

Cypher is the declarative query language for Neo4j, the world's leading graph database.

Key principles and capabilities of Cypher are as follows:

- Cypher matches patterns of nodes and relationship in the graph, to extract information or modify the data.
- Cypher has the concept of identifiers which denote named, bound elements and parameters.
- Cypher can create, update, and remove nodes, relationships, labels, and properties.
- Cypher manages indexes and constraints.

You can try Cypher snippets live in the Neo4j Console at console.neo4j.org or read the full Cypher documentation in the [Neo4j Manual](#). For live graph models using Cypher check out [GraphGist](#).

The Cypher Refcard is also [available in PDF format](#).

Note: `{value}` denotes either literals, for ad hoc Cypher queries; or parameters, which is the best practice for applications. Neo4j properties can be strings, numbers, booleans or arrays thereof. Cypher also supports maps and collections.

Syntax

Read Query Structure
<pre>[MATCH WHERE] [OPTIONAL MATCH WHERE] [WITH [ORDER BY] [SKIP] [LIMIT]] RETURN [ORDER BY] [SKIP] [LIMIT]</pre>

MATCH
<pre>MATCH (n:Person)-[:KNOWS]->(m:Person) WHERE n.name = "Alice"</pre> <p>Node patterns can contain labels and properties.</p> <pre>MATCH (n)-->(m)</pre> <p>Any pattern can be used in MATCH.</p> <pre>MATCH (n {name: "Alice"})-->(m)</pre> <p>Patterns with node properties.</p> <pre>MATCH p = (n)-->(m)</pre> <p>Assign a path to p.</p> <pre>OPTIONAL MATCH (n)-[r]->(m)</pre> <p>Optional pattern, NULLS will be used for missing parts.</p> <pre>WHERE m.name = "Alice"</pre> <p>Force the planner to use a label scan to solve the query (for manual performance tuning).</p>

WHERE
<pre>WHERE n.property <> {value}</pre> <p>Use a predicate to filter. Note that WHERE is always part of a MATCH, OPTIONAL MATCH, WITH or START clause. Putting it after a different clause in a query will alter what it does.</p>

Write-Only Query Structure
<pre>(CREATE [UNIQUE] MERGE)* [SET DELETE REMOVE FOREACH]* [RETURN [ORDER BY] [SKIP] [LIMIT]]</pre>

Read-Write Query Structure
<pre>[MATCH WHERE] [OPTIONAL MATCH WHERE] [WITH [ORDER BY] [SKIP] [LIMIT]] (CREATE [UNIQUE] MERGE)* [SET DELETE REMOVE FOREACH]* [RETURN [ORDER BY] [SKIP] [LIMIT]]</pre>

CREATE
<pre>CREATE (n {name: {value}})</pre> <p>Create a node with the given properties.</p>
<pre>CREATE (n {map})</pre> <p>Create a node with the given properties.</p>
<pre>CREATE (n {collectionOfMaps})</pre> <p>Create nodes with the given properties.</p>
<pre>CREATE (n)-[r:KNOWS]->(m)</pre> <p>Create a relationship with the given type and direction; bind an identifier to it.</p>
<pre>CREATE (n)-[:LOVES {since: {value}}]->(m)</pre> <p>Create a relationship with the given type, direction, and properties.</p>

MERGE
<pre>MERGE (n:Person {name: {value}}) ON CREATE SET n.created = timestamp() ON MATCH SET n.counter = coalesce(n.counter, 0) + 1, n.accessTime = timestamp()</pre> <p>Match pattern or create it if it does not exist. Use ON CREATE and ON MATCH for conditional updates.</p>
<pre>MATCH (a:Person {name: {value1}}), (b:Person {name: {value2}}) MERGE (a)-[r:LOVES]->(b)</pre> <p>MERGE finds or creates a relationship between the nodes.</p>
<pre>MATCH (a:Person {name: {value1}}) MERGE (a)-[r:KNOWS]->(b:Person {name: {value3}})</pre> <p>MERGE finds or creates subgraphs attached to the node.</p>

RETURN
<pre>RETURN *</pre> <p>Return the value of all identifiers.</p>
<pre>RETURN n AS columnName</pre> <p>Use alias for result column name.</p>
<pre>RETURN DISTINCT n</pre> <p>Return unique rows.</p>
<pre>ORDER BY n.property</pre> <p>Sort the result.</p>
<pre>ORDER BY n.property DESC</pre> <p>Sort the result in descending order.</p>
<pre>SKIP {skipNumber}</pre> <p>Skip a number of results.</p>
<pre>LIMIT {limitNumber}</pre> <p>Limit the number of results.</p>
<pre>SKIP {skipNumber} LIMIT {limitNumber}</pre> <p>Skip results at the top and limit the number of results.</p>
<pre>RETURN count(*)</pre> <p>The number of matching rows. See Aggregation for more.</p>

