Expressões regulares

```
MATCH (p:Person)
WHERE p.email =~ '.*@(email|gmail)\.com'
RETURN p
```

### Exemplo 7: Verificar tipo de dados

```
MATCH (n)
WHERE n.value IS NOT NULL AND n.value > 100
RETURN n
```

### Exemplo 8: Existência de relação

```
MATCH (p:Person)
WHERE (p)-[:WORKS_FOR]->(c:Company)
RETURN p
```

## 3. RETURN

A cláusula **RETURN** especifica os dados a retornar.

### Sintaxe

```
RETURN expressão1, expressão2, ...
```

### Exemplos

### Exemplo 1: Retornar nós inteiros

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

### Exemplo 2: Retornar propriedades específicas

```
MATCH (p:Person)
RETURN p.name, p.age, p.email
```

### Exemplo 3: Renomear com AS (alias)

```
MATCH (p:Person)-[:WORKS_FOR]->(c:Company)
RETURN p.name AS personName, c.name AS companyName, p.salary AS salary
```

### Exemplo 4: DISTINCT (valores únicos)

```
MATCH (p:Person)
RETURN DISTINCT p.department
```

### Exemplo 5: Expressões no RETURN

```
MATCH (p:Person)
RETURN p.name, p.salary * 1.1 AS newSalary, p.age + 1 AS nextAge
```

### Exemplo 6: Condicional no RETURN

```
MATCH (p:Person)
RETURN p.name,
CASE
  WHEN p.age < 30 THEN 'Junior'
  WHEN p.age < 50 THEN 'Senior'
  ELSE 'Executive'
END AS level
```

## 4. CREATE

A cláusula **CREATE** cria novos nós e relações.

### Criar Nó Simples

```
CREATE (n:Label {property: value})  
RETURN n
```

### Exemplo 1: Criar um nó

```
CREATE (p:Person {name: 'João', age: 30, email: 'joao@email.com'})  
RETURN p
```

### Exemplo 2: Criar múltiplos nós

```
CREATE  
  (p1:Person {name: 'João', age: 30}),  
  (p2:Person {name: 'Maria', age: 28}),  
  (p3:Person {name: 'Pedro', age: 35})  
RETURN p1, p2, p3
```

### Exemplo 3: Criar nó com múltiplos labels

```
CREATE (p:Person:Developer {name: 'Alice', language: 'Python'})  
RETURN p
```

### Exemplo 4: Criar relação entre nós existentes

```
CREATE (p1:Person {name: 'João'})-[:KNOWS]->(p2:Person {name: 'Maria'})  
RETURN p1, p2
```

### Exemplo 5: Criar relação com propriedades

```
CREATE (p1:Person {name: 'João'})-[:KNOWS {since: 2020, strength:  
  'strong'}]->(p2:Person {name: 'Maria'})  
RETURN p1, p2
```

### Exemplo 6: Criar nó com relação para nó existente

```
MATCH (c:Company {name: 'TechCorp'})  
CREATE (p:Person {name: 'Nova Pessoa'})-[:WORKS_FOR]->(c)  
RETURN p, c
```

**Importante:** CREATE sempre cria novos nós/relações, não verifica duplicatas.

## 5. MERGE

A cláusula **MERGE** combina MATCH e CREATE - procura o padrão e cria se não existir.

### Sintaxe

```
MERGE (n:Label {key_property: value})  
[ON CREATE SET prop = value]  
[ON MATCH SET prop = value]
```

### Exemplos

#### Exemplo 1: MERGE simples

```
MERGE (p:Person {email: 'joao@email.com'})  
SET p.name = 'João', p.age = 30  
RETURN p
```

#### Exemplo 2: MERGE com ON CREATE

```
MERGE (p:Person {email: 'maria@email.com'})  
ON CREATE SET p.name = 'Maria', p.age = 25, p.createdAt = datetime()  
RETURN p
```

#### Exemplo 3: MERGE com ON MATCH

```
MERGE (p:Person {email: 'joao@email.com'})  
ON MATCH SET p.lastLogin = datetime(), p.loginCount =  
coalesce(p.loginCount, 0) + 1  
RETURN p
```

