Cypher Cheat Sheet

Read Query

Read Query Structure

```
[USE]
[MATCH [WHERE]]
[OPTIONAL MATCH [WHERE]]
[WITH [ORDER BY] [SKIP] [LIMIT] [WHERE]]
RETURN [ORDER BY] [SKIP] [LIMIT]
```

Baseline for pattern search operations.

- `USE` clause.
- `MATCH` clause.
- `OPTIONAL MATCH` clause.
- `WITH` clause.
- `RETURN` clause.
- Cypher keywords are not case-sensitive.
- Cypher is case-sensitive for variables.

AI search

MATCH

```
MATCH (n)
RETURN n
```

Match all nodes and return all nodes.

```
MATCH (movie:Movie)
RETURN movie.title
```

Find all nodes with the `Movie` label.

```
MATCH (:Person {name: 'Oliver Stone'})-
[r]->()
RETURN type(r) AS relType
```

Find the types of an aliased relationship.

```
MATCH (:Movie {title: 'Wall Street'})<-
[:ACTED_IN]-(actor:Person)
RETURN actor.name AS actor
```

Relationship pattern filtering on the `ACTED_IN` relationship type.

```
MATCH path = ()-[:ACTED_IN]->
(movie:Movie)
RETURN path
```

Bind a path pattern to a path variable, and return the path pattern.

```
MATCH (movie:$($label))
RETURN movie.title AS movieTitle
```

Node labels and relationship types can be referenced dynamically in expressions, parameters, and variables when matching nodes and relationships. The expression must evaluate to a `STRING NOT NULL | LIST<STRING NOT NULL> NOT NULL` value.

```
CALL db.relationshipTypes()
YIELD relationshipType
MATCH ()-[r:$(relationshipType)]->()
RETURN relationshipType, count(r) AS
relationshipCount
```

Match nodes dynamically using a variable.

OPTIONAL MATCH

```
MATCH (p:Person {name: 'Martin Sheen'})
OPTIONAL MATCH (p)-[r:DIRECTED]->()
RETURN p.name, r
```

Use `MATCH` to find entities that must be present in the pattern. Use `OPTIONAL MATCH` to find entities that may not be present in the pattern. `OPTIONAL MATCH` returns `null` for empty rows.

WHERE

```
MATCH (n)
WHERE n:Swedish
RETURN n.name AS name
```

`WHERE` used to filter on node labels.

```
MATCH (n:Person)
WHERE n.age < 35
RETURN n.name AS name, n.age AS age
```

`WHERE` used to filter on node properties.

```
MATCH (:Person {name:'Andy'})-[k:KNOWS]->
(f)
WHERE k.since < 2000
RETURN f.name AS oldFriend
```

`WHERE` used to filter on relationship properties.

```
MATCH (n:Person)
WHERE n[$propname] > 40
RETURN n.name AS name, n.age AS age
```

To filter on a property using a dynamically computed name, use square brackets `[]`.

```
WITH 35 AS minAge
MATCH (a:Person WHERE a.name = 'Andy')-
[:KNOWS]->(b:Person WHERE b.age > minAge)
RETURN b.name AS name
```

`WHERE` used inside a fixed-length pattern.

```
MATCH (a:Person {name: 'Andy'})
RETURN [(a)-->(b WHERE b:Person) |
b.name] AS friends
```

`WHERE` can appear inside a pattern comprehension.

```
MATCH p = (a:Person {name: "Andy"})-
[r:KNOWS WHERE r.since < 2011]->{1,4}
(:Person)
RETURN [n IN nodes(p) | n.name] AS paths
```

`WHERE` can be used to filter variable-length patterns.

FILTER

```
MATCH (n:Person)
FILTER n.age < 35
RETURN n.name AS name, n.age AS age
```

`FILTER` is used to add filters to queries, similar to Cypher's `WHERE`. Unlike `WHERE`, `FILTER` is not a subclause, which means it can be used independently of the `MATCH`, `OPTIONAL MATCH`, and `WITH` clauses, but not within them.

```
MATCH (n:Person)
FILTER n[$propname] > 40
RETURN n.name AS name, n.age AS age
```

`FILTER` on dynamic properties.

```
LOAD CSV WITH HEADERS FROM
'file:///companies.csv' AS row
FILTER row.Id IS NOT NULL
MERGE (c:Company {id: row.Id})
```

`FILTER` can be used as a substitute for the `WITH * WHERE <predicate>` constructs in Cypher.

RETURN

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

Return a node.

```
MATCH (p:Person {name: 'Keanu Reeves'})-
[r:ACTED_IN]->(m)
RETURN type(r)
```

Return relationship types.

```
MATCH (p:Person {name: 'Keanu Reeves'})
RETURN p.bornIn
```

Return a specific property.

```
MATCH p = (keanu:Person {name: 'Keanu
Reeves'})-[r]->(m)
RETURN *
```

To return all nodes, relationships and paths found in a query, use the `*` symbol.

```
MATCH (p:Person {name: 'Keanu Reeves'})
RETURN p.nationality AS citizenship
```

Names of returned columns can be aliased using the `AS` operator.

```
MATCH (p:Person {name: 'Keanu Reeves'})--
>(m)
RETURN DISTINCT m
```

`DISTINCT` retrieves unique rows for the returned columns.

The `RETURN` clause can use:

- ORDER BY
- SKIP
- LIMIT
- WHERE

WITH

```
MATCH (c:Customer)-[:BUYS]->(:Product
{name: 'Chocolate'})
WITH c AS customers
RETURN customers.firstName AS
chocolateCustomers
```

`WITH` can be used in combination with the `AS` keyword to bind new variables which can then be passed to subsequent clauses. Any variables not explicitly referenced by `WITH` (or carried over by `WITH *`) are dropped from the scope of the query.

```
MATCH (supplier:Supplier)-[r]->
(product:Product)
WITH *
RETURN supplier.name AS company,
       type(r) AS relType,
       product.name AS product
```

Use the wildcard `*` to carry over all variables that are in scope.

```
WITH 11 AS x
CALL (x) {
  UNWIND [2, 3] AS y
  WITH y
  RETURN x*y AS a
}
RETURN x, a
```

`WITH` cannot de-scope variables imported to a `CALL` subquery, because variables imported to a subquery are considered global to its inner scope.

```
MATCH (customer:Customer)-[:BUYS]->
(chocolate:Product {name: 'Chocolate'})
WITH customer.firstName || ' ' ||
customer.lastName AS customerFullName,
     chocolate.price * (1 -
customer.discount) AS chocolateNetPrice
RETURN customerFullName,
       chocolateNetPrice
```

`WITH` can be used to assign the values of expressions to variables.

```
MATCH (p:Product)
WITH p, p.price >= 500 AS isExpensive
WITH p, isExpensive, NOT isExpensive AS
isAffordable
WITH p, isExpensive, isAffordable,
     CASE
         WHEN isExpensive THEN 'High-end'
```

`WITH` can be used to chain expressions.

```
        ELSE 'Budget'
    END AS discountCategory
RETURN p.name AS product,
       p.price AS price,
       isAffordable,
       discountCategory
ORDER BY price
```

```
MATCH (c:Customer)-[:BUYS]->(p:Product)
WITH c.firstName AS customer,
     sum(p.price) AS totalSpent,
     collect(p.name) AS productsBought
RETURN customer,
       totalSpent,
       productsBought
ORDER BY totalSpent DESC
```

`WITH` can be used to perform aggregations and bind the results to new variables.

```
MATCH (c:Customer)
WITH DISTINCT c.discount AS discountRates
RETURN discountRates
ORDER BY discountRates
```

`WITH` can be used to remove duplicate values from the result set if appended with the modifier `DISTINCT`.

```
MATCH (c:Customer)-[:BUYS]->(p:Product)
WITH c,
     sum(p.price) AS totalSpent
  ORDER BY totalSpent DESC
  LIMIT 3
SET c.topSpender = true
RETURN c.firstName AS customer,
       totalSpent,
       c.topSpender AS topSpender
```

`WITH` can order and paginate results if used together with the `ORDER BY`, `LIMIT`, and `SKIP` subclauses.

```
MATCH (s:Supplier)-[:SUPPLIES]->
(p:Product)<-[:BUYS]-(c:Customer)
WITH s,
     sum(p.price) AS totalSales,
     count(DISTINCT c) AS uniqueCustomers
  WHERE totalSales > 1000
RETURN s.name AS supplier,
       totalSales,
       uniqueCustomers
```

`WITH` can be followed by the `WHERE` subclause to filter results.

LET

```
MATCH (s:Supplier)-[:SUPPLIES]->
(p:Product)
LET supplier = s.name, product = p.name
RETURN supplier, product
```

LET is used to bind variables to the results of expressions.

```
MATCH (p:Product)
LET isExpensive = p.price >= 500
LET isAffordable = NOT isExpensive
LET discountCategory = CASE
    WHEN isExpensive THEN 'High-end'
    ELSE 'Budget'
END
RETURN p.name AS product, p.price AS
price, isAffordable, discountCategory
ORDER BY price
```

LET can be used to chain expressions.

Write query

Write-Only Query Structure

```
[USE]
[CREATE]
[MERGE [ON CREATE ...] [ON MATCH ...]]
[WITH [ORDER BY] [SKIP] [LIMIT] [WHERE]]
[SET]
[DELETE]
[REMOVE]
[RETURN [ORDER BY] [SKIP] [LIMIT]]
```

Baseline for write operations.

- CREATE clause.
- MERGE clause.
- WITH clause.
- SET clause.
- DELETE clause.
- REMOVE clause.
- RETURN clause.

Read-Write Query Structure

```
[USE]
[MATCH [WHERE]]
[OPTIONAL MATCH [WHERE]]
[WITH [ORDER BY] [SKIP] [LIMIT] [WHERE]]
[CREATE]
[MERGE [ON CREATE ...] [ON MATCH ...]]
[WITH [ORDER BY] [SKIP] [LIMIT] [WHERE]]
[SET]
[DELETE]
[REMOVE]
[RETURN [ORDER BY] [SKIP] [LIMIT]]
```

Baseline for pattern search and write operations.

- USE clause.
- MATCH clause
- OPTIONAL MATCH clause.
- CREATE clause
- MERGE clause.
- WITH clause.
- SET clause.
- DELETE clause.
- REMOVE clause.
- RETURN clause.

CREATE

`CREATE (n:Label {name: $value})`	Create a node with the given label and properties.
`CREATE (n:Label $map)`	Create a node with the given label and properties.
`CREATE (n:Label)-[r:TYPE]->(m:Label)`	Create a relationship with the given relationship type and direction; bind a variable `r` to it.
`CREATE (n:Label)-[:TYPE {name: $value}]->(m:Label)`	Create a relationship with the given type, direction, and properties.

`CREATE (greta:$($nodeLabels) {name: 'Greta Gerwig'})`
`WITH greta`
`UNWIND $movies AS movieTitle`
`CREATE (greta)-[rel:$($relType)]->(m:Movie {title: movieTitle})`
`RETURN greta.name AS name, labels(greta) AS labels, type(rel) AS relType, collect(m.title) AS movies`
	Node labels and relationship types can be referenced dynamically in expressions, parameters, and variables when merging nodes and relationships. The expression must evaluate to a `STRING NOT NULL | LIST<STRING NOT NULL> NOT NULL` value.

SET

```
SET e.property1 = $value1
```

Update or create a property.

```
SET
  e.property1 = $value1,
  e.property2 = $value2
```

Update or create several properties.

```
MATCH (n)
SET n[$key] = value
```

Dynamically set or update node properties.

```
MATCH (n)
SET n:$($label)
```

Dynamically set node labels.

```
SET e = $map
```

Set all properties. This will remove any existing properties.

```
SET e = {}
```

Using the empty map ({}), removes any existing properties.

```
SET e += $map
```

Add and update properties, while keeping existing ones.

```
MATCH (n:Label)
WHERE n.id = 123
SET n:Person
```

Add a label to a node. This example adds the label `Person` to a node.

MERGE

```
MERGE (n:Label {name: $value})
ON CREATE SET n.created = timestamp()
ON MATCH SET
  n.counter = coalesce(n.counter, 0) + 1,
  n.accessTime = timestamp()
```

Match a pattern or create it if it does not exist. Use `ON CREATE` and `ON MATCH` for conditional updates.

```
MATCH
  (a:Person {name: $value1}),
  (b:Person {name: $value2})
MERGE (a)-[r:LOVES]->(b)
```

`MERGE` finds or creates a relationship between the nodes.

```
MATCH (a:Person {name: $value1})
```

`MERGE` finds or creates paths attached to the node.

```
MERGE (greta:$($nodeLabels) {name: 'Greta
Gerwig'})
WITH greta
UNWIND $movies AS movieTitle
MERGE (greta)-[rel:$($relType)]->(m:Movie
{title: movieTitle})
RETURN greta.name AS name, labels(greta)
AS labels, type(rel) AS relType,
collect(m.title) AS movies
```

Node labels and relationship types can be referenced dynamically in expressions, parameters, and variables when merging nodes and relationships. The expression must evaluate to a `STRING NOT NULL | LIST<STRING NOT NULL> NOT NULL` value.

DELETE

```
MATCH (n:Label)-[r]->(m:Label)
WHERE r.id = 123
DELETE r
```

Delete a relationship.

```
MATCH ()-[r]->()
DELETE r
```

Delete all relationships.

```
MATCH (n:Label)
WHERE n.id = 123
DETACH DELETE n
```

Delete a node and all relationships connected to it.

```
MATCH (n:Label)-[r]-()
WHERE r.id = 123 AND n.id = 'abc'
DELETE n, r
```

Delete a node and a relationship. An error will be thrown if the given node is attached to more than one relationship.

```
MATCH (n1:Label)-[r {id: 123}]->
(n2:Label)
CALL (n1) {
  MATCH (n1)-[r1]-()
  RETURN count(r1) AS rels1
}
CALL (n2) {
  MATCH (n2)-[r2]-()
  RETURN count(r2) AS rels2
}
DELETE r
RETURN
  n1.name AS node1, rels1 - 1 AS
relationships1,
  n2.name AS node2, rels2 - 1 AS
relationships2
```

Delete a relationship and return the number of relationships for each node after the deletion. This example uses a variable scope clause (introduced in Neo4j 5.23) to import variables into the `CALL` subquery. If you are using an older version of Neo4j, use an importing `WITH` clause instead.

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

Delete all nodes and relationships from the database.

REMOVE

```
MATCH (n:Label)
WHERE n.id = 123
REMOVE n:Label
```

Remove a label from a node.

```
MATCH (n {name: 'Peter'})
REMOVE n:$($label)
RETURN n.name
```

Dynamically remove node labels.

```
MATCH (n:Label)
WHERE n.id = 123
REMOVE n.alias
```

Remove a property from a node.

```
MATCH (n)
REMOVE n[$key]
```

Dynamically remove properties from nodes.

```
MATCH (n:Label)
WHERE n.id = 123
SET n = {} # REMOVE ALL properties
```

`REMOVE` cannot be used to remove all existing properties from a node or relationship. All existing properties can be removed from a node or relationship by using the `SET` clause with the property replacement operator (=) and an empty map ({}) as the right operand.

Cypher query versioning

Select Cypher version for queries

```
CYPHER 25
MATCH (n:Order)-[r:SHIPPED_TO]->
(:Address)
SET n = properties(r)
```

Prepending a query with `CYPHER 25` ensures that the query will be executed using Cypher 25 as it exists in the version of Neo4j that the database is currently running, provided it is on Neo4j 2025.06 or later.

```
CYPHER 5
MATCH (n:Order)-[r:SHIPPED_TO]->
(:Address)
SET n = r
```

Selecting `CYPHER 5` ensures that the query will be executed using the Cypher 5 as it existed at the time of the Neo4j 2025.06 release. Any changes introduced after the 2025.06 release will not affect the query.

Composed queries

Combined queries (UNION)

```
MATCH (n:Actor)
RETURN n.name AS name
UNION
MATCH (n:Movie)
RETURN n.title AS name
```

Return the distinct union of all query results. Result column types and names must match.

```
MATCH (n:Actor)
RETURN n.name AS name
UNION ALL
MATCH (n:Movie)
RETURN n.title AS name
```

Return the union of all query results, including duplicate rows.

```
CALL () {
  MATCH (a:Actor)
  RETURN a.name AS name
UNION ALL
  MATCH (m:Movie)
  RETURN m.title AS name
}
RETURN name, count(*) AS count
ORDER BY count
```

The `UNION` clause can be used within a `CALL` subquery to further process the combined results before a final output is returned.

```
{
   MATCH (n:Actor)
   RETURN n.name AS name
   UNION
   MATCH (n:Director)
   RETURN n.name AS name
}
UNION ALL
MATCH (n:Movie)
RETURN n.title AS name
```

To combine `UNION` (or `UNION DISTINCT`) and `UNION ALL` in the same query, enclose one or more `UNION` operations of the same type in curly braces. This allows the enclosed query to act as an argument that can be combined with an outer `UNION` operation of any type.

Conditional queries (WHEN)

```
WHEN false THEN RETURN 1 AS x
WHEN true THEN RETURN 2 AS x
WHEN true THEN RETURN 3 AS x
ELSE RETURN 3 AS x
```

`WHEN`, together with `THEN` and `ELSE`, enables different branches of a query to execute based on certain conditions. The first branch with a predicate that evaluates to `true` will be executed. If no `WHEN` branches are executed and an `ELSE` branch exists, it is executed. If no `WHEN` branches evaluates to `true` and no `ELSE` branch is present, no branches are executed and no rows are produced.

```
WHEN true THEN {
  MATCH (n:Person) WHERE n.name STARTS
WITH "A"
  RETURN n.name AS name
}
ELSE {
  MATCH (n:Person)
  RETURN n.name AS name
}
```

Queries can be executed conditionally executed in standalone `WHEN` branches.

```
MATCH (n:Person)
OPTIONAL MATCH (n)-[:WORKS_FOR]->
(manager:Person)
CALL (*) {
  WHEN manager IS NULL THEN {
    MERGE (newManager: Person {name:
'Peter', age: 36})
    MERGE (n)-[:WORKS_FOR]->(newManager)
    RETURN newManager, n.name AS employee
  }
}
RETURN newManager.name AS newManager,
       collect(employee) AS employees
```

`WHEN` can be used inside one or several `CALL` subqueries to execute a set of operations only when a specified condition evaluates to `true`.

```
MATCH (n:Person)
WHERE EXISTS {
  WHEN n.age > 40 THEN {
    RETURN n.name AS x
```

`EXISTS`, `COLLECT`, and `COUNT` subquery expressions can also contain `WHEN` branches.

```
  }
  ELSE {
    MATCH (n)-[:LOVES]->(x:Person)
    RETURN x
  }
}
RETURN n.name AS name,
       n.age AS age
```

```
{
  WHEN true THEN RETURN 1 AS x
  WHEN false THEN RETURN 2 AS x
  ELSE RETURN 3 AS x
}
UNION
{
  WHEN false THEN RETURN 4 AS x
  WHEN false THEN RETURN 5 AS x
  ELSE RETURN 6 AS x
}
```

The results of multiple conditional queries can also be combined using `UNION [DISTINCT]` or `UNION ALL`. If the conditional query begins with `WHEN` and involves `UNION`, the `WHEN` branches **must** be enclosed within curly braces, `{}`.

Sequential queries (NEXT)

```
MATCH (c:Customer)-[:BUYS]->(p:Product
{name: 'Chocolate'})
RETURN c AS customer, p AS product

NEXT

RETURN customer.firstName AS
chocolateCustomer,
       product.price * (1 -
customer.discount) AS chocolatePrice
```

`NEXT` allows for linear composition of queries into a sequence of smaller, self-contained segments, passing the return values from one segment to the next.

```
MATCH (p:Product)
RETURN p

NEXT

MATCH (c:Customer)-[:BUYS]->(p)
RETURN collect(c.firstName) AS customers,
p

NEXT

RETURN p.name as product, customers
```

`NEXT` can serve as a more readable alternative to `CALL` subqueries. In this example, `NEXT` divides the query into three concise queries that avoids the indentation and parentheses of a `CALL` subquery.

```
MATCH (c:Customer)-[:BUYS]->(:Product)<-
[:SUPPLIES]-(s:Supplier)
RETURN c.firstName AS customer, s.name AS
supplier

NEXT

WHEN supplier = "TechCorp" THEN
  RETURN customer, "Tech enjoyer" AS
personality
WHEN supplier = "Foodies Inc." THEN
  RETURN customer, "Tropical plant
enjoyer" AS
        View all (9 more lines)

NEXT
```

`NEXT` can be used to chain conditional `WHEN` constructs.

```
MATCH (c:Customer)-[:BUYS]->(p:Product)
RETURN c AS customer, sum(p.price) AS sum

NEXT
```

If a conditional query has a `NEXT` in any of its `THEN` or `ELSE` blocks, it is necessary to wrap the part after `THEN` or `ELSE` with `{}`.

```
WHEN sum >= 1000 THEN {
  RETURN customer.firstName AS customer,
"club 1000 plus" AS customerType, sum AS
sum
}
ELSE {
  RETURN customer AS customer, sum * (1 -
customer
```

View all (3 more lines)

Patterns

Fixed-length patterns

```
MATCH (n:Station WHERE n.name STARTS WITH
'Preston')
RETURN n
```

Match a node pattern including a `WHERE` clause predicate.

```
MATCH (s:Stop)-[:CALLS_AT]->(:Station
{name: 'Denmark Hill'})
RETURN s.departs AS departureTime
```

Match a fixed-length path pattern to paths in a graph.

Variable-length patterns

```
MATCH (:Station { name: 'Denmark Hill' })
<-[:CALLS_AT]-(d:Stop)
      ((:Stop)-[:NEXT]->(:Stop)){1,3}
      (a:Stop)-[:CALLS_AT]->(:Station {
name: 'Clapham Junction' })
RETURN d.departs AS departureTime,
a.arrives AS arrivalTime
```

Quantified path pattern matching a sequence of paths whose length is constrained to a specific range (1 to 3 in this case) between two nodes.

```
MATCH (d:Station { name: 'Denmark Hill'
})<-[:CALLS_AT]-
        (n:Stop)-[:NEXT]->{1,10}(m:Stop)-
[:CALLS_AT]->
        (a:Station { name: 'Clapham
Junction' })
WHERE m.arrives < time('17:18')
RETURN n.departs AS departureTime
```

Quantified relationship matching paths where a specified relationship occurs between 1 and 10 times.

```
MATCH (bfr:Station {name: "London
Blackfriars"}),
      (ndl:Station {name: "North
Dulwich"})
MATCH p = (bfr)
          ((a)-[:LINK]-(b:Station)
            WHERE
point.distance(a.location, ndl.location)
>
              point.distance(b.location,
ndl.location))+ (ndl)
RETURN reduce(acc = 0, r in
relationships(p) | round(acc +
r.distance, 2))
  AS distance
```

Quantified path pattern including a predicate.

Shortest paths

```
MATCH p = SHORTEST 1 (wos:Station)-
[:LINK]-+(bmv:Station)
WHERE wos.name = "Worcester Shrub Hill"
AND bmv.name = "Bromsgrove"
RETURN length(p) AS result
```

`SHORTEST k` finds the shortest path(s) (by number of hops) between nodes, where `k` is the number of paths to match.

```
MATCH p = ALL SHORTEST (wos:Station)-
[:LINK]-+(bmv:Station)
WHERE wos.name = "Worcester Shrub Hill"
AND bmv.name = "Bromsgrove"
RETURN [n in nodes(p) | n.name] AS stops
```

Find all shortest paths between two nodes.

```
MATCH p = SHORTEST 2 GROUPS
(wos:Station)-[:LINK]-+(bmv:Station)
WHERE wos.name = "Worcester Shrub Hill"
AND bmv.name = "Bromsgrove"
RETURN [n in nodes(p) | n.name] AS stops,
length(p) AS pathLength
```

`SHORTEST k GROUPS` returns all paths that are tied for first, second, and so on, up to the kth shortest length. This example finds all paths with the first and second shortest lengths between two nodes.

```
MATCH path = ANY
  (:Station {name: 'Pershore'})-[l:LINK
WHERE l.distance < 10]-+(b:Station {name:
'Bromsgrove'})
RETURN [r IN relationships(path) |
r.distance] AS distances
```

The `ANY` keyword can be used to test the reachability of nodes from a given node(s). It returns the same as `SHORTEST 1`, but by using the `ANY` keyword the intent of the query is clearer.

Non-linear patterns

```
MATCH (n:Station {name: 'London Euston'})
<-[:CALLS_AT]-(s1:Stop)
  -[:NEXT]->(s2:Stop)-[:CALLS_AT]->
(:Station {name: 'Coventry'})
  <-[:CALLS_AT]-(s3:Stop)-[:NEXT]->
(s4:Stop)-[:CALLS_AT]->(n)
RETURN s1.departs+'-'+s2.departs AS
outbound,
  s3.departs+'-'+s4.departs AS `return`
```

An equijoin is an operation on paths that requires more than one of the nodes or relationships of the paths to be the same. The equality between the nodes or relationships is specified by declaring a node variable or relationship variable more than once. An equijoin on nodes allows cycles to be specified in a path pattern. Due to relationship uniqueness, an equijoin on relationships yields no solutions.

```
MATCH (:Station {name: 'Starbeck'})<-
[:CALLS_AT]-
        (a:Stop {departs:
time('11:11')})-[:NEXT]->*(b)-[:NEXT]->*
        (c:Stop)-[:CALLS_AT]->
(lds:Station {name: 'Leeds'}),
      (b)-[:CALLS_AT]->(l:Station)<-
[:CALLS_AT]-(m:Stop)-[:NEXT]->*
        (n:Stop)-[:CALLS_AT]->(lds),
      (lds)<-[:CALLS_AT]-(x:Stop)-
[:NEXT]->*(y:Stop)-[:CALLS_AT]->
        (:Station {name: 'Huddersfield'})
WHERE b.arrives < m.departs AND n.arrives
< x.departs
RETURN a.departs AS departs,
       l.name AS changeAt,
       m.departs AS changeDeparts,
       y.arrives AS arrives
ORDER BY y.arrives LIMIT 1
```

Multiple path patterns can be combined in a comma-separated list to form a graph pattern. In a graph pattern, each path pattern is matched separately, and where node variables are repeated in the separate path patterns, the solutions are reduced via equijoins.

Match modes

```
MATCH p = (:Location {name: 'Kneiphof'})-
-{7}()
RETURN count(p) AS pathCount
```

Cypher's default match mode is `DIFFERENT RELATIONSHIPS`. This is a restrictive match mode, which requires that all relationships matched across all constituent path patterns in a graph pattern must be unique. The match mode can be explicitly defined by adding `DIFFERENT RELATIONSHIPS` after `MATCH`. Explicitly defining `DIFFERENT RELATIONSHIPS` is functionally equivalent to not specifying a match mode.

```
MATCH REPEATABLE ELEMENTS p = (:Location
{name: 'Kneiphof'})-[:BRIDGE]-{7}()
WITH collect(p)[0] AS samplePath
RETURN [n IN nodes(samplePath) | n.name]
AS samplePathLocations,
       [r IN relationships(samplePath) |
r.id] AS samplePathBridges
```

`REPEATABLE ELEMENTS` is a non-restrictive match mode, in which relationships matched across all constituent path patterns in a graph pattern can be repeatedly traversed. Queries using `REPEATABLE ELEMENTS` must specify an upper bound to a pattern to ensure that a finite number of solutions are returned in a finite amount of time (i.e. quantifiers such as `*`, `+`, or `{1,}` are not allowed using `REPEATABLE ELEMENTS`).

Clauses

CALL procedure

```
CALL db.labels() YIELD label
```

Standalone call to the procedure `db.labels` to list all labels used in the database. Note that required procedure arguments are given explicitly in brackets after the procedure name.

```
MATCH (n)
OPTIONAL CALL apoc.neighbors.tohop(n,
"KNOWS>", 1)
YIELD node
RETURN n.name AS name, collect(node.name)
AS connections
```

Optionally `CALL` a procedure. Similar to `OPTIONAL MATCH`, any empty rows produced by the `OPTIONAL CALL` will return `null` and not affect the remainder of the procedure evaluation.

```
CALL db.labels() YIELD *
```

Standalone calls may use `YIELD *` to return all columns.

```
CALL java.stored.procedureWithArgs
```

Standalone calls may omit `YIELD` and also provide arguments implicitly via statement parameters, e.g. a standalone call requiring one argument input may be run by passing the parameter map `{input: 'foo'}`.

```
CALL db.labels() YIELD label
RETURN count(label) AS db_labels
```

Calls the built-in procedure `db.labels` inside a larger query to count all labels used in the database. Calls inside a larger query always requires passing arguments and naming results explicitly with `YIELD`.

FINISH

```
MATCH (p:Person)
FINISH
```

A query ending in `FINISH` — instead of `RETURN` — has no result but executes all its side effects.

FOREACH

```
MATCH p=(start)-[*]->(finish)
WHERE start.name = 'A' AND finish.name =
'D'
FOREACH (n IN nodes(p) | SET n.marked =
true)
```

`FOREACH` can be used to update data, such as executing update commands on elements in a path, or on a list created by aggregation. This example sets the property `marked` to `true` on all nodes along a path.

```
MATCH p=(start)-[*]->(finish)
WHERE start.name = 'A' AND finish.name =
'D'
FOREACH ( r IN relationships(p) | SET
r.marked = true )
```

This example sets the property `marked` to `true` on all relationships along a path.

```
WITH ['E', 'F', 'G'] AS names
FOREACH ( value IN names | CREATE
(:Person {name: value}) )
```

This example creates a new node for each label in a list.

LIMIT

```
MATCH (n)
ORDER BY n.name DESC
SKIP 2
LIMIT 2
RETURN collect(n.name) AS names
```

`LIMIT` constrains the number of returned rows. It can be used in conjunction with `ORDER BY` and `SKIP`.

```
MATCH (n)
LIMIT 2
RETURN collect(n.name) AS names
```

`LIMIT` can be used as a standalone clause.

LOAD CSV

```
LOAD CSV FROM 'file:///artists.csv' AS
row
MERGE (a:Artist {name: row[1], year:
toInteger(row[2])})
RETURN a.name, a.year
```

`LOAD CSV` is used to import data from CSV files into a Neo4j database. This example imports the name and year information of artists from a local file.

```
LOAD CSV FROM
'https://data.neo4j.com/bands/artists.csv'
AS row
MERGE (a:Artist {name: row[1], year:
toInteger(row[2])})
RETURN a.name, a.year
```

Import artists name and year information from a remote file URL.

```
LOAD CSV WITH HEADERS FROM
'file:///bands-with-headers.csv' AS line
MERGE (n:$(line.Label) {name: line.Name})
RETURN n AS bandNodes
```

CSV columns can be referenced dynamically to map labels to nodes in the graph. This enables flexible data handling, allowing labels to be be populated from CSV column values without manually specifying each entry.

```
LOAD CSV WITH HEADERS FROM
'https://data.neo4j.com/importing-
cypher/persons.csv' AS row
CALL (row) {
  MERGE (p:Person {tmdbId:
row.person_tmdbId})
  SET p.name = row.name, p.born =
row.born
} IN TRANSACTIONS OF 200 ROWS
```

Load a CSV file in several transactions. This example uses a variable scope clause (introduced in Neo4j 5.23) to import variables into the `CALL` subquery.

```
LOAD CSV FROM 'file:///artists.csv' AS
row
RETURN linenumber() AS number, row
```

Access line numbers in a CSV with the `linenumber()` function.

```
LOAD CSV FROM 'file:///artists.csv' AS
row
RETURN DISTINCT file() AS path
```

Access the CSV file path with the `file()` function.

```
LOAD CSV WITH HEADERS FROM
'file:///artists-with-headers.csv' AS row
MERGE (a:Artist {name: row.Name, year:
toInteger(row.Year)})
RETURN
  a.name AS name,
  a.year AS year
```

Load CSV data with headers.

```
LOAD CSV FROM 'file:///artists-
fieldterminator.csv' AS row
FIELDTERMINATOR ';'
MERGE (:Artist {name: row[1], year:
toInteger(row[2])})
```

Import a CSV using `;` as field delimiter.

ORDER BY

```
MATCH (o:Order)
RETURN o.id AS order,
       o.total AS total
  ORDER BY total
```

`ORDER BY` specifies how the output of a clause should be sorted. It can be used as a sub-clause following `RETURN` or `WITH`.

```
MATCH (o:Order)
RETURN o.id AS order,
       o.total AS total,
       o.orderDate AS orderDate
  ORDER BY total,
           orderDate
```

You can order by multiple properties by stating each variable in the `ORDER BY` clause.

```
MATCH (i:Item)
ORDER BY i.price DESC
SKIP 1
LIMIT 1
RETURN i.name AS secondMostExpensiveItem,
       i.price AS price
```

By adding `DESC[ENDING]` after the variable to sort on, the sort will be done in reverse order.

`ORDER BY` can be used in conjunction with `SKIP` and `LIMIT`.

```
MATCH (i:Item)
ORDER BY i.price
RETURN collect(i.name || " ($" ||
toString(i.price) || ")") AS
orderedPriceList
```

`ORDER BY` can be used as a standalone clause.

SHOW FUNCTIONS

`SHOW FUNCTIONS`	List all available functions, returns only the default outputs (`name` , `category` , and `description`).
`SHOW BUILT IN FUNCTIONS YIELD *`	List built-in functions, can also be filtered on `ALL` or `USER-DEFINED` .
`SHOW FUNCTIONS EXECUTABLE BY CURRENT USER YIELD *`	Filter the available functions for the current user.
`SHOW FUNCTIONS EXECUTABLE BY user_name`	Filter the available functions for the specified user.

SHOW PROCEDURES

`SHOW PROCEDURES`	List all available procedures, returns only the default outputs (`name` , `description` , `mode` , and `worksOnSystem`).
`SHOW PROCEDURES YIELD *`	List all available procedures.
`SHOW PROCEDURES EXECUTABLE YIELD name`	List all procedures that can be executed by the current user and return only the name of the procedures.

SHOW SETTINGS

Neo4j Community Edition Neo4j Enterprise Edition

`SHOW SETTINGS`	List configuration settings (within the instance), returns only the default outputs (`name` , `value` , `isDynamic` , `defaultValue` , and `description`).
`SHOW SETTINGS YIELD *`	List configuration settings (within the instance).
`SHOW SETTINGS 'server.bolt.advertised_address', 'server.bolt.listen_address' YIELD *`	List the configuration settings (within the instance) named `server.bolt.advertised_address` and `server.bolt.listen_address` . As long as the setting names evaluate to a string or a list of strings at runtime, they can be any expression.

SHOW TRANSACTIONS

`SHOW TRANSACTIONS`	List running transactions (within the instance), returns only the default outputs (`database` , `transactionId` , `currentQueryId` , `connectionId` , `clientAddress` , `username` , `currentQuery` , `startTime` , `status` , and `elapsedTime`).
`SHOW TRANSACTIONS YIELD *`	List running transactions (within the instance).
`SHOW TRANSACTIONS 'transaction_id' YIELD *`	List the running transaction (within the instance), with a specific `transaction_id` . As long as the transaction IDs evaluate to a string or a list of strings at runtime, they can be any expression.

SKIP

```
MATCH (n)
RETURN n.name
ORDER BY n.name
SKIP 1
LIMIT 2
```

`SKIP` defines from which row to start including the rows in the output. It can be used in conjunction with `LIMIT` and `ORDER BY`.

```
MATCH (n)
SKIP 2
RETURN collect(n.name) AS names
```

`SKIP` can be used as a standalone clause.

```
MATCH (n)
ORDER BY n.name
OFFSET 2
LIMIT 2
RETURN collect(n.name) AS names
```

`OFFSET` can be used as a synonym to `SKIP`.

TERMINATE TRANSACTIONS

```
TERMINATE TRANSACTIONS 'transaction_id'
```

Terminate a specific transaction, returns the outputs: `transactionId`, `username`, `message`.

```
TERMINATE TRANSACTIONS $value
  YIELD transactionId, message
  RETURN transactionId, message
```

Terminal transactions allow for `YIELD` clauses. As long as the transaction IDs evaluate to a string or a list of strings at runtime, they can be any expression.

```
 SHOW TRANSACTIONS
  YIELD transactionId AS txId, username
  WHERE username = 'user_name'
TERMINATE TRANSACTIONS txId
  YIELD message
  WHERE NOT message = 'Transaction
terminated.'
  RETURN txId
```

List all transactions by the specified user and terminate them. Return the transaction IDs of the transactions that failed to terminate successfully.

UNWIND

```
UNWIND [1, 2, 3, null] AS x
RETURN x, 'val' AS y
```

The `UNWIND` clause expands a list into a sequence of rows.

Four rows are returned.

```
UNWIND $events AS event
MERGE (y:Year {year: event.year})
MERGE (y)<-[:IN]-(e:Event {id: event.id})
RETURN e.id AS x ORDER BY x
```

Multiple `UNWIND` clauses can be chained to unwind nested list elements.

Five rows are returned.

```
UNWIND [1, 2, 3, null] AS x
RETURN x, 'val' AS y
```

Create a number of nodes and relationships from a parameter-list without using `FOREACH`.

USE

```
USE myDatabase
MATCH (n) RETURN n
```

The `USE` clause determines which graph a query is executed against. This example assumes that the DBMS contains a database named `myDatabase`.

```
USE myComposite.myConstituent
MATCH (n) RETURN n
```

This example assumes that the DBMS contains a composite database named `myComposite`, which includes an alias named `myConstituent`.