#### Exemplo 4: MERGE com ON CREATE e ON MATCH

```
MERGE (p:Person {email: 'joao@email.com'})  
ON CREATE SET p.name = 'João', p.createdAt = datetime()  
ON MATCH SET p.lastUpdated = datetime(), p.updateCount =  
coalesce(p.updateCount, 0) + 1  
RETURN p
```

#### Exemplo 5: MERGE em relações

```
MERGE (p1:Person {name: 'João'})-[r:KNOWS {since: 2020}]->(p2:Person
{name: 'Maria'})
ON CREATE SET r.createdAt = datetime()
RETURN p1, r, p2
```

### Exemplo 6: MERGE com padrão complexo

```
MERGE (p:Person {email: 'joao@email.com'})
-[:WORKS_FOR {since: 2020}]->
(c:Company {name: 'TechCorp'})
ON CREATE SET p.createdAt = datetime(), c.createdAt = datetime()
RETURN p, c
```

## 6. SET

A cláusula **SET** atualiza propriedades de nós ou relações.

### Sintaxe

```
SET n.property = value
SET n.property1 = value1, n.property2 = value2
SET n += {property1: value1, property2: value2}
```

### Exemplos

#### Exemplo 1: Atualizar uma propriedade

```
MATCH (p:Person {name: 'João'})
SET p.age = 31
RETURN p
```

#### Exemplo 2: Atualizar múltiplas propriedades

```
MATCH (p:Person {name: 'João'})
SET p.age = 31, p.salary = 5000, p.active = true
RETURN p
```

#### Exemplo 3: Usar merge operator (+=)

```
MATCH (p:Person {name: 'João'})
SET p += {age: 31, salary: 5000, city: 'Lisboa'}
```

```
RETURN p
```

#### Exemplo 4: Incrementar propriedade

```
MATCH (p:Person {name: 'João'})
SET p.loginCount = p.loginCount + 1
RETURN p
```

#### Exemplo 5: Adicionar label a nó existente

```
MATCH (p:Person {name: 'João'})
SET p:Developer
RETURN p
```

#### Exemplo 6: Atualizar relação

```
MATCH (p:Person {name: 'João'})-[r:WORKS_FOR]->(c:Company)
SET r.years = 5, r.salary = 4500
RETURN p, r, c
```

#### Exemplo 7: Copiar propriedade de outra

```
MATCH (p:Person)
SET p.fullName = p.name
RETURN p
```

## 7. DELETE e DETACH DELETE

A cláusula **DELETE** remove nós e relações. **DETACH DELETE** remove um nó e todas as suas relações.

### Sintaxe

```
DELETE n                -- Deleta nó (erro se tiver relações)
DETACH DELETE n         -- Deleta nó e todas as relações
DELETE r                -- Deleta relação específica
```

### Exemplos

#### Exemplo 1: Deletar nó sem relações

```
MATCH (p:Person {name: 'João'})
DELETE p
```

### Exemplo 2: Deletar nó com relações

```
MATCH (p:Person {name: 'João'})
DETACH DELETE p
```

### Exemplo 3: Deletar relação específica

```
MATCH (p:Person {name: 'João'})-[r:KNOWS]->(other:Person)
DELETE r
RETURN p, other
```

### Exemplo 4: Deletar múltiplos nós

```
MATCH (p:Person)
WHERE p.age < 18
DETACH DELETE p
```

### Exemplo 5: Deletar todos os nós e relações (cuidado!)

```
MATCH (n)
DETACH DELETE n
```

## 8. REMOVE

A cláusula **REMOVE** remove labels ou propriedades.

### Sintaxe

```
REMOVE n.property      -- Remove propriedade
REMOVE n:Label          -- Remove label
```

### Exemplos

#### Exemplo 1: Remover propriedade

```
MATCH (p:Person {name: 'João'})
REMOVE p.age
RETURN p
```

### Exemplo 2: Remover múltiplas propriedades

```
MATCH (p:Person {name: 'João'})
REMOVE p.age, p.email
RETURN p
```

### Exemplo 3: Remover label

```
MATCH (p:Person:Developer {name: 'João'})
REMOVE p:Developer
RETURN p
```

### Exemplo 4: Remover múltiplos labels

```
MATCH (p:Person:Developer:Manager)
REMOVE p:Developer, p:Manager
RETURN p
```

## 9. WITH

A cláusula **WITH** encadeia consultas, passando resultados de uma parte para a próxima.