Subqueries

CALL

```
UNWIND [0, 1, 2] AS x
CALL () {
  RETURN 'hello' AS innerReturn
}
RETURN innerReturn
```

A `CALL` subquery is executed once for each row. In this example, the `CALL` subquery executes three times.

```
MATCH (t:Team)
CALL (t) {
  MATCH (p:Player)-[:PLAYS_FOR]->(t)
  RETURN collect(p) as players
}
RETURN t AS team, players
```

Variables are imported into a `CALL` subquery using a variable scope clause, `CALL (<variable>)`, or an importing `WITH` clause (deprecated). In this example, the subquery will process each `Team` at a time and `collect` a list of all `Player` nodes.

```
MATCH (p:Player)
OPTIONAL CALL (p) {
    MATCH (p)-[:PLAYS_FOR]->(team:Team)
    RETURN team
}
RETURN p.name AS playerName, team.name AS
team
```

Optionally `CALL` a subquery. Similar to OPTIONAL MATCH, any empty rows produced by the `OPTIONAL CALL` will return `null` and not affect the remainder of the subquery evaluation.

`CALL` subqueries can be used to further process the results of a `UNION` query. This example finds the youngest and the oldest `Player` in the graph.

CALL subqueries in transactions

```
LOAD CSV FROM 'file:///friends.csv' AS
line
CALL (line) {
  CREATE (:Person {name: line[1], age:
toInteger(line[2])})
} IN TRANSACTIONS
```

`CALL` subqueries can execute in separate, inner transactions, producing intermediate commits.

```
LOAD CSV FROM 'file:///friends.csv' AS
line
CALL (line) {
  CREATE (:Person {name: line[1], age:
toInteger(line[2])})
} IN TRANSACTIONS OF 2 ROWS
```

Specify the number of rows processed in each transaction.

```
UNWIND [1, 0, 2, 4] AS i
CALL (i) {
  CREATE (n:Person {num: 100/i}) // Note,
fails when i = 0
  RETURN n
} IN TRANSACTIONS
  OF 1 ROW
  ON ERROR CONTINUE
RETURN n.num
```

There are four different option flags to control the behavior in case of an error occurring in any of the inner transactions:

- `ON ERROR CONTINUE` -ignores a recoverable error and continues the execution of subsequent inner transactions. The outer transaction succeeds.
- `ON ERROR BREAK` -ignores a recoverable error and stops the execution of subsequent inner transactions. The outer transaction succeeds.
- `ON ERROR FAIL` -acknowledges a recoverable error and stops the execution of subsequent inner transactions. The outer transaction fails.
- `ON ERROR RETRY` -uses an exponential delay between retry attempts for transaction batches that fail due to transient errors (i.e. errors where retrying a transaction can be expected to give a different result), with an optional maximum retry duration. If the transaction still fails after the maximum duration, the failure is handled according to an optionally specified fallback error handling mode (`THEN`

`CONTINUE`, `THEN BREAK`, `THEN FAIL` (default)).

```
LOAD CSV WITH HEADERS FROM
'https://data.neo4j.com/importing-
cypher/persons.csv' AS row
CALL (row) {
  CREATE (p:Person {tmdbId:
row.person_tmdbId})
  SET p.name = row.name, p.born =
row.born
} IN 3 CONCURRENT TRANSACTIONS OF 10 ROWS
RETURN count(*) AS personNodes
```

`CALL` subqueries can execute batches in parallel by appending `IN [n] CONCURRENT TRANSACTIONS`, where `n` is an optional concurrency value used to set the maximum number of transactions that can be executed in parallel.

```
LOAD CSV WITH HEADERS FROM
'https://data.neo4j.com/importing-
cypher/movies.csv' AS row
CALL (row) {
   MERGE (m:Movie {movieId: row.movieId})
   MERGE (y:Year {year: row.year})
   MERGE (m)-[r:RELEASED_IN]->(y)
} IN 2 CONCURRENT TRANSACTIONS OF 10 ROWS
ON ERROR RETRY FOR 3 SECONDS THEN
CONTINUE REPORT STATUS AS status
RETURN status.transactionId as
transaction, status.committed AS
successfulTransaction
```

`ON ERROR RETRY … THEN CONTINUE` can be used to retry the execution of a transaction for a specified maximum duration before continuing the execution of subsequent inner transactions by ignoring any recoverable errors.

COUNT, COLLECT, and EXISTS

```
MATCH (person:Person)
WHERE COUNT { (person)-[:HAS_DOG]->(:Dog)
} > 1
RETURN person.name AS name
```

A `COUNT` subquery counts the number of rows returned by the subquery. Unlike `CALL` subqueries, variables introduced by the outer scope can be used in `EXISTS`, `COLLECT`, and `COUNT` subqueries.

```
MATCH (person:Person)
WHERE EXISTS {
  MATCH (person)-[:HAS_DOG]->(dog:Dog)
  WHERE person.name = dog.name
}
RETURN person.name AS name
```

An `EXISTS` subquery determines if a specified pattern exists at least once in the graph. A `WHERE` clause can be used inside `COLLECT`, `COUNT`, and `EXISTS` patterns.

```
MATCH (person:Person) WHERE person.name =
"Peter"
SET person.dogNames = COLLECT { MATCH
(person)-[:HAS_DOG]->(d:Dog) RETURN
d.name }
RETURN person.dogNames as dogNames
```

A `COLLECT` subquery creates a list with the rows returned by the subquery. `COLLECT`, `COUNT`, and `EXISTS` subqueries can be used inside other clauses.

Predicates

Boolean operators

```
MATCH (n:Person)
WHERE n.age > 30 AND n.role = 'Software
developer'
RETURN n.name AS name, n.age AS age,
n.role AS role
```

The `AND` operator is used to combine multiple boolean expressions, returning `true` only if all conditions are true.

```
MATCH (n:Person)
WHERE n.age < 30 OR n.role = 'Software
developer'
RETURN n.name AS name, n.age AS age,
n.role AS role
```

The `OR` operator is used to combine multiple boolean expressions, returning `true` if at least one of the conditions is true.

```
MATCH (n:Person)
WHERE n.age > 30 XOR n.role = 'Software
developer'
RETURN n.name AS name, n.age AS age,
n.role AS role
```

The `XOR` operator returns `true` if exactly one of the two boolean expressions is true, but not both.

```
MATCH (n:Person)
WHERE NOT n.age = 39
RETURN n.name AS name, n.age AS age
```

The `NOT` operator negates a boolean expression, returning `true` if the expression is false and `false` if it is true.

Comparison operators

```
MATCH (n:Person)
WHERE n.role = 'Software developer'
RETURN n.name AS name, n.role AS role
```

The equality operator `=` checks for equality between two values.

```
MATCH (n:Person)
WHERE n.role <> 'Software developer'
RETURN n.name AS name, n.role AS role
```

The inequality operator `<>` checks if two values are not equal.

```
MATCH (n:Person)
WHERE n.age < 39
RETURN n.name AS name, n.age AS age
```

The less than operator `<` returns `true` if the value on the left is less than the value on the right.

```
MATCH (n:Person)
WHERE n.age <= 39
RETURN n.name AS name, n.age AS age
```

The less than or equal or operator `<=` returns `true` if the value on the left is less than or equal to the value on the right.

```
MATCH (n:Person)
WHERE n.age > 39
RETURN n.name AS name, n.age AS age
```

The greater than operator `>` returns `true` if the value on the left is greater than the value on the right.

```
MATCH (n:Person)
WHERE n.age >= 39
RETURN n.name AS name, n.age AS age
```

The greater than or equal operator `>=` returns `true` if the value on the left is greater than the value on the right.

```
MATCH (n:Person)
WHERE n.email IS NULL
RETURN n.name AS name
```

The `IS NULL` operator returns `true` if the value is `NULL`, and `false` otherwise.

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

The `IS NOT NULL` operator returns `true` if the value is not `NULL`, and `false` otherwise.

List operators

```
MATCH (n:Person)
WHERE n.role IN ['Software developer',
'Project manager']
RETURN n.name AS name, n.role AS role
```

The `IN` operator checks if a value is present in a `LIST`.

```
RETURN any(x IN [1, 2, null] WHERE x IS
NULL) AS containsNull
```

To check if `NULL` is a member of a `LIST`, use the `any()` function.

```
RETURN [3, 4] IN [[1, 2], [3, 4]] AS
listInNestedList
```

When used with nested `LIST` values, the `IN` operator evaluates whether a `LIST` is an exact match to any of the nested `LIST` values that are part of an outer `LIST`. Partial matches of individual elements within a nested `LIST` will return `false`.

```
WITH [1,3,4] AS sub, [3,5,1,7,6,2,8,4] AS
list
RETURN all(x IN sub WHERE x IN list) AS
subInList
```

A subset check using the `all()` function verifies if all elements of one `LIST` exist in another.

Path pattern expressions

```
MATCH (employee:Person)
WHERE (employee)-[:WORKS_FOR]->(:Person
{name: 'Alice'})
RETURN employee.name AS employee
```

Similar to EXISTS subqueries, path pattern expressions can be used to assert whether a specified path exists at least once in a graph.

```
MATCH (employee:Person)
WHERE NOT employee.name = 'Cecil' AND
(employee)-[:WORKS_FOR]->(:Person {name:
'Alice'})
RETURN employee.name AS employee
```

Path pattern expression with boolean operators.

String operators

```
MATCH (n:Person)
WHERE n.name STARTS WITH 'C'
RETURN n.name AS name
```

The `STARTS WITH` operator checks if a `STRING` value begins with a specified prefix.

```
MATCH (n:Person)
WHERE n.role ENDS WITH 'developer'
RETURN n.name AS name, n.role AS role
```

The `ENDS WITH` operator checks if a `STRING` value ends with a specified suffix

```
MATCH (n:Person)
WHERE n.role CONTAINS 'eng'
RETURN n.name AS name, n.role AS role
```

The `CONTAINS` operator checks if a `STRING` value contains a specified substring.

```
MATCH (n:Person)
WHERE n.email =~ '.*@company.com'
RETURN n.name AS name, n.email AS email
```

The regular expression operator `=~` checks if a `STRING`value matches a regular expression.

```
MATCH (n:Person)
WHERE n.name =~ '(?i)CEC.*'
RETURN n.name
```

The `=~` operator can be used with regular expression flags, such as `(?i)` for case-insensitive matching, to modify how the regex is applied.

```
RETURN 'the \u212B char' IS NORMALIZED AS
normalized
```

The `IS NORMALIZED` operator is used to check whether the given `STRING` value is in the `NFC` Unicode normalization form.

```
RETURN 'the \u212B char' IS NOT
NORMALIZED AS notNormalized
```

The `IS NOT NORMALIZED` operator is used to check whether the given `STRING` value is not in the `NFC` Unicode normalization form.

```
WITH 'the \u00E4 char' as myString
RETURN myString IS NFC NORMALIZED AS
```

It is possible to define which Unicode normalization type is used. The available

```
nfcNormalized,
    myString IS NFD NORMALIZED AS
nfdNormalized
```

normalization types are: `NFC` (default), `NFD`, `NFKC`, and `NFKD`.

Type predicate expressions

```
UNWIND [42, true, 'abc', null] AS val
RETURN val, val IS :: INTEGER AS
isInteger
```

A type predicate expression can be used to verify the type of a variable, literal, property or other Cypher expression.

```
UNWIND [42, true, 'abc', null] AS val
RETURN val, val IS NOT :: STRING AS
notString
```

It is possible to verify that a Cypher expression is not of a certain type, using the negated type predicate expression `IS NOT ::`.

```
RETURN
  NULL IS :: BOOLEAN AS isBoolean,
  NULL IS :: BOOLEAN NOT NULL AS
isNotNullBoolean
```

All Cypher types includes the `NULL` value. Type predicate expressions can be appended with `NOT NULL`. This means that `IS ::` returns `TRUE` for all expressions evaluating to `NULL`, unless `NOT NULL` is appended.

```
MATCH (n:Person)
WHERE n.age IS :: INTEGER AND n.age > 18
RETURN n.name AS name, n.age AS age
```

Type predicate expressions can also be used to filter out nodes or relationships with properties of a certain type.

```
UNWIND [42, 42.0, "42"] as val
RETURN val, val IS :: INTEGER | FLOAT AS
isNumber
```

Closed dynamic union types allow for the testing of multiple types in the same predicate.

Expressions

Conditional expressions(CASE)

```
MATCH (n:Person)
RETURN
CASE n.eyes
  WHEN 'blue'  THEN 1
  WHEN 'brown', 'hazel' THEN 2
  ELSE 3
END AS result, n.eyes
```

The simple `CASE` form is used to compare a single expression against multiple values, and is analogous to the `switch` construct of programming languages. The expressions are evaluated by the `WHEN` operator until a match is found. If no match is found, the expression in the `ELSE` operator is returned. If there is no `ELSE` case and no match is found, `null` will be returned.

```
MATCH (n:Person)
RETURN n.name,
CASE n.age
  WHEN IS NULL, IS NOT TYPED INTEGER |
FLOAT THEN "Unknown"
  WHEN = 0, = 1, = 2 THEN "Baby"
  WHEN <= 13 THEN "Child"
  WHEN < 20 THEN "Teenager"
  WHEN < 30 THEN "Young Adult"
  WHEN > 1000 THEN "Immortal"
  ELSE "Adult"
END AS result
```

The extended simple `CASE` can use comparison operators.

```
MATCH (n:Person)
RETURN
CASE
  WHEN n.eyes = 'blue' THEN 1
  WHEN n.age < 40      THEN 2
  ELSE 3
END AS result, n.eyes, n.age
```

The generic `CASE` expression supports multiple conditional statements, and is analogous to the `if-elseif-else` construct of programming languages. Each row is evaluated in order until a `true` value is found. If no match is found, the expression in the `ELSE` operator is returned. If there is no `ELSE` case and no match is found, `null` will be returned.

```
MATCH (n:Person)
WITH n,
CASE n.eyes
  WHEN 'blue'  THEN 1
  WHEN 'brown' THEN 2
```

The results of a `CASE` expression can be used to set properties on a node or relationship.

```
  ELSE 3
END AS colorCode
SET n.colorCode = colorCode
RETURN n.name, n.colorCode
```

Label expressions

```
MATCH (n:Movie|Person)
RETURN n.name AS name, n.title AS title
```

Node pattern using the `OR ( | )` label expression.

```
MATCH (n:!Movie)
RETURN labels(n) AS label, count(n) AS
labelCount
```

Node pattern using the negation `( ! )` label expression.

```
MATCH (:Movie {title: 'Wall Street'})<-
[:ACTED_IN|DIRECTED]-(person:Person)
RETURN person.name AS person
```

Relationship pattern using the `OR ( | )` label expression. As relationships can only have exactly one type each, `()-[:A&B]→()` will never match a relationship.

List expressions

```
WITH [1, 2, 3, 4] AS list
RETURN list[0] AS firstElement,
       list[2] AS thirdElement,
       list[-1] AS finalElement
```

The subscript operator, `[]`, can be used to access specific elements in a `LIST`. `[0]` refers to the first element in a `LIST`, `[1]` to the second, and so on. `[-1]` refers to the last element in a `LIST`, `[-2]` to the penultimate element, and so on.

```
WITH [[1, 2], [3, 4], [5, 6]] AS
nestedList
RETURN nestedList[1] AS secondList
```

Access a `LIST` within a nested `LIST`.

```
WITH [[1, 2], [3, 4], [5, 6]] AS
nestedList
RETURN nestedList[1] AS secondList,
       nestedList[1][0] AS
firstElementOfSecondList
```

Access specific elements in a nested `LIST`.

```
WITH [1, 2, 3, 4, 5, 6] AS list
RETURN list[2..4] AS middleElements,
       list[..2] AS noLowerBound,
       list[2..] AS noUpperBound
```

`LIST` values can be sliced if a range is provided within the subscript operator `[]`. The bounds of the range are separated using two dots (`..`). This allows for extracting a subset of a `LIST` rather than a single element. List slicing is inclusive at the start of the range, but exclusive at the end (e.g. `list[start..end]` includes `start`, but excludes `end`).

```
WITH [1, 2, 3, 4, 5, 6] AS list
RETURN list[..-1] AS finalElementRemoved,
       list[..-2] AS
finalTwoElementsRemoved,
       list[-3..-1] AS
removedFirstThreeAndLast
```

Negative indexing in list slicing references elements from the end of the `LIST`; `..-1` excludes the last element, `..-2` excludes the last two elements, and so on.

```
WITH [[1, 2, 3], [4, 5, 6], [7, 8, 9]] AS
nestedList
RETURN nestedList[1][0..2] AS
slicedInnerList
```

Slicing inner `LIST` values require two `[]` operators; the first `[]` accesses elements from the outer `LIST`, while the second slices or accesses elements from the inner `LIST`.

```
RETURN [1,2] || [3,4] AS list1,
       [1,2] + [3,4] AS list2
```

Cypher contains two list concatenation operators: `||` and `+`. They are functionally equivalent but `||` is GQL conformant and `+` is not.

```
WITH [1, 2, 3, 4] AS list
RETURN 0 + list AS newBeginning,
       list + 5 AS newEnd
```

The `+` operator can add elements to the beginning or end of a `LIST` value. This is not possible using the `||` operator.

```
WITH [1, 2, 3, 4, 5] AS list
RETURN [n IN list WHERE n > 2 | n] AS
filteredList
```

List comprehension is used to create new `LIST` values by iterating over existing `LIST` values and transforming the elements based on certain conditions or operations. This process effectively maps each element in the original `LIST` to a new value. The result is a new `LIST` that consists of the transformed elements.

```
MATCH (p:Person) WHERE p.skills IS NOT
NULL
ORDER BY p.name
RETURN p.name AS name,
      [skill IN p.skills | skill + "
expert"] AS modifiedSkills
```

List comprehension using node properties.

```
MATCH (p:Person)
RETURN [person IN collect(p) WHERE
'Python' IN person.skills | person.name]
AS pythonExperts
```

List comprehension with a `WHERE` predicate.

```
RETURN [x IN ([1, null, 3] || [null, 5,
null]) WHERE x IS NOT NULL] AS
listWithoutNull
```

List comprehension can be used to remove any unknown `NULL` values when concatenating `LIST` values.

```
MATCH (alice:Person {name: 'Alice'})
RETURN [(employee:Person)-[:WORKS_FOR]->
(alice) | employee.name] AS employees
```

Pattern comprehension is used to create new `LIST` values by matching graph patterns and applying conditions to the matched elements, returning custom projections.

```
MATCH (alice:Person {name: 'Alice'})
RETURN [(employee:Person)-[:WORKS_FOR]->
(alice) WHERE employee.age > 30 |
employee.name || ', ' ||
toString(employee.age)] AS
employeesAbove30
```

Pattern comprehension with a `WHERE` predicate.

```
MATCH (cecil:Person {name: 'Cecil'})
WITH [(cecil)-[:WORKS_FOR*]->
(superior:Person) | superior.skills] AS
allSuperiorsSkills
WITH reduce(accumulatedSkills = [],
superiorSkills IN allSuperiorsSkills |
accumulatedSkills || superiorSkills) AS
allSkills
UNWIND allSkills AS superiorsSkills
RETURN collect(DISTINCT superiorsSkills)
AS distinctSuperiorsSkills
```

Variable-length pattern comprehension.