### Sintaxe

```
MATCH pattern
WITH variable [ORDER BY] [SKIP] [LIMIT]
MATCH otherPattern
RETURN variable
```

### Exemplos

#### Exemplo 1: Filtrar antes de segunda busca

```
MATCH (p:Person)
WHERE p.age > 25
WITH p
```

```
MATCH (p)-[:WORKS_FOR]->(c:Company)
RETURN p.name, c.name
```

### Exemplo 2: Agregar e depois filtrar

```
MATCH (p:Person)-[:WORKS_FOR]->(c:Company)
WITH c, count(p) AS employees
WHERE employees > 5
RETURN c.name, employees
```

### Exemplo 3: Encadear múltiplas WITH

```
MATCH (p:Person)
WITH p WHERE p.age > 25
WITH p ORDER BY p.salary DESC
LIMIT 10
RETURN p.name, p.salary
```

### Exemplo 4: Coletar e processar lista

```
MATCH (p:Person)-[:KNOWS]->(friend:Person)
WITH p, collect(friend.name) AS friends
RETURN p.name, friends
```

## 10. OPTIONAL MATCH

A cláusula **OPTIONAL MATCH** é como um LEFT OUTER JOIN - retorna nulo se não houver match.

### Sintaxe

```
MATCH pattern
OPTIONAL MATCH otherPattern
RETURN variables
```

### Exemplos

#### Exemplo 1: OPTIONAL MATCH simples

```
MATCH (p:Person)
OPTIONAL MATCH (p)-[:WORKS_FOR]->(c:Company)
RETURN p.name, c.name
```

## Exemplo 2: Múltiplas OPTIONAL MATCH

```
MATCH (p:Person)
OPTIONAL MATCH (p)-[:KNOWS]->(friend:Person)
OPTIONAL MATCH (p)-[:WORKS_FOR]->(company:Company)
RETURN p.name, friend.name, company.name
```

## Exemplo 3: OPTIONAL MATCH com WHERE

```
MATCH (p:Person)
OPTIONAL MATCH (p)-[:KNOWS]->(friend:Person) WHERE friend.age > 30
RETURN p.name, friend.name
```

## 11. UNWIND

A cláusula **UNWIND** transforma uma lista em linhas individuais.

### Sintaxe

```
WITH [value1, value2, value3] AS list
UNWIND list AS item
RETURN item
```

### Exemplos

#### Exemplo 1: UNWIND lista simples

```
WITH [1, 2, 3, 4, 5] AS numbers
UNWIND numbers AS num
RETURN num * 2 AS doubled
```

### Resultado:

```
doubled
2
4
6
8
10
```

#### Exemplo 2: UNWIND com criação de nós

```
WITH ['João', 'Maria', 'Pedro'] AS names
UNWIND names AS name
CREATE (p:Person {name: name})
RETURN p
```

### Exemplo 3: UNWIND com mapa

```
WITH [{name: 'João', age: 30}, {name: 'Maria', age: 28}] AS people
UNWIND people AS person
CREATE (p:Person {name: person.name, age: person.age})
RETURN p
```

### Exemplo 4: UNWIND para explodir lista de propriedade

```
MATCH (p:Person)
UNWIND p.hobbies AS hobby
RETURN p.name, hobby
```

## 12. ORDER BY, SKIP, LIMIT

Estas cláusulas controlam a ordenação e paginação dos resultados.

### Sintaxe

```
RETURN variables
ORDER BY expression [ASC|DESC]
SKIP n
LIMIT n
```

### Exemplos

#### Exemplo 1: ORDER BY simples

```
MATCH (p:Person)
RETURN p.name, p.age
ORDER BY p.age ASC
```

#### Exemplo 2: ORDER BY descendente

```
MATCH (p:Person)
RETURN p.name, p.salary
```

```
ORDER BY p.salary DESC
```

### Exemplo 3: ORDER BY múltiplas colunas

```
MATCH (p:Person)
RETURN p.department, p.name, p.salary
ORDER BY p.department ASC, p.salary DESC
```

### Exemplo 4: SKIP e LIMIT (paginação)

```
MATCH (p:Person)
RETURN p
ORDER BY p.name
SKIP 10
LIMIT 20
```

### Exemplo 5: Paginação com ORDER BY

```
-- Página 3, com 10 registos por página
MATCH (p:Person)
RETURN p.name, p.age
ORDER BY p.name
SKIP 20
LIMIT 10
```

---

## OPERADORES E PREDICADOS

### Operadores Aritméticos

Operador	Descrição	Exemplo
+	Adição	$5 + 3 = 8$
-	Subtração	$5 - 3 = 2$
*	Multiplicação	$5 * 3 = 15$
/	Divisão	$15 / 3 = 5$
%	Módulo (resto)	$17 \% 5 = 2$
^	Potência	$2 ^ 3 = 8$

```
-- Exemplos
MATCH (p:Person)
```

```
RETURN p.name,  
       p.salary * 1.1 AS newSalary,  
       p.age - 1 AS previousAge,  
       p.salary / 12 AS monthlySalary
```

## Operadores de Comparação

Operador	Descrição
=	Igual
<>	Não igual
<	Menor que
<=	Menor ou igual
>	Maior que
>=	Maior ou igual
IN	Está em lista
IS NULL	É nulo
IS NOT NULL	Não é nulo

```
-- Exemplos  
WHERE p.age > 25  
WHERE p.department IN ['IT', 'HR', 'Finance']  
WHERE p.email IS NOT NULL  
WHERE p.age BETWEEN 25 AND 35
```

## Operadores Lógicos

Operador	Descrição	Exemplo
AND	E lógico	a AND b
OR	OU lógico	a OR b
NOT	Negação	NOT a
XOR	OU exclusivo	a XOR b

```
WHERE (p.age > 25 AND p.department = 'IT') OR p.role = 'Manager'  
WHERE NOT p.active
```

## Operadores de String

Operador	Descrição	Exemplo
+	Concatenação	"Hello" + " " + "World"
CONTAINS	Contém substring	p.name CONTAINS 'Silva'
STARTS WITH	Começa com	p.email STARTS WITH 'joao'
ENDS WITH	Termina com	p.email ENDS WITH '@email.com'
=~	Regex match	p.email =~ '.*@email\.com'

```
-- Exemplos
RETURN "Hello" + " " + "World"           -- "Hello World"
WHERE p.name CONTAINS 'Silva'
WHERE p.email STARTS WITH 'joao'
WHERE p.code =~ '[A-Z]{3}[0-9]{3}'
```

## Operador IN

```
-- Verificar se valor está em lista
WHERE p.department IN ['IT', 'HR', 'Finance']

-- Verificar se lista contém elemento
WHERE 'IT' IN p.departments

-- Iterar sobre propriedades de nó
MATCH (n)
WHERE n.status IN ['active', 'pending']
RETURN n
```

## Predicados de Relação

```
-- Verificar existência de relação
WHERE (p)-[:KNOWS]->()
WHERE (p)-[:MANAGED_BY]-()

-- Verificar relação específica
MATCH (p:Person)
WHERE (p)-[:WORKS_FOR {active: true}]->(:Company)
RETURN p

-- Negar relação
WHERE NOT (p)-[:KNOWS]->(:Person)
```

## FUNÇÕES PREDEFINIDAS

Neo4j oferece centenas de funções predefinidas para manipular dados. Vamos cobrir as mais importantes.

## Funções de Agregação

As funções de agregação operam sobre conjuntos de valores e retornam um único resultado.