Node and relationship operators

```
MATCH (employee:Person)-[r:WORKS_FOR]->
(manager:Person)
RETURN employee.firstName AS employee,
       r.since AS employedSince,
       manager.firstName AS manager
```

Property values of nodes and relationships can be accessed statically by specifying a property name after the `.` operator.

```
LET nodeProperty = 'lastName'
MATCH (p:Person)
RETURN p[nodeProperty] AS lastName
```

Property values can be accessed dynamically by using the subscript operator `[]`.

```
MATCH (p:Person)
RETURN p.firstName || coalesce(' ' +
p.middleName, '') || ' ' || p.lastName AS
fullName
```

If a property (or property value) is missing in an expression that tries to access a property statically or dynamically, the whole expression will evaluate to `NULL`. The `coalesce()` function can be used to skip the first `NULL` value in an expression.

Mathematical operators

`RETURN 10 + 5 AS result`	The addition operator `+` is used to add numeric values.
`RETURN 10 - 5 AS result`	The subtraction operator `-` is used to subtract numeric values.
`RETURN 10 * 5 AS result`	The multiplication operator `*` is used to multiply numeric values.
`RETURN 10 / 5 AS result`	The division operator `/` is used to divide numeric values.
`RETURN 10 % 3 AS result`	The modulo division operation `%` returns the remainder when one number is divided by another.
`RETURN 10 ^ 5 AS result`	The exponentiation operator `^` raises a number to the power of another.

Map expressions

```
WITH {a: 10, b: 20, c: 30} AS map
RETURN map.a AS firstValue,
       map.c AS lastValue
```

`MAP` values can be accessed statically by specifying a key after the `.` operator.

```
WITH {a: 10, b: 20, c: 30, innerMap: {x:
100, y: 200, z: 300}} AS map
RETURN map.a AS firstOuterValue,
       map.innerMap.y AS secondInnerValue
```

To statically access a value in a nested `MAP`, use chained `.` operators. Each `.` operator traverses one level deeper into the nested structure.

```
WITH {a: 10, b: 20, c: 30} AS map,
     'a' AS dynamicKey
RETURN map[dynamicKey] AS dynamicValue
```

To dynamically access a `MAP` value, use the subscript operator, `[]`. The key can be provided by a variable or a parameter.

```
WITH {a: 10, b: 20, c: 30, innerMap: {x:
100, y: 200, z: 300}} AS map,
     'z' AS dynamicInnerKey
RETURN map.innerMap[dynamicInnerKey] AS
dynamicInnerValue
```

Dynamically access a nested `MAP` value.

```
WITH {a: 10, b: 20, c: 30} AS map
RETURN map{.a, .c} AS projectedMap
```

Map projection with a key selector to extract specific key-value pairs from a `MAP`.

```
WITH {a: 10, b: 20, c: 30} AS map
RETURN map{a: map.a, valueSum: map.a +
map.b + map.c} AS projectedMap
```

Map projection with a literal entry to add custom values to a projected `MAP` value without modifying the original data structure.

```
MATCH (keanu:Person {name: 'Keanu
Reeves'})
LET dob = date('1964-09-02'), birthPlace
=  'Beirut, Lebanon'
```

Map projection with a variable selector to project values based on a variable name.

```
RETURN keanu{.name, dob, birthPlace} AS
projectedKeanu
```

```
WITH {a: 10, b: 20, c: 30} AS map
RETURN map{.*} AS projectedMap
```

Map projection with an all-map projection to project all key-value pairs from a `MAP` without explicitly listing them.

String concatenation operators

```
RETURN 'Neo' || '4j' AS result1,
       'Neo' + '4j' AS result2
```

Cypher contains two operators for the concatenation of `STRING` values: `||` and `+`. The two operators are functionally equivalent. However, `||` is GQL conformant, while `+` is not.

```
RETURN 'Alpha' || 'Beta' AS result1,
       'Alpha' || ' ' || 'Beta' AS
result2
```

Cypher does not insert spaces when concatenating `STRING` values.

```
CREATE (p:Person {firstName: 'Keanu',
lastName: 'Reeves'})
SET p.fullName = p.firstName || ' ' ||
p.lastName
RETURN p.fullName AS fullName
```

String concatenation on two `STRING` properties.

```
RETURN 'My favorite fruits are: ' ||
'apples' || ', ' || 'bananas' || ', and '
|| 'oranges' || '.' AS result
```

String concatenation adding a prefix, suffix, and separator.

```
WITH ['Neo', '4j'] AS list
RETURN reduce(acc = '', item IN list| acc
|| item) AS result
```

`STRING` values in a `LIST` can be concatenated using the `reduce()` function.

```
WITH ['Apples', null, 'Bananas', null,
'Oranges', null] AS list
RETURN 'My favorite fruits are: ' ||
reduce(acc = head(list), item IN
tail(list) | acc || coalesce(', ' ||
item, '')) || '.' AS result
```

Concatenating a `STRING` value with `NULL` returns `NULL`. To skip the first `NULL` value in a list of expressions, use the `coalesce()` function.

```
WITH ['Apples', 'Bananas', 'Oranges'] AS
list
RETURN [item IN list | 'Eat more ' ||
item || '!'] AS result
```

List comprehension allows concatenating a `STRING` value to each item in a `LIST` to generate a new `LIST` of modified `STRING` values.

Temporal operators

```
WITH localdatetime({year:1984, month:10,
day:11, hour:12, minute:31, second:14})
AS aDateTime,
     duration({years: 12, nanoseconds:
2}) AS aDuration
RETURN aDateTime + aDuration AS addition,
      aDateTime - aDuration AS
subtraction
```

`DURATION` values can be added and subtracted from temporal instant values, such as `LOCAL DATETIME`.

```
WITH duration({days: 14, minutes: 12,
seconds: 70, nanoseconds: 1}) AS
aDuration
RETURN aDuration,
       aDuration * 2 AS
multipliedDuration,
       aDuration / 3 AS dividedDuration
```

When multiplying or dividing a `DURATION`, each component is handled separately. In multiplication, the value of each component is multiplied by the given factor, while in division, each component is divided by the given number. If the result of the division does not fit into the original components, it overflows into smaller components (e.g. converting days into hours).

Functions

Aggregating functions

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

The `avg` function returns the average of a set of `INTEGER` or `FLOAT` values.

```
UNWIND [duration('P2DT3H'),
duration('PT1H45S')] AS dur
RETURN avg(dur)
```

The `avg duration` function returns the average of a set of `DURATION` values.

```
MATCH (p:Person)
RETURN collect(p.age)
```

The `collect` function returns a single aggregated list containing the non- `null` values returned by an expression.

```
MATCH (p:Person {name: 'Keanu Reeves'})--
>(x)
RETURN labels(p), p.age, count(*)
```

The `count` function returns the number of values or rows. When `count(*)` is used, the function returns the number of matching rows.

```
MATCH (p:Person)
RETURN count(p.age)
```

The `count` function can also be passed an expression. If so, it returns the number of non- `null` values returned by the given expression.

```
MATCH (p:Person)
RETURN max(p.age)
```

The `max` function returns the maximum value in a set of values.

```
MATCH (p:Person)
RETURN min(p.age)
```

The `min` function returns the minimum value in a set of values.

```
MATCH (p:Person)
RETURN percentileCont(p.age, 0.4)
```

The `percentileCont` function returns the percentile of the given value over a group, with a percentile from `0.0` to `1.0`. It uses a

linear interpolation method, calculating a weighted average between two values if the desired percentile lies between them.

```
MATCH (p:Person)
RETURN percentileDisc(p.age, 0.5)
```

The `percentileDisc` function returns the percentile of the given value over a group, with a percentile from `0.0` to `1.0`. It uses a rounding method and calculates the nearest value to the percentile.

```
MATCH (p:Person)
WHERE p.name IN ['Keanu Reeves', 'Liam
Neeson', 'Carrie Anne Moss']
RETURN stDev(p.age)
```

The `stDev` function returns the standard deviation for the given value over a group. It uses a standard two-pass method, with `N - 1` as the denominator, and should be used when taking a sample of the population for an unbiased estimate.

```
MATCH (p:Person)
WHERE p.name IN ['Keanu Reeves', 'Liam
Neeson', 'Carrie Anne Moss']
RETURN stDevP(p.age)
```

The `stDevP` function returns the standard deviation for the given value over a group. It uses a standard two-pass method, with `N` as the denominator, and should be used when calculating the standard deviation for an entire population.

```
MATCH (p:Person)
RETURN sum(p.age)
```

The `sum` function returns the sum of a set of numeric values.

```
UNWIND [duration('P2DT3H'),
duration('PT1H45S')] AS dur
RETURN sum(dur)
```

The `sum duration` function returns the sum of a set of durations.

Database functions

```
WITH "2:efc7577d-022a-107c-a736-
dbcdfc189c03:0" AS eid
RETURN db.nameFromElementId(eid) AS name
```

The `db.nameFromElementId` function returns the name of a database to which the element id belongs. The name of the database can only be returned if the provided element id belongs to a standard database in the DBMS.

Duration functions

```
UNWIND [
duration({days: 14, hours:16, minutes:
12}),
duration({months: 5, days: 1.5}),
duration({months: 0.75}),
duration({weeks: 2.5}),
duration({minutes: 1.5, seconds: 1,
milliseconds: 123, microseconds: 456,
nanoseconds: 789}),
duration({minutes: 1.5, seconds: 1,
nanoseconds: 123456789})
] AS aDuration
RETURN aDuration
```

The `duration` function can construct a `DURATION` from a `MAP` of its components.

```
UNWIND [
duration("P14DT16H12M"),
duration("P5M1.5D"),
duration("P0.75M"),
duration("PT0.75M"),
duration("P2012-02-02T14:37:21.545")
] AS aDuration
RETURN aDuration
```

The `duration from a string` function returns the `DURATION` value obtained by parsing a `STRING` representation of a temporal amount.

```
UNWIND [
duration.between(date("1984-10-11"),
date("1985-11-25")),
duration.between(date("1985-11-25"),
date("1984-10-11")),
duration.between(date("1984-10-11"),
datetime("1984-10-
12T21:40:32.142+0100")),
duration.between(date("2015-06-24"),
localtime("14:30")),
duration.between(localtime("14:30"),
time("16:30+0100")),
duration.between(localdatetime("2015-07-
21T21:40:32.142"), localdatetime("2016-
07-21T21:45:22.142")),
duration.between(datetime({year: 2017,
month: 10, day: 29, hour: 0, timezone:
'Europe/Stockholm'}), datetime({year:
2017, month: 10, day: 29, hour: 0,
timezone: 'Europe/London'}))
```

The `duration.between` function returns the `DURATION` value equal to the difference between the two given instants.

```
] AS aDuration
RETURN aDuration
```

```
UNWIND [
duration.inMonths(date("1984-10-11"),
date("1985-11-25")),
duration.inMonths(date("1985-11-25"),
date("1984-10-11")),
duration.inMonths(date("1984-10-11"),
datetime("1984-10-
12T21:40:32.142+0100")),
duration.inMonths(date("2015-06-24"),
localtime("14:30")),
duration.inMonths(localdatetime("2015-07-
21T21:40:32.142"), localdatetime("2016-
07-21T21:45:22.142")),
duration.inMonths(datetime({year: 2017,
month: 10, day: 29, hour: 0, timezone:
```

View all (3 more lines)

The `duration.inDays` function returns the `DURATION` value equal to the difference in whole days or weeks between the two given instants.

```
UNWIND [
duration.inDays(date("1984-10-11"),
date("1985-11-25")),
duration.inDays(date("1985-11-25"),
date("1984-10-11")),
duration.inDays(date("1984-10-11"),
datetime("1984-10-
12T21:40:32.142+0100")),
duration.inDays(date("2015-06-24"),
localtime("14:30")),
duration.inDays(localdatetime("2015-07-
21T21:40:32.142"), localdatetime("2016-
07-21T21:45:22.142")),
duration.inDays(datetime({year: 2017,
month: 10, day: 29, hour: 0, timezone:
'Europe/Stockholm'}), datetime({year:
2017, month: 10, day: 29, hour: 0,
timezone: 'Europe/London'}))
] AS aDuration
RETURN aDuration
```

The `duration.inMonths` function returns the `DURATION` value equal to the difference in whole months between the two given instants.

```
UNWIND [
duration.inSeconds(date("1984-10-11"),
date("1984-10-12")),
duration.inSeconds(date("1984-10-12"),
date("1984-10-11")),
duration.inSeconds(date("1984-10-11"),
datetime("1984-10-
12T01:00:32.142+0100")),
duration.inSeconds(date("2015-06-24"),
localtime("14:30")),
```

The `duration.inSeconds` function returns the `DURATION` value equal to the difference in seconds and nanoseconds between the two given instants.

```
duration.inSeconds(datetime({year: 2017,
month: 10, day: 29, hour: 0, timezone:
'Europe/Stockholm'}), datetime({year:
2017, month: 10, day: 29, hour: 0,
timezone: 'Europe/London'}))
] AS aDuration
RETURN aDuration
```

Graph functions

```
RETURN graph.names() AS name
```

The `graph.names` function returns a list containing the names of all graphs on the current composite database. It is only supported on composite databases.

```
UNWIND graph.names() AS name
RETURN name, graph.propertiesByName(name)
AS props
```

The `graph.propertiesByName` function returns a map containing the properties associated with the given graph. The properties are set on the alias that adds the graph as a constituent of a composite database. It is only supported on composite databases.

```
UNWIND graph.names() AS graphName
CALL () {
  USE graph.byName(graphName)
  MATCH (n)
  RETURN n
}
RETURN n
```

The `graph.byName` function resolves a constituent graph by name. It is only supported in the USE clause on composite databases.

```
USE graph.byElementId("4:c0a65d96-4993-
4b0c-b036-e7ebd9174905:0")
MATCH (n) RETURN n
```

The `graph.byElementId` function is used in the USE clause to resolve a constituent graph to which a given element id belongs. If the constituent database is not a standard database in the DBMS, an error will be thrown.

List functions

```
MATCH (a) WHERE a.name = 'Alice'
RETURN keys(a)
```

The `keys` function returns a `LIST<STRING>` containing the `STRING` representations for all the property names of a `NODE`, `RELATIONSHIP`, or `MAP`.

```
MATCH (a) WHERE a.name = 'Alice'
RETURN labels(a)
```

The `labels` function returns a `LIST<STRING>` containing the `STRING` representations for all the labels of a `NODE`.

```
MATCH p = (a)-->(b)-->(c)
WHERE a.name = 'Alice' AND c.name =
'Eskil'
RETURN nodes(p)
```

The `nodes` function returns a `LIST<NODE>` containing all the `NODE` values in a `PATH`.

```
RETURN range(0, 10), range(2, 18, 3),
range(0, 5, -1)
```

The `range` function returns a `LIST<INTEGER>` comprising all `INTEGER` values within a range bounded by a start value and an end value, where the difference step between any two consecutive values is constant; i.e. an arithmetic progression.

```
MATCH p = (a)-->(b)-->(c)
WHERE a.name = 'Alice' AND b.name = 'Bob'
AND c.name = 'Daniel'
RETURN reduce(totalAge = 0, n IN nodes(p)
| totalAge + n.age) AS reduction
```

The `reduce` function returns the value resulting from the application of an expression on each successive element in a list in conjunction with the result of the computation thus far.

```
MATCH p = (a)-->(b)-->(c)
WHERE a.name = 'Alice' AND c.name =
'Eskil'
RETURN relationships(p)
```

The `relationships` function returns a `LIST<RELATIONSHIP>` containing all the `RELATIONSHIP` values in a `PATH`.

```
WITH [4923,'abc',521, null, 487] AS ids
RETURN reverse(ids)
```

The `reverse` function returns a `LIST<ANY>` in which the order of all elements in the given `LIST<ANY>` have been reversed.

```
MATCH (a) WHERE a.name = 'Eskil'
RETURN a.likedColors, tail(a.likedColors)
```

The `tail` function returns a `LIST<ANY>` containing all the elements, excluding the first one, from a given `LIST<ANY>`.

```
RETURN toBooleanList(null) as noList,
toBooleanList([null, null]) as
nullsInList,
toBooleanList(['a string', true, 'false',
null, ['A','B']]) as mixedList
```

The `toBooleanList` converts a `LIST<ANY>` and returns a `LIST<BOOLEAN>`. If any values are not convertible to `BOOLEAN` they will be `null` in the `LIST<BOOLEAN>` returned.

```
RETURN toFloatList(null) as noList,
toFloatList([null, null]) as nullsInList,
toFloatList(['a string', 2.5, '3.14159',
null, ['A','B']]) as mixedList
```

The `toFloatList` converts a `LIST<ANY>` of values and returns a `LIST<FLOAT>`. If any values are not convertible to `FLOAT` they will be `null` in the `LIST<FLOAT>` returned.

```
RETURN toIntegerList(null) as noList,
toIntegerList([null, null]) as
nullsInList,
toIntegerList(['a string', 2, '5', null,
['A','B']]) as mixedList
```

The `toIntegerList` converts a `LIST<ANY>` of values and returns a `LIST<INTEGER>`. If any values are not convertible to `INTEGER` they will be `null` in the `LIST<INTEGER>` returned.

```
RETURN toStringList(null) as noList,
toStringList([null, null]) as
nullsInList,
toStringList(['already a string', 2,
date({year:1955, month:11, day:5}), null,
['A','B']]) as mixedList
```

The `toStringList` converts a `LIST<ANY>` of values and returns a `LIST<STRING>`. If any values are not convertible to `STRING` they will be `null` in the `LIST<STRING>` returned.