### COUNT()

Conta o número de linhas.

```
-- Contar todos os resultados
MATCH (p:Person)
RETURN count(p) AS totalPessoas

-- Contar com DISTINCT
MATCH (p:Person)
RETURN count(DISTINCT p.department) AS departments

-- Contar nulos/não-nulos
MATCH (p:Person)
RETURN count(p.age) AS peopleWithAge
```

#### Resultado:

```
totalPessoas
5
```

### SUM()

Soma valores.

```
MATCH (p:Person)-[:WORKS_FOR]->(c:Company)
RETURN c.name, sum(p.salary) AS totalPayroll
```

### AVG()

Calcula a média.

```
MATCH (p:Person)
RETURN avg(p.age) AS averageAge

-- Com tipo conversion
MATCH (p:Person)
RETURN avg(toFloat(p.salary)) AS averageSalary
```

## MIN() e MAX()

Encontram o mínimo e máximo.

```
MATCH (p:Person)
RETURN min(p.age) AS youngest, max(p.age) AS oldest

MATCH (p:Person)-[:WORKS_FOR]->(c:Company)
RETURN c.name, min(p.salary) AS minSalary, max(p.salary) AS maxSalary
```

## COLLECT()

Coleta valores em uma lista.

```
-- Coletar nós
MATCH (p:Person)-[:WORKS_FOR]->(c:Company)
RETURN c.name, collect(p.name) AS employees

-- Coletar com DISTINCT
MATCH (p:Person)
RETURN collect(DISTINCT p.department) AS departments

-- Resultado:
-- {c.name: "TechCorp", employees: ["João", "Maria", "Pedro"]}
```

## GROUP BY

Agrupamento implícito com agregação.

```
-- Agrupar por departamento
MATCH (p:Person)
RETURN p.department, count(p) AS employees, avg(p.salary) AS avgSalary
ORDER BY avgSalary DESC
```

## PERCENTILECONT() e PERCENTILEDISC()

Calcula percentis.

```
MATCH (p:Person)
RETURN percentileCont(p.salary, 0.5) AS median,
       percentileCont(p.salary, 0.95) AS percentile95
```

## STDEV() e STDEV.P()

Desvio padrão (amostra e população).

```
MATCH (p:Person)
RETURN stdev(p.salary) AS stdDeviation
```

## Funções de Escalar

Funções escalares operam sobre valores individuais.

### STRING Functions

#### toUpper() e toLower()

```
MATCH (p:Person)
RETURN toUpper(p.name) AS upperName, toLower(p.name) AS lowerName
```

#### substring()

```
WITH "Hello World" AS text
RETURN substring(text, 0, 5) -- "Hello"
RETURN substring(text, 6)   -- "World"
```

#### length()

```
MATCH (p:Person)
RETURN p.name, length(p.name) AS nameLength
WHERE length(p.name) > 5
```

#### trim(), ltrim(), rtrim()

```
WITH " Hello " AS text
RETURN trim(text)      -- "Hello"
RETURN ltrim(text)     -- "Hello "
RETURN rtrim(text)     -- " Hello"
```

#### split()

```
WITH "João Silva" AS fullName
RETURN split(fullName, " ") -- ["João", "Silva"]
```

**replace()**

```
WITH "Hello World" AS text
RETURN replace(text, "World", "Neo4j") -- "Hello Neo4j"
```

**reverse()**

```
WITH "Hello" AS text
RETURN reverse(text) -- "olleH"
```

**NUMERIC Functions****abs() - Valor absoluto**

```
RETURN abs(-10) AS resultado -- 10
```

**ceil() e floor()**

```
RETURN ceil(3.2) -- 4
RETURN floor(3.8) -- 3
```

**round()**

```
RETURN round(3.14159, 2) -- 3.14
```

**sqrt() - Raiz quadrada**

```
RETURN sqrt(16) -- 4.0
```

**randomInteger() e rand()**

```
RETURN randomInteger(100) -- Número aleatório entre 0 e 99
RETURN rand() -- Float entre 0 e 1
```

**sign()**

```
RETURN sign(-5)    -- -1
RETURN sign(5)     -- 1
RETURN sign(0)     -- 0
```

## MATHEMATICAL Functions

```
RETURN sin(1.0)
RETURN cos(0)
RETURN tan(1.0)
RETURN log(10)
RETURN log10(100)
RETURN exp(1)
```

## Funções de Lista

### size()

Retorna o comprimento de uma lista.

```
WITH [1, 2, 3, 4, 5] AS numbers
RETURN size(numbers) -- 5

MATCH (p:Person)
RETURN p.name, size(p.hobbies) AS hobbyCount
```