Mathematical functions - numerical

```
MATCH (a), (e) WHERE a.name = 'Alice' AND
e.name = 'Eskil'
RETURN a.age, e.age, abs(a.age - e.age)
```

The `abs` function returns the absolute value of the given number.

```
RETURN ceil(0.1)
```

The `ceil` function returns the smallest `FLOAT` that is greater than or equal to the given number and equal to an `INTEGER`.

```
RETURN floor(0.9)
```

The `floor` function returns the largest `FLOAT` that is less than or equal to the given number and equal to an `INTEGER`.

```
RETURN isNaN(0/0.0)
```

The `isNan` function returns `true` if the given numeric value is `NaN` (Not a Number).

```
RETURN rand()
```

The `rand` function returns a random `FLOAT` in the range from 0 (inclusive) to 1 (exclusive). The numbers returned follow an approximate uniform distribution.

```
RETURN round(3.141592)
```

The `round` function returns the value of the given number rounded to the nearest `INTEGER`, with ties always rounded towards positive infinity.

```
RETURN round(3.141592, 3)
```

The `round` with precision function returns the value of the given number rounded to the closest value of given precision, with ties always being rounded away from zero (using rounding mode `HALF_UP`). The exception is for precision 0, where ties are rounded towards positive infinity to align with `round()` without precision.

```
RETURN round(1.249, 1, 'UP') AS positive,
round(-1.251, 1, 'UP') AS negative,
round(1.25, 1, 'UP') AS positiveTie,
round(-1.35, 1, 'UP') AS negativeTie
```

The `round` with precision and rounding mode function returns the value of the given number rounded with the specified precision and the specified rounding mode.

```
RETURN sign(-17), sign(0.1)
```

The `sign` function returns the signum of the given number: `0` if the number is 0, `-1` for any negative number, and `1` for any positive number.

Mathematical functions - logarithmic

`RETURN e()`	The `e` function returns the base of the natural logarithm, *e*.
`RETURN exp(2)`	The `exp` function returns e^n, where `e` is the base of the natural logarithm, and `n` is the value of the argument expression.
`RETURN log(27)`	The `log` function returns the natural logarithm of a number.
`RETURN log10(27)`	The `log10` function returns the common logarithm (base 10) of a number.
`RETURN sqrt(256)`	The `sqrt` function returns the square root of a number.

Mathematical Functions - trigonometric

`RETURN acos(0.5)`	The `acos` function returns the arccosine of a `FLOAT` in radians.
`RETURN asin(0.5)`	The `asin` function returns the arcsine of a `FLOAT` in radians.
`RETURN atan(0.5)`	The `atan` function returns the arctangent of a `FLOAT` in radians.
`RETURN atan2(0.5, 0.6)`	The `atan2` function returns the arctangent2 of a set of coordinates in radians.
`RETURN cos(0.5)`	The `cos` function returns the cosine of a `FLOAT`.
`RETURN cosh(0.7)`	The `cosh` function returns the hyperbolic cosine of a `FLOAT`.
`RETURN cot(0.5)`	The `cot` function returns the cotangent of a `FLOAT`.
`RETURN coth(0.7)`	The `coth` function returns the hyperbolic cotangent of a `FLOAT`.
`RETURN degrees(3.14159)`	The `degrees` function converts radians to degrees.
`RETURN haversin(0.5)`	The `haversin` function converts half the versine of a number.

`RETURN pi()`	The `pi` function returns the mathematical constant *pi*.
`RETURN radians(180)`	The `radians` function converts degrees to radians.
`RETURN sin(0.5)`	The `sin` function returns the sine of a number.
`RETURN sinh(0.7)`	The `sinh` function returns the hyperbolic sine of a `FLOAT`.
`RETURN tan(0.5)`	The `tan` function returns the tangent of a number.
`RETURN tanh(0.7)`	The `tanh` function returns the hyperbolic tangent of a `FLOAT`.

Predicate functions

```
MATCH p = (a)-[*]->(b)
WHERE
  a.name = 'Keanu Reeves'
  AND b.name = 'Guy Pearce'
  AND all(x IN nodes(p) WHERE x.age < 60)
RETURN p
```

The `all` function returns `true` if the predicate holds for all elements in the given `LIST<ANY>`.

```
MATCH (p:Person)
WHERE any(nationality IN p.nationality
WHERE nationality = 'American')
RETURN p
```

The `any` function returns `true` if the predicate holds for at least one element in the given `LIST<ANY>`.

```
MATCH (p:Person)
RETURN
  p.name AS name,
  exists((p)-[:ACTED_IN]->()) AS
has_acted_in_rel
```

The `exists` function returns `true` if a match for the given pattern exists in the graph.

```
MATCH (p:Person)
WHERE NOT isEmpty(p.nationality)
RETURN p.name, p.nationality
```

The `isEmpty` function returns `true` if the given `LIST<ANY>` or `MAP` contains no elements, or if the given `STRING` contains no characters.

```
MATCH p = (n)-[*]->(b)
WHERE
  n.name = 'Keanu Reeves'
  AND none(x IN nodes(p) WHERE x.age >
60)
RETURN p
```

The `none` function returns `true` if the predicate does not hold for any element in the given `LIST<ANY>`.

```
MATCH p = (n)-->(b)
WHERE
  n.name = 'Keanu Reeves'
  AND single(x IN nodes(p) WHERE
```

The `single` function returns `true` if the predicate holds for exactly *one* of the elements in the given `LIST<ANY>`.

```
        x.nationality = 'Northern Irish')
        RETURN p
```

Scalar functions

```
RETURN char_length('Alice')
```

The `char_length` function returns the number of Unicode characters in a `STRING`. This function is an alias of the `size` function.

```
RETURN character_length('Alice')
```

The `character_length` function returns the number of Unicode characters in a `STRING`. This function is an alias of the `size` function.

```
MATCH (a)
WHERE a.name = 'Alice'
RETURN coalesce(a.hairColor, a.eyes)
```

The `coalesce` function returns the first given non-null argument.

```
MATCH (n:Developer)
RETURN elementId(n)
```

The `elementId` function returns a `STRING` representation of a node or relationship identifier, unique within a specific transaction and DBMS.

```
MATCH (x:Developer)-[r]-()
RETURN endNode(r)
```

The `endNode` function returns the the end `NODE` of a `RELATIONSHIP`.

```
MATCH (a)
WHERE a.name = 'Eskil'
RETURN a.likedColors, head(a.likedColors)
```

The `head` function returns the first element of the list. Returns `null` for an empty list. Equivalent to the list indexing `$list[0]`.

```
MATCH (a)
RETURN id(a)
```

The `id` function returns an `INTEGER` (the internal ID of a node or relationship). Do not rely on the internal ID for your business domain; the internal ID can change between transactions. The `id` function will be

removed in the next major release. It is recommended to use `elementId` instead.

```
MATCH (a)
WHERE a.name = 'Eskil'
RETURN a.likedColors, last(a.likedColors)
```

The `last` function returns the last element of the list. Returns `null` for an empty list. Equivalent to the list indexing `$list[-1]`.

```
MATCH p = (a)-->(b)-->(c)
WHERE a.name = 'Alice'
RETURN length(p)
```

The `length` function returns the length of a `PATH`.

```
RETURN nullIf("abc", "def")
```

The `nullIf` function returns `null` if the two given parameters are equivalent, otherwise it returns the value of the first parameter.

```
CREATE (p:Person {name: 'Stefan', city:
'Berlin'})
RETURN properties(p)
```

The `properties` function returns a `MAP` containing all the properties of a node or relationship.

```
RETURN randomUUID() AS uuid
```

The `randomUUID` function returns a `STRING`; a randomly-generated universally unique identifier (UUID).

```
RETURN size(['Alice', 'Bob'])
```

The `size` function returns the number of elements in the list.

```
MATCH (x:Developer)-[r]-()
RETURN startNode(r)
```

The function `startNode` function returns the start `NODE` of a `RELATIONSHIP`.

```
RETURN timestamp()
```

The `timestamp` function returns the time in milliseconds since `midnight, January 1, 1970 UTC.` and the current time.

```
RETURN toBoolean('true'), toBoolean('not
a boolean'), toBoolean(0)
```

The `toBoolean` function converts a `STRING`, `INTEGER` or `BOOLEAN` value to a `BOOLEAN` value.

```
RETURN toBooleanOrNull('true'),
toBooleanOrNull('not a boolean'),
toBooleanOrNull(0), toBooleanOrNull(1.5)
```

The `toBooleanOrNull` function converts a `STRING`, `INTEGER` or `BOOLEAN` value to a `BOOLEAN` value. For any other input value, `null` will be returned.

```
RETURN toFloat('11.5'), toFloat('not a
number')
```

The `toFloat` function converts an `INTEGER`, `FLOAT` or a `STRING` value to a `FLOAT`.

```
RETURN toFloatOrNull('11.5'),
toFloatOrNull('not a number'),
toFloatOrNull(true)
```

The `toFloatOrNull` function converts an `INTEGER`, `FLOAT` or a `STRING` value to a `FLOAT`. For any other input value, `null` will be returned.

```
RETURN toInteger('42'), toInteger('not a
number'), toInteger(true)
```

The `toInteger` function converts a `BOOLEAN`, `INTEGER`, `FLOAT` or a `STRING` value to an `INTEGER` value.

```
RETURN toIntegerOrNull('42'),
toIntegerOrNull('not a number'),
toIntegerOrNull(true),
toIntegerOrNull(['A', 'B', 'C'])
```

The `toIntegerOrNull` function converts a `BOOLEAN`, `INTEGER`, `FLOAT` or a `STRING` value to an `INTEGER` value. For any other input value, `null` will be returned.

```
MATCH (n)-[r]->()
WHERE n.name = 'Alice'
RETURN type(r)
```

The `type` function returns the `STRING` representation of the `RELATIONSHIP` type.

```
UNWIND ["abc", 1, 2.0, true, [date()]] AS
value
RETURN valueType(value) AS result
```

The `valueType` function returns a `STRING` representation of the most precise value type that the given expression evaluates to.

String functions

```
RETURN btrim('   hello    '),
btrim('xxyyhelloxyxy', 'xy')
```

The `btrim` function returns the original `STRING` with leading and trailing `trimCharacterString` characters removed. If `trimCharacterString` is not specified then all leading and trailing whitespace will be removed.

```
RETURN left('hello', 3)
```

The `left` function returns a `STRING` containing the specified number of leftmost characters of the given `STRING`.

```
RETURN lower('HELLO')
```

The `lower` function returns the given `STRING` in lowercase. This function is an alias of the `toLower` function.

```
RETURN ltrim('   hello'),
ltrim('xxyyhelloxyxy', 'xy')
```

The `ltrim` function returns the original `STRING` with leading `trimCharacterString` characters removed. If `trimCharacterString` is not specified then all leading whitespace will be removed.

```
RETURN normalize('\u212B') = '\u00C5' AS
result
```

The `normalize` function returns a given `STRING` normalized using the `NFC` Unicode normalization form.

```
RETURN replace("hello", "l", "w")
```

```
RETURN replace("hello", "l", "w", 1)
```

The `replace` function returns a `STRING` in which all occurrences of a specified `STRING` in the given `STRING` have been replaced by another (specified) replacement `STRING`.

```
RETURN reverse('palindrome')
```

The `reverse` function returns a `STRING` in which the order of all characters in the given `STRING` have been reversed.

```
RETURN right('hello', 3)
```

The `right` function returns a `STRING` containing the specified number of rightmost characters in the given `STRING`.

```
RETURN rtrim('hello   '),
rtrim('xxyyhelloxyxy', 'xy')
```

The `rtrim` function returns the given `STRING` with trailing `trimCharacterString` characters removed. If `trimCharacterString` is not specified then all trailing whitespace will be removed.

```
RETURN split('one,two', ',')
```

The `split` function returns a `LIST<STRING>` resulting from the splitting of the given `STRING` around matches of the given delimiter.

```
RETURN substring('hello', 1, 3),
substring('hello', 2)
```

The `substring` function returns a substring of the given `STRING`, beginning with a zero-based index start and length.

```
RETURN toLower('HELLO')
```

The `toLower` function returns the given `STRING` in lowercase.

```
RETURN
  toString(11.5),
  toString('already a string'),
  toString(true),
  toString(date({year: 1984, month: 10,
day: 11})) AS dateString,
  toString(datetime({year: 1984, month:
10, day: 11, hour: 12, minute: 31,
second: 14, millisecond: 341, timezone:
'Europe/Stockholm'})) AS datetimeString,
```

The `toString` function converts an `INTEGER`, `FLOAT`, `BOOLEAN`, `STRING`, `POINT`, `DURATION`, `DATE`, `ZONED TIME`, `LOCAL TIME`, `LOCAL DATETIME` or `ZONED DATETIME` value to a `STRING`.

```
  toString(duration({minutes: 12,
seconds: -60})) AS durationString
```

```
RETURN toStringOrNull(11.5),
toStringOrNull('already a string'),
toStringOrNull(true),
toStringOrNull(date({year: 1984, month:
10, day: 11})) AS dateString,
toStringOrNull(datetime({year: 1984,
month: 10, day: 11, hour: 12, minute: 31,
second: 14, millisecond: 341, timezone:
'Europe/Stockholm'})) AS datetimeString,
toStringOrNull(duration({minutes: 12,
seconds: -60})) AS durationString,
toStringOrNull(['A', 'B', 'C']) AS list
```

The `toStringOrNull` function converts an `INTEGER`, `FLOAT`, `BOOLEAN`, `STRING`, `POINT`, `DURATION`, `DATE`, `ZONED TIME`, `LOCAL TIME`, `LOCAL DATETIME` or `ZONED DATETIME` value to a `STRING`. For any other input value, `null` will be returned.

```
RETURN toUpper('hello')
```

The `toUpper` function returns the given `STRING` in uppercase.

```
RETURN trim('   hello   '), trim(BOTH 'x'
FROM 'xxxhelloxxx')
```

The `trim` function returns the given `STRING` with leading and trailing whitespace removed.

```
RETURN upper('hello')
```

The `upper` function returns the given `STRING` in uppercase. This function is an alias of the `toUpper` function.

Spatial functions

```
WITH
  point({longitude: 12.53, latitude:
55.66}) AS lowerLeft,
  point({longitude: 12.614, latitude:
55.70}) AS upperRight
MATCH (t:TrainStation)
WHERE point.withinBBox(point({longitude:
t.longitude, latitude: t.latitude}),
lowerLeft, upperRight)
RETURN count(t)
```

The `point` Cartesian 2D function returns a 2D `POINT` in the *Cartesian* CRS corresponding to the given coordinate values.

```
RETURN
  point.withinBBox(
    null,
    point({longitude: 56.7, latitude:
12.78}),
    point({longitude: 57.0, latitude:
13.0})
  ) AS in
```

The `point` Cartesian 3D function returns a 3D `POINT` in the *Cartesian* CRS corresponding to the given coordinate values.

```
MATCH (t:TrainStation)-[:TRAVEL_ROUTE]->
(o:Office)
WITH
  point({longitude: t.longitude,
latitude: t.latitude}) AS trainPoint,
  point({longitude: o.longitude,
latitude: o.latitude}) AS officePoint
RETURN round(point.distance(trainPoint,
officePoint)) AS travelDistance
```

The `point` WGS 84 2D function returns a 2D `POINT` in the *WGS 84 CRS* corresponding to the given coordinate values.

```
WITH
  point({x: 0, y: 0, crs: 'cartesian'})
AS lowerLeft,
  point({x: 10, y: 10, crs: 'cartesian'})
AS upperRight
RETURN point.withinBBox(point({x: 5, y:
5, crs: 'cartesian'}), lowerLeft,
upperRight) AS result
```

The `point` WGS 84 3D function returns a 3D `POINT` in the *WGS 84 CRS* corresponding to the given coordinate values.

```
MATCH (p:Office)
RETURN point({longitude: p.longitude,
latitude: p.latitude}) AS officePoint
```

The `point.distance` function returns returns a `FLOAT` representing the geodesic distance between two points in the same Coordinate Reference System (CRS).

```
RETURN point({x: 2.3, y: 4.5}) AS point
```

The `point.withinBBox` function takes the following arguments: the `POINT` to check, the lower-left (south-west) `POINT` of a bounding box, and the upper-right (or north-east) `POINT` of a bounding box. The return value will be true if the provided point is contained in the bounding box (boundary included), otherwise the return value will be false.

Temporal functions

```
RETURN date() AS currentDate
```

The `date` function returns the current `DATE` value. If no time zone parameter is specified, the local time zone will be used.

```
RETURN date.realtime() AS currentDate
```

The `date.realtime()` function returns the current `DATE` instant using the realtime clock.

```
RETURN date.statement() AS currentDate
```

The `date.statement()` function returns the current `DATE` instant using the statement clock.

```
RETURN date.transaction() AS currentDate
```

The `date.transaction()` function returns the current `DATE` instant using the transaction clock.

```
WITH
  datetime({
    year: 2017, month: 11, day: 11,
    hour: 12, minute: 31, second: 14,
nanosecond: 645876123,
    timezone: '+01:00'
  }) AS d
RETURN
  date.truncate('millennium', d) AS
truncMillenium,
  date.truncate('century', d) AS
truncCentury,
  date.truncate('decade', d) AS
truncDecade,
  date.truncate('year', d, {day: 5}) AS
truncYear,
  date.truncate('weekYear', d) AS
truncWeekYear,
  date.truncate('quarter', d) AS
truncQuarter,
  date.truncate('month', d) AS
truncMonth,
  date.truncate('week', d, {dayOfWeek:
2}) AS truncWeek,
  date.truncate('day', d) AS truncDay
```

The `date.truncate()` function truncates the given temporal value to a `DATE` instant using the specified unit.

```
RETURN datetime() AS currentDateTime
```

The `datetime()` function creates a `ZONED DATETIME` instant.

```
WITH datetime.fromEpoch(1683000000,
123456789) AS dateTimeFromEpoch
RETURN dateTimeFromEpoch
```

The `datetime.fromEpoch()` function creates a `ZONED DATETIME` given the seconds and nanoseconds since the start of the epoch.

```
WITH
datetime.fromEpochMillis(1724198400000)
AS dateTimeFromMillis
RETURN dateTimeFromMillis
```

The `datetime.fromEpochMillis()` function creates a `ZONED DATETIME` given the milliseconds since the start of the epoch.

```
RETURN datetime.realtime() AS
currentDateTime
```

The `datetime.realtime()` function returns the current `ZONED DATETIME` instant using the realtime clock.

```
RETURN datetime.statement() AS
currentDateTime
```

The `datetime.statement()` function returns the current `ZONED DATETIME` instant using the statement clock.

```
RETURN datetime.transaction() AS
currentDateTime
```

The `datetime.transaction()` function returns the current `ZONED DATETIME` instant using the transaction clock.

```
WITH
  datetime({
    year:2017, month:11, day:11,
    hour:12, minute:31, second:14,
nanosecond: 645876123,
    timezone: '+03:00'
  }) AS d
RETURN
  datetime.truncate('millennium', d,
{timezone: 'Europe/Stockholm'}) AS
truncMillenium,
  datetime.truncate('year', d, {day: 5})
AS truncYear,
  datetime.truncate('month', d) AS
truncMonth,
```