### RANGE()

Cria uma lista de números.

```
RETURN range(1, 5)           -- [1, 2, 3, 4, 5]
RETURN range(0, 10, 2)       -- [0, 2, 4, 6, 8, 10]
RETURN range(10, 1, -2)      -- [10, 8, 6, 4, 2]
```

## Acesso a elementos

```
WITH [10, 20, 30, 40, 50] AS numbers
RETURN numbers[0]           -- 10 (primeiro elemento)
RETURN numbers[2]           -- 30 (terceiro elemento)
RETURN numbers[-1]          -- 50 (último elemento)
RETURN numbers[1..3]        -- [20, 30, 40] (slice)
```

## head() e tail()

```
WITH [1, 2, 3, 4, 5] AS numbers
RETURN head(numbers)  -- 1 (primeiro)
RETURN tail(numbers)  -- [2, 3, 4, 5] (resto)
```

## REVERSE() em listas

```
WITH [1, 2, 3, 4, 5] AS numbers
RETURN reverse(numbers)  -- [5, 4, 3, 2, 1]
```

## FILTER()

Filtra lista baseada em condição.

```
WITH [1, 2, 3, 4, 5, 6] AS numbers
RETURN filter(x IN numbers WHERE x > 3)  -- [4, 5, 6]
```

## REDUCE()

Reduz lista a valor único.

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

-- Exemplo mais complexo
WITH [10, 20, 30] AS salaries
RETURN reduce(total = 0, salary IN salaries | total + salary) AS
totalSalary
```

## ALL(), ANY(), NONE(), SINGLE()

```
WITH [2, 4, 6, 8] AS numbers
RETURN all(x IN numbers WHERE x % 2 = 0)  -- true (todos pares)

WITH [2, 4, 5, 8] AS numbers
RETURN any(x IN numbers WHERE x % 2 = 1)  -- true (existe um ímpar)

WITH [1, 3, 5, 7] AS numbers
RETURN none(x IN numbers WHERE x % 2 = 0)  -- true (nenhum par)
```

```
WITH [1, 2, 2, 3] AS numbers
RETURN single(x IN numbers WHERE x = 2)  -- false (mais de um 2)
```

## Funções de Tipo

### **type()**

Retorna tipo de um valor.

```
WITH [1, "hello", 3.14, true, null] AS values
RETURN [v IN values | type(v)]
-- ["INTEGER", "STRING", "FLOAT", "BOOLEAN", "NULL"]
```

### **labels() e properties()**

```
MATCH (p:Person:Developer)
RETURN labels(p)          -- ["Person", "Developer"]

MATCH (p:Person)
RETURN properties(p)      -- {name: "João", age: 30, ...}
```

### **id()**

Retorna ID interno de um nó.

```
MATCH (p:Person)
RETURN id(p), p.name
```

### **keys()**

Retorna chaves de um map/propriedades.

```
MATCH (p:Person)
WITH properties(p) AS props
RETURN keys(props)  -- ["name", "age", "email", ...]
```

## Funções de Valor NULL

### **coalesce()**

Retorna o primeiro valor não-nulo.

```
MATCH (p:Person)
RETURN coalesce(p.nickname, p.name, "Unknown") AS displayName

-- Exemplo útil para defaults
RETURN coalesce(p.age, 0) AS age
```

### CASE (mencionado antes, mas importante)

```
MATCH (p:Person)
RETURN p.name,
CASE
  WHEN p.age IS NULL THEN 'Unknown'
  WHEN p.age < 18 THEN 'Minor'
  WHEN p.age < 65 THEN 'Adult'
  ELSE 'Senior'
END AS ageGroup
```

## Funções de Data e Hora

Neo4j suporta tipos temporais nativos.

### date()

```
-- Criar data
RETURN date('2025-12-12')
RETURN date({year: 2025, month: 12, day: 12})

-- Em query
MATCH (p:Person)
WHERE p.birthDate > date('2000-01-01')
RETURN p
```

### datetime()

```
-- Data e hora com timezone
RETURN datetime('2025-12-12T15:30:00Z')
RETURN datetime({year: 2025, month: 12, day: 12, hour: 15, minute: 30})

-- Hora atual
RETURN datetime() AS nowUtc
```

### localtime() e localtimedatetime()

```
-- Hora sem timezone
RETURN localtime()

-- Data e hora sem timezone
RETURN localdatetime()
```

Funções de Data

```
WITH date('2025-12-12') AS d
RETURN d.year, d.month, d.day, d.week, d.dayOfWeek, d.dayOfYear
```

Durações

```
-- Criar duração
RETURN duration('P1Y2M')           -- 1 ano 2 meses
RETURN duration('P1DT2H30M')       -- 1 dia 2h 30min
RETURN duration({days: 1, hours: 2, minutes: 30})

-- Aritmética de datas
WITH date('2025-01-01') AS startDate
RETURN startDate + duration('P1M') AS oneMonthLater
```

Funções de Tipo de Dado (Casting)

Função	Descrição	Exemplo
toString()	Converter para string	toString(123) → "123"
toInteger()	Converter para inteiro	toInteger("123") → 123
toFloat()	Converter para float	toFloat("3.14") → 3.14
toBoolean()	Converter para boolean	toBoolean("true") → true

```
MATCH (p:Person)
RETURN p.name,
       toString(p.age) AS ageStr,
       toFloat(p.salary) AS salaryFloat
```

Funções de Relação e Path

nodes() e relationships()

```
MATCH p = (a:Person)-[:KNOWS]-(b:Person)-[:WORKS_FOR]->(c:Company)
RETURN nodes(p) AS allNodes,
       relationships(p) AS allRelations

-- Resultado:
-- nodes(p): [<Person>, <Person>, <Company>]
-- relationships(p): [<KNOWS>, <WORKS_FOR>]
```

## length()

Comprimento de um path.

```
MATCH p = (a:Person)-[:KNOWS*1..3]-(b:Person)
WHERE length(p) = 2
RETURN a, b, length(p) AS distance
```

## last() e head()

```
MATCH p = (a)-[*]->(b)
RETURN head(nodes(p)) AS startNode,
       last(nodes(p)) AS endNode
```

## Funções de Texto Avançadas

### apoc.text (usando APOC)

```
-- Requer APOC instalado
RETURN apoc.text.uppercase('hello')
RETURN apoc.text.format('%s %s', ['Hello', 'World'])
```

---

## CONSULTAS AVANÇADAS

### Variable-Length Relationships

Busca caminhos de qualquer comprimento.

```
-- Exatamente 1 hop
MATCH (a:Person)-[:KNOWS]->(b:Person)
RETURN a, b

-- 1 a 3 hops
MATCH (a:Person)-[:KNOWS*1..3]-(b:Person)
```

```
RETURN a, b

-- Qualquer número de hops
MATCH (a:Person)-[:KNOWS*]-(b:Person)
RETURN a, b

-- Padrão negado (até 3, nenhum direto)
MATCH (a:Person)-[:KNOWS*2..3]->(b:Person)
WHERE a.name = 'Alice'
RETURN distinct b.name
```

## Shortest Path

Encontra o caminho mais curto entre dois nós.

```
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
MATCH p = shortestPath((a)-[*]-(b))
RETURN p, length(p) AS distance
```

## All Shortest Paths

```
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
MATCH p = allShortestPaths((a)-[*]-(b))
RETURN p
```

## Recursion with FOREACH

Processamento iterativo.

```
MATCH (n:Node {id: 1})
FOREACH (x IN range(1, 5) | CREATE (n)-[:HAS]->(m:Node {value: x}))
```

## UNION

Combina resultados de múltiplas queries.

```
-- UNION (remove duplicatas)
MATCH (p:Person) WHERE p.age > 30
RETURN p.name AS name

UNION

MATCH (m:Manager)
RETURN m.name AS name
```

```
-- UNION ALL (mantém duplicatas)
MATCH (p:Person) WHERE p.age > 30
RETURN p.name

UNION ALL

MATCH (p:Person) WHERE p.salary > 5000
RETURN p.name
```

## Conditional Relationships

```
-- Caminho condicional
MATCH (p:Person)-[r:KNOWS|WORKS_WITH]-(other:Person)
WHERE r.since > 2015
RETURN p, r, other

-- Com labels múltiplos
MATCH (n)-[r:WORKS_FOR|MANAGES|OWNS]->(m)
RETURN n, r, m
```

## Multi-hop with Different Relationships