The `datetime.truncate()` function truncates the given temporal value to a `ZONED DATETIME` instant using the specified unit.

```
  datetime.truncate('day', d,
{millisecond: 2}) AS truncDay,
  datetime.truncate('hour', d) AS
truncHour,
  datetime.truncate('second', d) AS
truncSecond
```

```
RETURN
  localdatetime({
    year: 1984, ordinalDay: 202,
    hour: 12, minute: 31, second: 14,
microsecond: 645876
  }) AS theDate
```

The localdatetime() function creates a LOCAL DATETIME instant.

```
RETURN localdatetime.realtime() AS now
```

The localdatetime.realtime() function returns the current LOCAL DATETIME instant using the realtime clock.

```
RETURN localdatetime.statement() AS now
```

The localdatetime.statement() function returns the current LOCAL DATETIME instant using the statement clock.

```
RETURN localdatetime.transaction() AS now
```

The localdatetime.transaction() function returns the current LOCAL DATETIME instant using the transaction clock.

```
WITH
  localdatetime({
    year: 2017, month: 11, day: 11,
    hour: 12, minute: 31, second: 14,
nanosecond: 645876123
  }) AS d
RETURN
  localdatetime.truncate('millennium', d)
AS truncMillenium,
  localdatetime.truncate('year', d, {day:
2}) AS truncYear,
  localdatetime.truncate('month', d) AS
truncMonth,
  localdatetime.truncate('day', d) AS
truncDay,
```

The localdatetime.truncate() function truncates the given temporal value to a LOCAL DATETIME instant using the specified unit.

```
  localdatetime.truncate('hour', d,
{nanosecond: 2}) AS truncHour,
  localdatetime.truncate('second', d) AS
truncSecond
```

```
RETURN localtime() AS now
```

The `localtime()` function creates a `LOCAL TIME` instant.

```
RETURN localtime.realtime() AS now
```

The `localtime.realtime()` function creates a `LOCAL TIME` instant. function returns the current `LOCAL TIME` instant using the realtime clock.

```
RETURN localtime.statement() AS now
```

The `localtime.statement()` function creates a `LOCAL TIME` instant. function returns the current `LOCAL TIME` instant using the statement clock.

```
RETURN localtime.transaction() AS now
```

The `localtime.transaction()` function returns the current `LOCAL TIME` instant using the transaction clock.

```
WITH time({hour: 12, minute: 31, second:
14, nanosecond: 645876123, timezone:
'-01:00'}) AS t
RETURN
  localtime.truncate('day', t) AS
truncDay,
  localtime.truncate('hour', t) AS
truncHour,
  localtime.truncate('minute', t,
{millisecond: 2}) AS truncMinute,
  localtime.truncate('second', t) AS
truncSecond,
  localtime.truncate('millisecond', t) AS
truncMillisecond,
  localtime.truncate('microsecond', t) AS
truncMicrosecond
```

The `localtime.truncate()` function truncates the given temporal value to a `LOCAL TIME` instant using the specified unit.

```
RETURN time() AS currentTime
```

The `time()` function creates a `ZONED TIME` instant.

```
RETURN time.realtime() AS currentTime
```

The `time.realtime()` function returns the current `ZONED TIME` instant using the realtime clock.

```
RETURN time.statement() AS currentTime
```

The `time.statement()` function returns the current `ZONED TIME` instant using the statement clock.

```
RETURN time.transaction() AS currentTime
```

The `time.transaction()` function returns the current `ZONED TIME` instant using the transaction clock.

```
WITH time({hour: 12, minute: 31, second:
14, nanosecond: 645876123, timezone:
'-01:00'}) AS t
RETURN
  time.truncate('day', t) AS truncDay,
  time.truncate('hour', t) AS truncHour,
  time.truncate('minute', t) AS
truncMinute,
  time.truncate('second', t) AS
truncSecond,
  time.truncate('millisecond', t,
{nanosecond: 2}) AS truncMillisecond,
  time.truncate('microsecond', t) AS
truncMicrosecond
```

The `time.truncate()` function truncates the given temporal value to a `ZONED TIME` instant using the specified unit.

Vector functions

```
MATCH (n:Label)
WITH n,
vector.similarity.euclidean($query,
n.vector) AS score
RETURN n, score
```

The `vector.similarity.euclidean` function returns a `FLOAT` representing the similarity between the argument vectors based on their Euclidean distance.

```
MATCH (n:Label)
WITH n, vector.similarity.cosine($query,
n.vector) AS score
RETURN n, score
```

The `vector.similarity.cosine` function returns a `FLOAT` representing the similarity between the argument vectors based on their cosine.

Schema

Search-performance indexes

Cypher includes four search-performance indexes: range (default), text, point, and token lookup.

```
CREATE INDEX index_name
FOR (p:Person) ON (p.name)
```

Create a range index with the name `index_name` on nodes with label `Person` and property `name`.

It is possible to omit the `index_name`, if not specified the index name will be decided by the DBMS. Best practice is to always specify a sensible name when creating an index.

The create syntax is `CREATE [RANGE|TEXT|POINT|LOOKUP|FULLTEXT|VECTOR] INDEX ...`. Defaults to range if not explicitly stated.

```
CREATE RANGE INDEX index_name
FOR ()-[k:KNOWS]-() ON (k.since)
```

Create a range index on relationships with type `KNOWS` and property `since` with the name `index_name`.

```
CREATE INDEX $nameParam
FOR (p:Person) ON (p.name, p.age)
```

Create a composite range index with the name given by the parameter `nameParam` on nodes with label `Person` and the properties `name` and `age`, throws an error if the index already exist.

```
CREATE INDEX index_name IF NOT EXISTS
FOR (p:Person) ON (p.name, p.age)
```

Create a composite range index with the name `index_name` on nodes with label `Person` and the properties `name` and `age` if it does not already exist, does nothing if it did exist.

```
CREATE TEXT INDEX index_name
FOR (p:Person) ON (p.name)
```

Create a text index on nodes with label `Person` and property `name`. Text indexes only solve predicates involving `STRING` property values.

```
CREATE TEXT INDEX index_name
FOR ()-[r:KNOWS]-() ON (r.city)
```

Create a text index on relationships with type `KNOWS` and property `city`. Text indexes only solve predicates involving `STRING` property values.

```
CREATE POINT INDEX index_name
FOR (p:Person) ON (p.location)
OPTIONS {
  indexConfig: {
    `spatial.cartesian.min`: [-100.0,
-100.0],
    `spatial.cartesian.max`: [100.0,
100.0]
  }
}
```

Create a point index on nodes with label `Person` and property `location` with the name `index_name` and the given `spatial.cartesian` settings. The other index settings will have their default values. Point indexes only solve predicates involving `POINT` property values.

```
CREATE POINT INDEX $nameParam
FOR ()-[h:STREET]-() ON (h.intersection)
```

Create a point index with the name given by the parameter `nameParam` on relationships with the type `STREET` and property `intersection`. Point indexes only solve predicates involving `POINT` property values.

```
CREATE LOOKUP INDEX index_name
FOR (n) ON EACH labels(n)
```

Create a token lookup index on nodes with any label.

```
CREATE LOOKUP INDEX index_name
FOR ()-[r]-() ON EACH type(r)
```

Create a token lookup index on relationships with any relationship type.

```
SHOW INDEXES
```

List all indexes, returns only the default outputs (`id`, `name`, `state`, `populationPercent`, `type`, `entityType`, `labelsOrTypes`, `properties`, `indexProvider`, `owningConstraint`, `lastRead`, and `readCount`).

List all indexes and return all columns.

`SHOW INDEXES YIELD *`	
`SHOW INDEX YIELD name, type, entityType, labelsOrTypes, properties`	List all indexes and return only specific columns.
`SHOW INDEXES YIELD name, type, options, createStatement RETURN name, type, options.indexConfig AS config, createStatement`	List all indexes and return only specific columns using the `RETURN` clause. Note that `YIELD` is mandatory if `RETURN` is used.
`SHOW RANGE INDEXES`	List range indexes, can also be filtered on `ALL`, `FULLTEXT`, `LOOKUP`, `POINT`, `TEXT`, and `VECTOR`.
`DROP INDEX index_name`	Drop the index named `index_name`, throws an error if the index does not exist.
`DROP INDEX index_name IF EXISTS`	Drop the index named `index_name` if it exists, does nothing if it does not exist.
`DROP INDEX $nameParam`	Drop an index using a parameter.
`MATCH (n:Person) USING INDEX n:Person(name) WHERE n.name = $value`	Index usage can be enforced when Cypher uses a suboptimal index, or when more than one index should be used.

Full-text indexes

```
CREATE FULLTEXT INDEX node_fulltext_index
FOR (n:Friend) ON EACH [n.name]
OPTIONS {
  indexConfig: {
    `fulltext.analyzer`: 'swedish'
  }
}
```

Create a fulltext index on nodes with the name `index_name` and analyzer `swedish`. The other index settings will have their default values.

```
CREATE FULLTEXT INDEX
relationship_fulltext_index
FOR ()-[r:KNOWS]-() ON EACH [r.info,
r.note]
OPTIONS {
  indexConfig: {
    `fulltext.analyzer`: 'english'
  }
}
```

Create a fulltext index on relationships with the name `index_name` and analyzer `english`. The other index settings will have their default values.

```
CALL db.index.fulltext.queryNodes("node_fulltext_index", "Alice") YIELD node, score
```

Query a full-text index on nodes.

```
CALL db.index.fulltext.queryRelationships("relationship_fulltext_index", "Alice") YIELD
relationship, score
```

Query a full-text index on relationships.

```
SHOW FULLTEXT INDEXES
```

List all full-text indexes.

```
DROP INDEX node_fulltext_index
```

Drop a full-text index.

Vector indexes

```
CREATE VECTOR INDEX `abstract-embeddings`
FOR (a:Abstract) ON (a.embedding)
OPTIONS {
  indexConfig: {
    `vector.dimensions`: 1536,
    `vector.similarity_function`:
'cosine'
  }
}
```

Create a vector index on nodes with label `Abstract`, property `embedding`, and a vector dimension of `1536` using the `cosine` similarity function and the name `abstract-embeddings`. Note that the `OPTIONS` map is mandatory since a vector index cannot be created without setting the vector dimensions and similarity function.

```
CREATE VECTOR INDEX `review-embeddings`
FOR ()-[r:REVIEWED]-() ON (r.embedding)
OPTIONS {
  indexConfig: {
    `vector.dimensions`: 256,
    `vector.similarity_function`:
'cosine'
  }
}
```

Create a vector index on relationships with relationship type `REVIEWED`, property `embedding`, and a vector dimension of `256` using the `cosine` similarity function and the name `review-embeddings`. Note that the `OPTIONS` map is mandatory since a vector index cannot be created without setting the vector dimensions and similarity function.

```
CALL
db.index.vector.queryNodes('abstract-
embeddings', 10, abstract.embedding)
```

Query the node vector index `abstract-embeddings` for a neighborhood of `10` similar abstracts.

```
CALL
db.index.vector.queryRelationships('review-
embeddings', 10, $query)
```

Query the relationship vector index `review-embeddings` for a neighborhood of `10` similar reviews to the vector given by the `query` parameter.

```
MATCH (n:Node {id: $id})
CALL db.create.setNodeVectorProperty(n,
'propertyKey', $vector)
```

Set the vector properties of a node using `db.create.setNodeVectorProperty`.

```
MATCH ()-[r:Relationship {id: $id}]->()
CALL db.create.setRelationshipVectorProperty(r, 'propertyKey', $vector)
```

Set the vector properties of a relationship using `db.create.setRelationshipVectorProperty`.

```
SHOW VECTOR INDEXES
```

List all vector indexes.

```
DROP INDEX `abstract-embeddings`
```

Drop a vector index.

Constraints

`SHOW ALL CONSTRAINTS`	List all constraints, returns only the default outputs (`id` , `name` , `type` , `entityType` , `labelsOrTypes` , `properties` , `ownedIndex` , and `propertyType`). Can also be filtered on `NODE UNIQUENESS` , `RELATIONSHIP UNIQUENESS` , `UNIQUENESS` , `NODE EXISTENCE` , `RELATIONSHIP EXISTENCE` , `EXISTENCE` , `NODE PROPERTY TYPE` , `RELATIONSHIP PROPERTY TYPE` , `PROPERTY TYPE` , `NODE KEY` , `RELATIONSHIP KEY` , and `KEY` . For more information, see Constraints → Syntax → SHOW CONSTRAINTS.
`SHOW CONSTRAINTS YIELD *`	List all constraints. For more information, see Constraints → Create, show, and drop constraints → SHOW CONSTRAINTS.
`DROP CONSTRAINT constraint_name`	Drop the constraint with the name `constraint_name` , throws an error if the constraint does not exist.
`DROP CONSTRAINT $nameParam IF EXISTS`	Drop the constraint with the name given by the parameter `nameParam` if it exists, does nothing if it does not exist.

`CREATE CONSTRAINT constraint_name IF NOT EXISTS`
`FOR (p:Person)`
`REQUIRE p.name IS UNIQUE`
	Create a node property uniqueness constraint on the label `Person` and property `name` . Using the keyword `IF NOT EXISTS` makes the command idempotent, and no error will be thrown if an attempt is made to create the same constraint twice. If any other node with that label is updated or

created with a name that already exists, the write operation will fail.

Best practice is to always specify a sensible name when creating a constraint.

```
CREATE CONSTRAINT constraint_name
FOR (p:Person)
REQUIRE (p.name, p.age) IS UNIQUE
```

Create a node property uniqueness constraint on the label `Person` and properties `name` and `age`. An error will be thrown if an attempt is made to create the same constraint twice. If any node with that label is updated or created with a name and age combination that already exists, the write operation will fail.

```
CREATE CONSTRAINT constraint_name
FOR ()-[r:LIKED]-()
REQUIRE r.when IS UNIQUE
```

Create a relationship property uniqueness constraint on the relationship type `LIKED` and property `when`. If any other relationship with that relationship type is updated or created with a `when` property value that already exists, the write operation will fail.

Best practice is to always specify a sensible name when creating a constraint.

Not available on Neo4j Community Edition

```
CREATE CONSTRAINT $nameParam
FOR (p:Person)
REQUIRE p.name IS NOT NULL
```

Create a node property existence constraint with the name given by the parameter `nameParam` on the label `Person` and property `name`. If a node with that label is created without a `name` property, or if the `name` property on the existing node with the label `Person` is removed, the write operation will fail.

Not available on Neo4j Community Edition

```
CREATE CONSTRAINT constraint_name
FOR ()-[r:LIKED]-()
REQUIRE r.when IS NOT NULL
```

Create a relationship property existence constraint on the type `LIKED` and property `when`. If a relationship with that type is created without a `when` property, or if the property `when` is removed from an existing relationship with the type `LIKED`, the write operation will fail.

Not available on Neo4j Community Edition

```
CREATE CONSTRAINT constraint_name
FOR (p:Person)
REQUIRE p.name IS :: STRING
```

Create a node property type constraint on the label `Person` and property `name`, restricting the property to `STRING`. If a node with that label is created with a `name` property of a different Cypher type, the write operation will fail.

Not available on Neo4j Community Edition

```
CREATE CONSTRAINT constraint_name
FOR ()-[r:LIKED]-()
REQUIRE r.when IS :: DATE
```

Create a relationship property type constraint on the type `LIKED` and property `when`, restricting the property to `DATE`. If a relationship with that type is created with a `when` property of a different Cypher type, the write operation will fail.

Not available on Neo4j Community Edition

```
CREATE CONSTRAINT constraint_name
FOR (p:Person)
REQUIRE (p.name, p.surname) IS NODE KEY
```

Create a node key constraint on the label `Person` and properties `name` and `surname` with the name `constraint_name`. If a node with that label is created without both the `name` and `surname` properties, or if the combination of the two is not unique, or if the `name` and/or `surname` properties on an existing node with the label `Person` is modified to

violate these constraints, the write operation will fail.

Not available on AuraDB Business Critical Not available on Neo4j Community Edition

```
CREATE CONSTRAINT constraint_name
FOR ()-[r:KNOWS]-()
REQUIRE (r.since, r.isFriend) IS
RELATIONSHIP KEY
```

Create a relationship key constraint with the name `constraint_name` on the relationship type `KNOWS` and properties `since` and `isFriend`. If a relationship with that relationship type is created without both the `since` and `isFriend` properties, or if the combination of the two is not unique, the write operation will fail. The write operation will also fail if the `since` and/or `isFriend` properties on an existing relationship with the relationship type `KNOWS` is modified to violate these constraints.

Performance

Performance

Use parameters instead of literals when possible. This allows Neo4j DBMS to cache your queries instead of having to parse and build new execution plans.

Always set an upper limit for your variable length patterns. It is possible to have a query go wild and touch all nodes in a graph by mistake.

Return only the data you need. Avoid returning whole nodes and relationships; instead, pick the data you need and return only that.

Use `PROFILE / EXPLAIN` to analyze the performance of your queries. See Query Tuning for more information on these and other topics, such as planner hints.

Database Management

DATABASE Management

```
dba
`db1`
`database-name`
`database-name-123`
`database.name`
`database.name.123`
```

The naming rules for a database:

- The character length of a database name must be at least `3` characters; and not more than `63` characters.
- The first character of a database name must be an ASCII alphabetic character.
- Subsequent characters must be ASCII alphabetic or numeric characters, dots or dashes; `[a..z][0..9].-`.
- Database names are case-insensitive and normalized to lowercase.
- Database names that begin with an underscore (`_`) or with the prefix `system` are reserved for internal use.

Database names may include dots (`.`) without being quoted with backticks, although this behavior is deprecated as it may introduce ambiguity when addressing composite databases. Naming a database `foo.bar.baz` is valid, but deprecated. `` `foo.bar.baz` `` is valid.

```
SHOW DATABASES
```

List all databases in Neo4j DBMS and information about them, returns only the default outputs (`name`, `type`, `aliases`, `access`, `address`, `role`, `writer`, `requestedStatus`, `currentStatus`, `statusMessage`, `default`, `home`, and `constituents`).

```
SHOW DATABASES YIELD *
```

List all databases in Neo4j DBMS and information about them.

```
SHOW DATABASES
YIELD name, currentStatus
```

List information about databases, filtered by `name` and `currentStatus` and further refined by conditions on these.

```
WHERE name CONTAINS 'my'
  AND currentStatus = 'online'
```

```
SHOW DATABASE `database-name` YIELD *
```

List information about the database `database-name`.

```
SHOW DATABASES YIELD name,
defaultLanguage
```

List the default Cypher version of databases.

```
SHOW DEFAULT DATABASE
```

List information about the default database, for the Neo4j DBMS.

```
SHOW HOME DATABASE
```

List information about the current users home database.

Neo4j Enterprise Edition

```
DROP DATABASE `database-name` IF EXISTS
```

Delete the database `database-name`, if it exists. This command can delete both standard and composite databases.

Neo4j Enterprise Edition

```
DROP COMPOSITE DATABASE `composite-
database-name`
```

Delete the database named `composite-database-name`. In case the given database name does not exist or is not composite, and error will be thrown.

Neo4j Enterprise Edition

```
DROP DATABASE `database-name` CASCADE
ALIASES
```

Drop the database `database-name` and any database aliases referencing the database. This command can drop both standard and composite databases. For standard databases, the database aliases that will be

dropped are any local database aliases targeting the database. For composite databases, the database aliases that will be dropped are any constituent database aliases belonging to the composite database.

Neo4j Enterprise Edition

```
CREATE DATABASE `database-name` IF NOT
EXISTS
```

Create a standard database named `database-name` if it does not already exist.

Neo4j Enterprise Edition

```
CREATE OR REPLACE DATABASE `database-
name`
```

Create a standard database named `database-name`. If a database with that name exists, then the existing database is deleted and a new one created.

Neo4j Enterprise Edition

```
CREATE DATABASE `topology-example` IF NOT
EXISTS
TOPOLOGY 1 PRIMARY 0 SECONDARIES
```

Create a standard database named `topology-example` in a cluster environment, to use 1 primary server and 0 secondary servers.

Neo4j Enterprise Edition

```
CREATE COMPOSITE DATABASE `composite-
database-name`
```

Create a composite database named `composite-database-name`.

Neo4j Enterprise Edition

```
CREATE [COMPOSITE] DATABASE actors SET
DEFAULT LANGUAGE CYPHER 25
```

Set the default Cypher version for a standard or composite database when creating it. The available versions are

`CYPHER 25` and `CYPHER 5`. If not specified, the default language for the database is set to the default language of the DBMS.

Neo4j Enterprise Edition

```
STOP DATABASE `database-name`
```

Stop a database named `database-name`.

Neo4j Enterprise Edition

```
START DATABASE `database-name`
```

Start a database named `database-name`.

Neo4j Enterprise Edition

```
ALTER DATABASE `database-name` IF EXISTS
SET ACCESS READ ONLY
```

Modify a standard database named `database-name` to accept only read queries.

Not available on Neo4j Community Edition

```
ALTER DATABASE movies SET DEFAULT
LANGUAGE CYPHER 25
```

Alter the default Cypher version of an existing standard or composite database. The available versions are `CYPHER 25` and `CYPHER 5`.

Neo4j Enterprise Edition

```
ALTER DATABASE `database-name` IF EXISTS
SET ACCESS READ WRITE
```

Modify a standard database named `database-name` to accept write and read queries.

Neo4j Enterprise Edition

```
ALTER DATABASE `topology-example`
```

Modify a standard database named `topology-example` in a cluster

```
SET TOPOLOGY 1 PRIMARY 0 SECONDARIES
```

environment to use 1 primary server and 0 secondary servers.

Neo4j Enterprise Edition

```
ALTER DATABASE `topology-example`
SET TOPOLOGY 1 PRIMARY
SET ACCESS READ ONLY
```

Modify a standard database named `topology-example` in a cluster environment to use 1 primary servers and 0 secondary servers, and to only accept read queries.

ALIAS Management

AuraDB Business Critical AuraDB Virtual Dedicated Cloud Neo4j Enterprise Edition

`SHOW ALIASES FOR DATABASE`	List all database aliases in Neo4j DBMS and information about them, returns only the default outputs (`name` , `composite` , `database` , `location` , `url` , and `user`).
`SHOW ALIASES `database-alias` FOR DATABASE`	List the database alias named `database-alias` and the information about it. Returns only the default outputs (`name` , `composite` , `database` , `location` , `url` , and `user`).
`SHOW ALIASES FOR DATABASE YIELD *`	List all database aliases in Neo4j DBMS and information about them.
`SHOW ALIAS `remote-with-default-language` FOR DATABASE YIELD name, defaultLanguage`	Show the default Cypher version of a remote database alias.
`CREATE ALIAS `database-alias` IF NOT EXISTS FOR DATABASE `database-name``	Create a local alias named `database-alias` for the database named `database-name` .
`CREATE OR REPLACE ALIAS `database-alias` FOR DATABASE `database-name``	Create or replace a local alias named `database-alias` for the database named `database-name` .
`CREATE ALIAS `database-alias` FOR DATABASE `database-name` PROPERTIES { property = $value }`	Database aliases can be given properties.

```
CREATE ALIAS `database-alias`
FOR DATABASE `database-name`
AT $url
USER user_name
PASSWORD $password
```

Create a remote alias named `database-alias` for the database named `database-name`.

```
CREATE ALIAS `remote-with-default-
language`
FOR DATABASE `northwind-graph-2020`
    AT "neo4j+s://location:7687"
    USER alice
    PASSWORD 'example_secret'
    DEFAULT LANGUAGE CYPHER 25
```

Set the default Cypher version for a remote database alias when creating it. The available versions are `CYPHER 5` and `CYPHER 25`. Local database aliases and database aliases in composite databases cannot be assigned a default Cypher version. Local database aliases always have the Cypher version of their target database and database aliases in composite databases always have the Cypher version of the composite database they belong to.

```
CREATE ALIAS `composite-database-
name`.`alias-in-composite-name`
FOR DATABASE `database-name`
AT $url
USER user_name
PASSWORD $password
```

Create a remote alias named `alias-in-composite-name` as a constituent alias in the composite database named `composite-database-name` for the database with name `database-name`.

```
ALTER ALIAS `database-alias` IF EXISTS
SET DATABASE TARGET `database-name`
```

Alter the alias named `database-alias` to target the database named `database-name`.

```
ALTER ALIAS `remote-database-alias` IF
EXISTS
SET DATABASE
USER user_name
PASSWORD $password
```

Alter the remote alias named `remote-database-alias`, set the username (`user_name`) and the password.

```
ALTER ALIAS `database-alias`
```

Update the properties for the database alias named `database-alias`.

```
SET DATABASE PROPERTIES { key: value }
```

```
ALTER ALIAS `remote-with-default-language` SET DATABASE DEFAULT LANGUAGE CYPHER 25
```

Alter the default Cypher version of a remote database alias. The available versions are `CYPHER 25` and `CYPHER 5`. It is not possible to alter the default Cypher version of a local database alias or an alias belonging to a composite database. Local database aliases always have the Cypher version of their target database and aliases belonging to composite databases always have the Cypher version of the composite database.

```
DROP ALIAS `database-alias` IF EXISTS FOR DATABASE
```

Delete the alias named `database-alias`.

SERVER Management

AuraDB Business Critical AuraDB Virtual Dedicated Cloud

```
SHOW SERVERS
```

Display all servers running in the cluster, including servers that have yet to be enabled as well as dropped servers. Default outputs are: `name`, `address`, `state`, `health`, and `hosting`.

Neo4j Enterprise Edition

```
ENABLE SERVER 'serverId'
```

Make the server with the ID `serverId` an active member of the cluster.

AuraDB Business Critical AuraDB Virtual Dedicated Cloud Neo4j Enterprise Edition

```
RENAME SERVER 'oldName' TO 'newName'
```

Change the name of a server.

Neo4j Enterprise Edition

```
ALTER SERVER 'name' SET OPTIONS
{modeConstraint: 'PRIMARY'}
```

Only allow the specified server to host databases in primary mode.

Neo4j Enterprise Edition

```
REALLOCATE DATABASES
```

Re-balance databases among the servers in the cluster.

Neo4j Enterprise Edition

```
DEALLOCATE DATABASES FROM SERVER 'name'
```

Remove all databases from the specified server, adding them to other servers as

needed. The specified server is not allowed to host any new databases.

Neo4j Enterprise Edition

```
DROP SERVER 'name'
```

Remove the specified server from the cluster.

Access Control

USER Management

`SHOW USERS`	List all users in Neo4j DBMS, returns only the default outputs (`user` , `roles` , `passwordChangeRequired` , `suspended` , and `home`).
`SHOW CURRENT USER`	List the currently logged-in user, returns only the default outputs (`user` , `roles` , `passwordChangeRequired` , `suspended` , and `home`).
`SHOW USERS` `WHERE suspended = true`	Not available on Neo4j Community Edition List users that are suspended.
`SHOW USERS` `WHERE passwordChangeRequired`	List users that must change their password at the next login.
`SHOW USERS WITH AUTH`	List users with their auth providers. Will return one row per user per auth provider.
`SHOW USERS WITH AUTH WHERE provider = 'oidc1'`	List users who have the `oidc1` auth provider.
`DROP USER user_name`	Delete the specified user.
`CREATE USER user_name` `SET PASSWORD $password`	Create a new user and set the password. This password must be changed on the first login.

```
CREATE USER user_name
SET AUTH 'native' {
  SET PASSWORD $password
  SET PASSWORD CHANGE REQUIRED
}
```

Create a new user and set the password using the auth provider syntax. This password must be changed on the first login.

```
RENAME USER user_name TO other_user_name
```

Rename the specified user.

```
ALTER CURRENT USER
SET PASSWORD FROM $oldPassword TO
$newPassword
```

Change the password of the logged-in user. The user will not be required to change this password on the next login.

```
ALTER USER user_name
SET PASSWORD $password
CHANGE NOT REQUIRED
```

Set a new password (a String) for a user. This user will not be required to change this password on the next login.

```
ALTER USER user_name IF EXISTS
SET PASSWORD CHANGE REQUIRED
```

If the specified user exists, force this user to change the password on the next login.

Neo4j Enterprise Edition

```
ALTER USER user_name
SET AUTH 'externalProviderName' {
  SET ID 'userIdForExternalProvider'
}
```

Add another way for the user to authenticate and authorize using the external provider `externalProviderName`. This provider needs to be defined in the configurations settings.

Not available on Neo4j Community Edition

```
ALTER USER user_name
SET STATUS SUSPENDED
```

Change the status to `SUSPENDED`, for the specified user.

Not available on Neo4j Community Edition

```
ALTER USER user_name
SET STATUS ACTIVE
```

Change the status to `ACTIVE`, for the specified user.

AuraDB Business Critical AuraDB Virtual Dedicated Cloud Neo4j Enterprise Edition

```
ALTER USER user_name
SET HOME DATABASE `database-name`
```

Set the home database for the specified user. The home database can either be a database or an alias.

AuraDB Business Critical AuraDB Virtual Dedicated Cloud Neo4j Enterprise Edition

```
ALTER USER user_name
REMOVE HOME DATABASE
```

Unset the home database for the specified user and fallback to the default database.

ROLE Management

AuraDB Business Critical | AuraDB Virtual Dedicated Cloud | Neo4j Enterprise Edition

`SHOW ROLES`	List all roles in the system, returns the output `role`.
`SHOW ROLES` `WHERE role CONTAINS $subString`	List roles that contains a given string.
`SHOW POPULATED ROLES`	List all roles that are assigned to at least one user in the system.
`SHOW POPULATED ROLES WITH USERS`	List all roles that are assigned to at least one user in the system, and the users assigned to those roles. The returned outputs are `role` and `member`.

`SHOW POPULATED ROLES WITH USERS`
`YIELD member, role`
`WHERE member = $user`
`RETURN role`
	List all roles that are assigned to a `$user`.

`DROP ROLE role_name`	Delete a role.
`CREATE ROLE role_name IF NOT EXISTS`	Create a role, unless it already exists.
`CREATE ROLE role_name AS COPY OF other_role_name`	Create a role, as a copy of the existing `other_role_name`.

`RENAME ROLE role_name TO other_role_name`	Rename a role.
`GRANT ROLE role_name1, role_name2 TO user_name`	Assign roles to a user.
`REVOKE ROLE role_name FROM user_name`	Remove the specified role from a user.

SHOW Privileges

AuraDB Business Critical | AuraDB Virtual Dedicated Cloud | Neo4j Enterprise Edition

`SHOW PRIVILEGES`	List all privileges in the system, and the roles that they are assigned to. Outputs returned are: `access`, `action`, `resource`, `graph`, `segment`, `role`, and `immutable`.
`SHOW PRIVILEGES AS COMMANDS`	List all privileges in the system as Cypher commands, for example `GRANT ACCESS ON DATABASE * TO \`admin\``. Returns only the default output (`command`).
`SHOW USER PRIVILEGES`	List all privileges of the currently logged-in user, and the roles that they are assigned to. Outputs returned are: `access`, `action`, `resource`, `graph`, `segment`, `role`, `immutable`, and `user`.
`SHOW USER PRIVILEGES AS COMMANDS`	List all privileges of the currently logged-in user, and the roles that they are assigned to as Cypher commands, for example `GRANT ACCESS ON DATABASE * TO $role`. Returns only the default output (`command`).
`SHOW USER user_name PRIVILEGES`	List all privileges assigned to each of the specified users (multiple users can be specified separated by commas `n1, n2, n3`), and the roles that they are assigned to. Outputs returned are: `access`, `action`, `resource`, `graph`, `segment`, `role`, `immutable`, and `user`.
`SHOW USER user_name PRIVILEGES AS`	List all privileges assigned to each of the specified users (multiple users can be

`COMMANDS YIELD *`	specified separated by commas `n1, n2, n3`), as generic Cypher commands, for example `GRANT ACCESS ON DATABASE * TO $role` . Outputs returned are: `command` and `immutable` .
`SHOW ROLE role_name PRIVILEGES`	List all privileges assigned to each of the specified roles (multiple roles can be specified separated by commas `r1, r2, r3`). Outputs returned are: `access` , `action` , `resource` , `graph` , `segment` , `role` , and `immutable` .
`SHOW ROLE role_name PRIVILEGES AS COMMANDS`	List all privileges assigned to each of the specified roles (multiple roles can be specified separated by commas `r1, r2, r3`) as Cypher commands, for example `GRANT ACCESS ON DATABASE * TO` `` `admin` `` . Returns only the default output (`command`).

SHOW SUPPORTED Privileges

AuraDB Business Critical AuraDB Virtual Dedicated Cloud Neo4j Enterprise Edition

`SHOW SUPPORTED PRIVILEGES`	List all privileges that are possible to grant or deny on a server. Outputs returned are: `action` , `qualifier` , `target` , `scope` , and `description` .

IMMUTABLE Privileges

Neo4j Enterprise Edition

```
GRANT IMMUTABLE TRAVERSE
ON GRAPH * TO role_name
```

Grant immutable `TRAVERSE` privilege on all graphs to the specified role.

```
DENY IMMUTABLE START
ON DATABASE * TO role_name
```

Deny immutable `START` privilege to start all databases to the specified role.

```
REVOKE IMMUTABLE CREATE ROLE
ON DBMS FROM role_name
```

Revoke immutable `CREATE ROLE` privilege from the specified role. When immutable is specified in conjunction with a `REVOKE` command, it will act as a filter and only remove the matching immutable privileges.

Load Privileges

AuraDB Business Critical AuraDB Virtual Dedicated Cloud Neo4j Enterprise Edition

```
GRANT LOAD
ON ALL DATA
TO role_name
```

Grant `LOAD` privilege on `ALL DATA` to allow loading all data to the specified role.

```
DENY LOAD
ON CIDR "127.0.0.1/32"
TO role_name
```

Deny `LOAD` privilege on CIDR range `127.0.0.1/32` to disallow loading data from sources in that range to the specified role.

ON GRAPH

ON GRAPH Read Privileges

AuraDB Business Critical | AuraDB Virtual Dedicated Cloud | Neo4j Enterprise Edition

```
GRANT TRAVERSE
ON GRAPH * NODE * TO role_name
```

Grant `TRAVERSE` privilege on all graphs and all nodes to the specified role.

- `GRANT` – gives privileges to roles.
- `DENY` – denies privileges to roles.

```
REVOKE GRANT TRAVERSE
ON GRAPH * NODE * FROM role_name
```

To remove a granted or denied privilege, prepend the privilege query with `REVOKE` and replace the `TO` with `FROM`.

```
GRANT TRAVERSE
ON GRAPH * RELATIONSHIP * TO role_name
```

Grant `TRAVERSE` privilege on all graphs and all relationships to the specified role.

```
DENY READ {prop}
ON GRAPH `database-name` RELATIONSHIP
rel_type TO role_name
```

Deny `READ` privilege on a specified property, on all relationships with a specified type in a specified graph, to the specified role.

```
REVOKE READ {prop}
ON GRAPH `database-name` FROM role_name
```

Revoke `READ` privilege on a specified property in a specified graph from the specified role.

```
GRANT MATCH {*}
ON HOME GRAPH ELEMENTS label_or_type TO
role_name
```

Grant `MATCH` privilege on all nodes and relationships with the specified label/type, on the home graph, to the specified role. This is semantically the same as having both `TRAVERSE` privilege and `READ {*}` privilege.

```
GRANT READ {*}
ON GRAPH *
```

Grant `READ` privilege on all graphs and all nodes with a `secret` property set to

```
FOR (n) WHERE n.secret = false
TO role_name
```

`false` to the specified role.

```
DENY TRAVERSE
ON GRAPH *
FOR (n:label) WHERE n.secret <> false
TO role_name
```

Deny `TRAVERSE` privilege on all graphs and all nodes with the specified label and with a `secret` property not set to `false` to the specified role.

```
REVOKE MATCH {*}
ON GRAPH *
FOR (n:foo_label|bar_label) WHERE
n.secret IS NULL
FROM role_name
```

Revoke `MATCH` privilege on all graphs and all nodes with either `foo_label` or `bar_label` and with a `secret` property that is `null` from the specified role.

ON GRAPH Write Privileges

AuraDB Business Critical | AuraDB Virtual Dedicated Cloud | Neo4j Enterprise Edition

```
GRANT ALL GRAPH PRIVILEGES
ON GRAPH `database-name` TO role_name
```

Grant `ALL GRAPH PRIVILEGES` privilege on a specified graph to the specified role.

```
GRANT ALL ON GRAPH `database-name` TO
role_name
```

Short form for grant `ALL GRAPH PRIVILEGES` privilege.

- `GRANT` – gives privileges to roles.
- `DENY` – denies privileges to roles.

To remove a granted or denied privilege, prepend the privilege query with `REVOKE` and replace the `TO` with `FROM`; (`REVOKE GRANT ALL ON GRAPH` `` `database-name` FROM role_name ``).

```
DENY CREATE
ON GRAPH * NODES node_label TO role_name
```

Deny `CREATE` privilege on all nodes with a specified label in all graphs to the specified role.

```
REVOKE DELETE
ON GRAPH `database-name` TO role_name
```

Revoke `DELETE` privilege on all nodes and relationships in a specified graph from the specified role.

```
GRANT SET LABEL node_label
ON GRAPH * TO role_name
```

Grant `SET LABEL` privilege for the specified label on all graphs to the specified role.

```
DENY REMOVE LABEL *
ON GRAPH `database-name` TO role_name
```

Deny `REMOVE LABEL` privilege for all labels on a specified graph to the specified role.