```
MATCH (a:Person)
  -[:KNOWS]->(b:Person)
  -[:WORKS_FOR]->(c:Company)
  -[:LOCATED_IN]->(l:Location)
WHERE a.name = 'João'
RETURN b.name, c.name, l.name
```

---

# BOAS PRÁTICAS

## 1. Performance

### Usar Índices

```
CREATE INDEX ON :Person(email)
CREATE INDEX ON :Company(name)

-- Verificar índices
SHOW INDEXES
```

### Limitar Resultados

```
-- ✓ Bom
MATCH (p:Person) RETURN p LIMIT 100

-- x Ruim (sem limite)
MATCH (p:Person) RETURN p
```

## Usar EXPLAIN e PROFILE

```
-- Ver plano de execução
EXPLAIN MATCH (p:Person)-[:WORKS_FOR]->(c:Company) RETURN p

-- Ver custos reais
PROFILE MATCH (p:Person)-[:WORKS_FOR]->(c:Company) RETURN p
```

## 2. Naming Conventions

```
-- ✓ Bom: CamelCase para labels, UPPER_SNAKE_CASE para relações
CREATE (p:Person:Employee)-[:WORKS_FOR]->(c:Company)

-- ✓ Bom: lowercase para variáveis
MATCH (person:Person)-[:WORKS_FOR]->(company:Company)

-- ✓ Bom: Propriedades descritivas
{firstName: 'João', lastName: 'Silva', emailAddress: 'joao@email.com'}

-- x Evitar
CREATE (p:person:employee)-[:works_for]->(c:company)
```

## 3. Error Handling

```
-- Usar COALESCE para valores nulos
RETURN coalesce(p.age, 0) AS age

-- Usar CASE para lógica condicional
RETURN CASE
  WHEN p.status = 'active' THEN 1
  ELSE 0
END AS isActive

-- Validação
MATCH (p:Person)
WHERE p.email IS NOT NULL
RETURN p
```

## 4. Query Structure

```
-- ✓ Bom: Estruturado e legível
MATCH (person:Person)-[:WORKS_FOR]->(company:Company)
WHERE person.age > 25
      AND company.industry = 'Technology'
WITH company, count(person) AS employeeCount
WHERE employeeCount > 10
RETURN company.name, employeeCount
ORDER BY employeeCount DESC

-- ✗ Ruim: Tudo em uma linha
MATCH (p:Person)-[:WORKS_FOR]->(c:Company) WHERE p.age>25 AND
c.industry='Technology' WITH c, count(p) AS cnt WHERE cnt>10 RETURN
c.name, cnt
```

## 5. Transaction Management

```
-- Neo4j é auto-transacional, mas você pode:
-- Em aplicações client
BEGIN TRANSACTION
  // Múltiplas queries
COMMIT

-- Ou em caso de erro
ROLLBACK
```

## 6. Constraints

```
-- Constraint de unicidade
CREATE CONSTRAINT ON (p:Person) ASSERT p.email IS UNIQUE

-- Constraint de existência (Neo4j 4.1+)
CREATE CONSTRAINT ON (p:Person) ASSERT p.name IS NOT NULL

-- Listar constraints
SHOW CONSTRAINTS
```

---

# RESUMO E CONCLUSÃO

Este manual cobriu:

- ✓ **Conceitos Fundamentais:** Nós, relações, propriedades, labels
- ✓ **Sintaxe Cypher:** ASCII-art, structure de queries
- ✓ **Cláusulas Principais:** MATCH, CREATE, MERGE, DELETE, etc.
- ✓ **Operadores:** Aritméticos, lógicos, comparação, string
- ✓ **Funções:** Agregação, escalar, lista, data, tipo

✅ **Consultas Avançadas:** Variable-length, shortest path, UNION

✅ **Boas Práticas:** Performance, naming, error handling

Neo4j é uma ferramenta poderosa para trabalhar com dados altamente conectados. Pratique regularmente e sempre consulte a documentação oficial em <https://neo4j.com/docs/cypher-manual/>

---

**Manual Versão:** 1.0

**Neo4j Versão:** 5.x

**Data:** Dezembro 2025

**Autor:** Neo4j Learning Resource