`GRANT SET PROPERTY {prop_name} ON GRAPH `database-name` RELATIONSHIPS rel_type TO role_name`	Grant `SET PROPERTY` privilege on a specified property, on all relationships with a specified type in a specified graph, to the specified role.
`GRANT MERGE {*} ON GRAPH * NODES node_label TO role_name`	Grant `MERGE` privilege on all properties, on all nodes with a specified label in all graphs, to the specified role.
`REVOKE WRITE ON GRAPH * FROM role_name`	Revoke `WRITE` privilege on all graphs from the specified role.

ON DATABASE

ON DATABASE Privileges

AuraDB Business Critical AuraDB Virtual Dedicated Cloud Neo4j Enterprise Edition

```
GRANT ALL DATABASE PRIVILEGES
ON DATABASE * TO role_name
```

Grant `ALL DATABASE PRIVILEGES` privilege for all databases to the specified role.

- Allows access (`GRANT ACCESS`).
- Index management (`GRANT INDEX MANAGEMENT`).
- Constraint management (`GRANT CONSTRAINT MANAGEMENT`).
- Name management (`GRANT NAME MANAGEMENT`).

Note that the privileges for starting and stopping all databases, and transaction management, are not included.

```
GRANT ALL ON DATABASE * TO role_name
```

Short form for grant `ALL DATABASE PRIVILEGES` privilege.

- `GRANT` –gives privileges to roles.
- `DENY` –denies privileges to roles.

To remove a granted or denied privilege, prepend the privilege query with `REVOKE` and replace the `TO` with `FROM` ; (`REVOKE GRANT ALL ON DATABASE * FROM role_name`).

```
REVOKE ACCESS
ON HOME DATABASE FROM role_name
```

Revoke `ACCESS` privilege to access and run queries against the home database from the specified role.

```
GRANT START
ON DATABASE * TO role_name
```

Grant `START` privilege to start all databases to the specified role.

```
DENY STOP
ON HOME DATABASE TO role_name
```

Deny `STOP` privilege to stop the home database to the specified role.

ON DATABASE - INDEX MANAGEMENT Privileges

AuraDB Business Critical AuraDB Virtual Dedicated Cloud Neo4j Enterprise Edition

```
GRANT INDEX MANAGEMENT
ON DATABASE * TO role_name
```

Grant `INDEX MANAGEMENT` privilege to create, drop, and list indexes for all database to the specified role.

- Allow creating an index - (`GRANT CREATE INDEX`).
- Allow removing an index - (`GRANT DROP INDEX`).
- Allow listing an index - (`GRANT SHOW INDEX`).

```
GRANT CREATE INDEX
ON DATABASE `database-name` TO role_name
```

Grant `CREATE INDEX` privilege to create indexes on a specified database to the specified role.

```
GRANT DROP INDEX
ON DATABASE `database-name` TO role_name
```

Grant `DROP INDEX` privilege to drop indexes on a specified database to the specified role.

```
GRANT SHOW INDEX
ON DATABASE * TO role_name
```

Grant `SHOW INDEX` privilege to list indexes on all databases to the specified role.

ON DATABASE - CONSTRAINT MANAGEMENT Privileges

AuraDB Business Critical | AuraDB Virtual Dedicated Cloud | Neo4j Enterprise Edition

```
GRANT CONSTRAINT MANAGEMENT
ON DATABASE * TO role_name
```

Grant `CONSTRAINT MANAGEMENT` privilege to create, drop, and list constraints for all database to the specified role.

- Allow creating a constraint - (`GRANT CREATE CONSTRAINT`).
- Allow removing a constraint - (`GRANT DROP CONSTRAINT`).
- Allow listing a constraint - (`GRANT SHOW CONSTRAINT`).

```
GRANT CREATE CONSTRAINT
ON DATABASE * TO role_name
```

Grant `CREATE CONSTRAINT` privilege to create constraints on all databases to the specified role.

```
GRANT DROP CONSTRAINT
ON DATABASE * TO role_name
```

Grant `DROP CONSTRAINT` privilege to create constraints on all databases to the specified role.

```
GRANT SHOW CONSTRAINT
ON DATABASE `database-name` TO role_name
```

Grant `SHOW CONSTRAINT` privilege to list constraints on a specified database to the specified role.

ON DATABASE - NAME MANAGEMENT Privileges

AuraDB Business Critical | AuraDB Virtual Dedicated Cloud | Neo4j Enterprise Edition

```
GRANT NAME MANAGEMENT
ON DATABASE * TO role_name
```

Grant `NAME MANAGEMENT` privilege to create new labels, new relationship types, and new property names for all databases to the specified role.

- Allow creating a new label - (`GRANT CREATE NEW LABEL`).
- Allow creating a new relationship type - (`GRANT CREATE NEW TYPE`).
- Allow creating a new property name - (`GRANT CREATE NEW NAME`).

```
GRANT CREATE NEW LABEL
ON DATABASE * TO role_name
```

Grant `CREATE NEW LABEL` privilege to create new labels on all databases to the specified role.

```
DENY CREATE NEW TYPE
ON DATABASE * TO role_name
```

Deny `CREATE NEW TYPE` privilege to create new relationship types on all databases to the specified role.

```
GRANT CREATE NEW NAME
ON DATABASE * TO role_name
```

Grant `CREATE NEW NAME` privilege to create new property names on all databases to the specified role.

ON DATABASE - TRANSACTION MANAGEMENT Privileges

AuraDB Business Critical | AuraDB Virtual Dedicated Cloud | Neo4j Enterprise Edition

```
GRANT TRANSACTION MANAGEMENT (*)
ON DATABASE * TO role_name
```

Grant `TRANSACTION MANAGEMENT` privilege to show and terminate transactions on all users, for all databases, to the specified role.

- Allow listing transactions - (`GRANT SHOW TRANSACTION`).
- Allow terminate transactions - (`GRANT TERMINATE TRANSACTION`).

```
GRANT SHOW TRANSACTION (*)
ON DATABASE * TO role_name
```

Grant `SHOW TRANSACTION` privilege to list transactions on all users on all databases to the specified role.

```
GRANT SHOW TRANSACTION (user_name1,
user_name2)
ON HOME DATABASE TO role_name1,
role_name2
```

Grant `SHOW TRANSACTION` privilege to list transactions by the specified users on home database to the specified roles.

```
GRANT TERMINATE TRANSACTION (*)
ON DATABASE * TO role_name
```

Grant `TERMINATE TRANSACTION` privilege to terminate transactions on all users on all databases to the specified role.

ON DBMS

ON DBMS Privileges

AuraDB Business Critical AuraDB Virtual Dedicated Cloud Neo4j Enterprise Edition

```
GRANT ALL DBMS PRIVILEGES
ON DBMS TO role_name
```

Grant `ALL DBMS PRIVILEGES` privilege to perform management for roles, users, databases, aliases, and privileges to the specified role. Also privileges to execute procedures and user defined functions are granted.

- Allow controlling roles - (`GRANT ROLE MANAGEMENT`).
- Allow controlling users - (`GRANT USER MANAGEMENT`).
- Allow controlling databases - (`GRANT DATABASE MANAGEMENT`).
- Allow controlling aliases - (`GRANT ALIAS MANAGEMENT`).
- Allow controlling privileges - (`GRANT PRIVILEGE MANAGEMENT`).
- Allow user impersonation - (`GRANT IMPERSONATE (*)`).
- Allow to execute all procedures with elevated privileges.
- Allow to execute all user defined functions with elevated privileges.

```
GRANT ALL
ON DBMS TO role_name
```

Short form for grant `ALL DBMS PRIVILEGES` privilege.

- `GRANT` – gives privileges to roles.
- `DENY` – denies privileges to roles.

To remove a granted or denied privilege, prepend the privilege query with `REVOKE` and replace the `TO` with `FROM` ; (`REVOKE GRANT ALL ON DBMS FROM role_name`).

```
DENY IMPERSONATE (user_name1, user_name2)
ON DBMS TO role_name
```

Deny `IMPERSONATE` privilege to impersonate the specified users

	(`user_name1` and `user_name2`) to the specified role.
`REVOKE IMPERSONATE (*)` `ON DBMS TO role_name`	Revoke `IMPERSONATE` privilege to impersonate all users from the specified role.
`GRANT EXECUTE PROCEDURE *` `ON DBMS TO role_name`	Enables the specified role to execute all procedures.
`GRANT EXECUTE BOOSTED PROCEDURE *` `ON DBMS TO role_name`	Enables the specified role to use elevated privileges when executing all procedures.
`GRANT EXECUTE ADMIN PROCEDURES` `ON DBMS TO role_name`	Enables the specified role to execute procedures annotated with `@Admin` . The procedures are executed with elevated privileges.
`GRANT EXECUTE FUNCTIONS *` `ON DBMS TO role_name`	Enables the specified role to execute all user defined functions.
`GRANT EXECUTE BOOSTED FUNCTIONS *` `ON DBMS TO role_name`	Enables the specified role to use elevated privileges when executing all user defined functions.
`GRANT SHOW SETTINGS *` `ON DBMS TO role_name`	Enables the specified role to view all configuration settings.

ON DBMS - ROLE MANAGEMENT Privileges

AuraDB Business Critical | AuraDB Virtual Dedicated Cloud | Neo4j Enterprise Edition

```
GRANT ROLE MANAGEMENT
ON DBMS TO role_name
```

Grant `ROLE MANAGEMENT` privilege to manage roles to the specified role.

- Allow creating roles - (`GRANT CREATE ROLE`).
- Allow renaming roles - (`GRANT RENAME ROLE`).
- Allow deleting roles - (`GRANT DROP ROLE`).
- Allow assigning (`GRANT`) roles to a user - (`GRANT ASSIGN ROLE`).
- Allow removing (`REVOKE`) roles from a user - (`GRANT REMOVE ROLE`).
- Allow listing roles - (`GRANT SHOW ROLE`).

```
GRANT CREATE ROLE
ON DBMS TO role_name
```

Grant `CREATE ROLE` privilege to create roles to the specified role.

```
GRANT RENAME ROLE
ON DBMS TO role_name
```

Grant `RENAME ROLE` privilege to rename roles to the specified role.

```
DENY DROP ROLE
ON DBMS TO role_name
```

Deny `DROP ROLE` privilege to delete roles to the specified role.

```
GRANT ASSIGN ROLE
ON DBMS TO role_name
```

Grant `ASSIGN ROLE` privilege to assign roles to users to the specified role.

```
DENY REMOVE ROLE
ON DBMS TO role_name
```

Deny `REMOVE ROLE` privilege to remove roles from users to the specified role.

```
GRANT SHOW ROLE
ON DBMS TO role_name
```

Grant `SHOW ROLE` privilege to list roles to the specified role.

ON DBMS - USER MANAGEMENT Privileges

AuraDB Business Critical | AuraDB Virtual Dedicated Cloud | Neo4j Enterprise Edition

```
GRANT USER MANAGEMENT
ON DBMS TO role_name
```

Grant `USER MANAGEMENT` privilege to manage users to the specified role.

- Allow creating users - (`GRANT CREATE USER`).
- Allow renaming users - (`GRANT RENAME USER`).
- Allow modifying a user - (`GRANT ALTER USER`).
- Allow deleting users - (`GRANT DROP USER`).
- Allow listing users - (`GRANT SHOW USER`).

```
DENY CREATE USER
ON DBMS TO role_name
```

Deny `CREATE USER` privilege to create users to the specified role.

```
GRANT RENAME USER
ON DBMS TO role_name
```

Grant `RENAME USER` privilege to rename users to the specified role.

```
GRANT ALTER USER
ON DBMS TO my_role
```

Grant `ALTER USER` privilege to alter users to the specified role.

- Allow changing a user's password - (`GRANT SET PASSWORD`).
- Allow adding or removing a user's auth providers - (`GRANT SET AUTH`).
- Allow changing a user's home database - (`GRANT SET USER HOME DATABASE`).
- Allow changing a user's status - (`GRANT USER STATUS`).

```
DENY SET PASSWORD
ON DBMS TO role_name
```

Deny `SET PASSWORD` privilege to alter a user password to the specified role.

```
GRANT SET AUTH
ON DBMS TO role_name
```

Grant `SET AUTH` privilege to add/remove auth providers to the specified role.

```
GRANT SET USER HOME DATABASE
ON DBMS TO role_name
```

Grant `SET USER HOME DATABASE` privilege to alter the home database of users to the specified role.

```
GRANT SET USER STATUS
ON DBMS TO role_name
```

Grant `SET USER STATUS` privilege to alter user account status to the specified role.

```
GRANT DROP USER
ON DBMS TO role_name
```

Grant `DROP USER` privilege to delete users to the specified role.

```
DENY SHOW USER
ON DBMS TO role_name
```

Deny `SHOW USER` privilege to list users to the specified role.

ON DBMS - DATABASE MANAGEMENT Privileges

AuraDB Business Critical | AuraDB Virtual Dedicated Cloud | Neo4j Enterprise Edition

```
GRANT DATABASE MANAGEMENT
ON DBMS TO role_name
```

Grant `DATABASE MANAGEMENT` privilege to manage databases to the specified role.

- Allow creating standard databases - (`GRANT CREATE DATABASE`).
- Allow deleting standard databases - (`GRANT DROP DATABASE`).
- Allow modifying standard databases - (`GRANT ALTER DATABASE`).
- Allow managing composite databases - (`GRANT COMPOSITE DATABASE MANAGEMENT`).

```
GRANT CREATE DATABASE
ON DBMS TO role_name
```

Grant `CREATE DATABASE` privilege to create standard databases to the specified role.

```
GRANT DROP DATABASE
ON DBMS TO role_name
```

Grant `DROP DATABASE` privilege to delete standard databases to the specified role.

```
GRANT ALTER DATABASE
ON DBMS TO role_name
```

Grant `ALTER DATABASE` privilege to alter standard databases the specified role.

- Allow modifying access mode for standard databases - (`GRANT SET DATABASE ACCESS`).
- Allow modifying topology settings for standard databases.

```
GRANT SET DATABASE ACCESS
ON DBMS TO role_name
```

Grant `SET DATABASE ACCESS` privilege to set database access mode for standard databases to the specified role.

```
GRANT COMPOSITE DATABASE MANAGEMENT
```

Grant all privileges to manage composite databases to the specified role.

`ON DBMS TO role_name`	◦ Allow creating composite databases - (`CREATE COMPOSITE DATABASE`). ◦ Allow deleting composite databases - (`DROP COMPOSITE DATABASE`).
`DENY CREATE COMPOSITE DATABASE ON DBMS TO role_name`	Denies the specified role the privilege to create composite databases.
`REVOKE DROP COMPOSITE DATABASE ON DBMS FROM role_name`	Revokes the granted and denied privileges to delete composite databases from the specified role.
`GRANT SERVER MANAGEMENT ON DBMS TO role_name`	Enables the specified role to show, enable, rename, alter, reallocate, deallocate, and drop servers.
`DENY SHOW SERVERS ON DBMS TO role_name`	Denies the specified role the privilege to show information about the serves.

ON DBMS - ALIAS MANAGEMENT Privileges

AuraDB Business Critical | AuraDB Virtual Dedicated Cloud | Neo4j Enterprise Edition

```
GRANT ALIAS MANAGEMENT
ON DBMS TO role_name
```

Grant `ALIAS MANAGEMENT` privilege to manage aliases to the specified role.

- Allow creating aliases - (`GRANT CREATE ALIAS`).
- Allow deleting aliases - (`GRANT DROP ALIAS`).
- Allow modifying aliases - (`GRANT ALTER ALIAS`).
- Allow listing aliases - (`GRANT SHOW ALIAS`).

```
GRANT CREATE ALIAS
ON DBMS TO role_name
```

Grant `CREATE ALIAS` privilege to create aliases to the specified role.

```
GRANT DROP ALIAS
ON DBMS TO role_name
```

Grant `DROP ALIAS` privilege to delete aliases to the specified role.

```
GRANT ALTER ALIAS
ON DBMS TO role_name
```

Grant `ALTER ALIAS` privilege to alter aliases to the specified role.

```
GRANT SHOW ALIAS
ON DBMS TO role_name
```

Grant `SHOW ALIAS` privilege to list aliases to the specified role.

ON DBMS - ROLE MANAGEMENT Privileges

AuraDB Business Critical | AuraDB Virtual Dedicated Cloud | Neo4j Enterprise Edition

```
GRANT ROLE MANAGEMENT
ON DBMS TO role_name
```

Grant `ROLE MANAGEMENT` privilege to manage roles to the specified role.

- Allow creating roles - (`GRANT CREATE ROLE`).
- Allow renaming roles - (`GRANT RENAME ROLE`).
- Allow deleting roles - (`GRANT DROP ROLE`).
- Allow assigning (`GRANT`) roles to a user - (`GRANT ASSIGN ROLE`).
- Allow removing (`REVOKE`) roles from a user - (`GRANT REMOVE ROLE`).
- Allow listing roles - (`GRANT SHOW ROLE`).

```
GRANT CREATE ROLE
ON DBMS TO role_name
```

Grant `CREATE ROLE` privilege to create roles to the specified role.

```
GRANT RENAME ROLE
ON DBMS TO role_name
```

Grant `RENAME ROLE` privilege to rename roles to the specified role.

```
DENY DROP ROLE
ON DBMS TO role_name
```

Deny `DROP ROLE` privilege to delete roles to the specified role.

```
GRANT ASSIGN ROLE
ON DBMS TO role_name
```

Grant `ASSIGN ROLE` privilege to assign roles to users to the specified role.

```
DENY REMOVE ROLE
ON DBMS TO role_name
```

Deny `REMOVE ROLE` privilege to remove roles from users to the specified role.

```
GRANT SHOW ROLE
ON DBMS TO role_name
```

Grant `SHOW ROLE` privilege to list roles to the specified role.

ON DBMS - PRIVILEGE MANAGEMENT Privileges

AuraDB Business Critical AuraDB Virtual Dedicated Cloud Neo4j Enterprise Edition

```
GRANT PRIVILEGE MANAGEMENT
ON DBMS TO role_name
```

Grant `PRIVILEGE MANAGEMENT` privilege to manage privileges for the Neo4j DBMS to the specified role.

- Allow assigning (`GRANT` | `DENY`) privileges for a role - (`GRANT ASSIGN PRIVILEGE`).
- Allow removing (`REVOKE`) privileges for a role - (`GRANT REMOVE PRIVILEGE`).
- Allow listing privileges - (`GRANT SHOW PRIVILEGE`).

```
GRANT ASSIGN PRIVILEGE
ON DBMS TO role_name
```

Grant `ASSIGN PRIVILEGE` privilege, allows the specified role to assign privileges for roles.

```
GRANT REMOVE PRIVILEGE
ON DBMS TO role_name
```

Grant `REMOVE PRIVILEGE` privilege, allows the specified role to remove privileges for roles.

```
GRANT SHOW PRIVILEGE
ON DBMS TO role_name
```

Grant `SHOW PRIVILEGE` privilege to list privileges to the specified role.