WITH
<pre>MATCH (user)-[:FRIEND]-(friend) WHERE user.name = {name} WITH user, count(friend) AS friends WHERE friends > 10 RETURN user</pre> <p>The WITH syntax is similar to RETURN. It separates query parts explicitly, allowing you to declare which identifiers to carry over to the next part.</p>
<pre>MATCH (user)-[:FRIEND]-(friend) WITH user, count(friend) AS friends ORDER BY friends DESC SKIP 1 LIMIT 3 RETURN user</pre> <p>You can also use ORDER BY, SKIP, LIMIT with WITH.</p>

UNION
<pre>MATCH (a)-[:KNOWS]->(b) RETURN b.name UNION MATCH (a)-[:LOVES]->(b) RETURN b.name</pre> <p>Returns the distinct union of all query results. Result column types and names have to match.</p>
<pre>MATCH (a)-[:KNOWS]->(b) RETURN b.name UNION ALL MATCH (a)-[:LOVES]->(b) RETURN b.name</pre> <p>Returns the union of all query results, including duplicated rows.</p>

SET
<pre>SET n.property1 = {value1}, n.property2 = {value2}</pre> <p>Update or create a property.</p>
<pre>SET n = {map}</pre> <p>Set all properties. This will remove any existing properties.</p>
<pre>SET n += {map}</pre> <p>Add and update properties, while keeping existing ones.</p>
<pre>SET n:Person</pre> <p>Adds a label Person to a node.</p>

DELETE
<pre>DELETE n, r</pre> <p>Delete a node and a relationship.</p>
<pre>DETACH DELETE n</pre> <p>Delete a node and all relationships connected to it.</p>
<pre>MATCH (n) DETACH DELETE n</pre> <p>Delete all nodes and relationships from the database.</p>

REMOVE
<pre>REMOVE n:Person</pre> <p>Remove a label from n.</p>
<pre>REMOVE n.property</pre> <p>Remove a property.</p>

FOREACH
<pre>FOREACH (r IN rels(path) SET r.marked = TRUE)</pre> <p>Execute a mutating operation for each relationship of a path.</p>
<pre>FOREACH (value IN coll CREATE (:Person {name:value}))</pre> <p>Execute a mutating operation for each element in a collection.</p>

Operators	
Mathematical	+, -, *, /, %, ^
Comparison	=, <>, <, >, <=, >=
Boolean	AND, OR, XOR, NOT
String	+
Collection	+, IN, [x], [x .. y]
Regular Expression	=~
String matching	STARTS WITH, ENDS WITH, CONTAINS

INDEX
<pre>CREATE INDEX ON :Person(name)</pre> <p>Create an index on the label Person and property name.</p>
<pre>MATCH (n:Person) WHERE n.name = {value}</pre> <p>An index can be automatically used for the equality comparison. Note that for example <code>lower(n.name) = {value}</code> will not use an index.</p>
<pre>MATCH (n:Person) WHERE n.name IN [{value}]</pre> <p>An index can be automatically used for the IN collection checks.</p>
<pre>MATCH (n:Person) USING INDEX n:Person(name) WHERE n.name = {value}</pre> <p>Index usage can be enforced, when Cypher uses a suboptimal index or more than one index should be used.</p>
<pre>DROP INDEX ON :Person(name)</pre> <p>Drop the index on the label Person and property name.</p>

CONSTRAINT
<pre>CREATE CONSTRAINT ON (p:Person) ASSERT p.name IS UNIQUE</pre> <p>Create a unique property constraint on the label Person and property name. If any other node with that label is updated or created with a name that already exists, the write operation will fail. This constraint will create an accompanying index.</p>
<pre>DROP CONSTRAINT ON (p:Person) ASSERT p.name IS UNIQUE</pre> <p>Drop the unique constraint and index on the label Person and property name.</p>
<pre>CREATE CONSTRAINT ON (p:Person) ASSERT exists(p.name)</pre> <p>Create a node property existence constraint on the label Person and property name. If a node with that label is created without a name, or if the name property is removed from an existing node with the Person label, the write operation will fail.</p>
<pre>DROP CONSTRAINT ON (p:Person) ASSERT exists(p.name)</pre> <p>Drop the node property existence constraint on the label Person and property name.</p>

<pre>CREATE CONSTRAINT ON ()-[l:LIKED]-() ASSERT exists(l.when)</pre> <p>Create a relationship property existence constraint on the type LIKED and property when. If a relationship with that type is created without a when, or if the when property is removed from an existing relationship with the LIKED type, the write operation will fail.</p>
<pre>DROP CONSTRAINT ON ()-[l:LIKED]-() ASSERT exists(l.when)</pre> <p>Drop the relationship property existence constraint on the type LIKED and property when.</p>

Import
<pre>LOAD CSV FROM 'http://neo4j.com/docs/2.3.3/cypher-refcard/csv/artists.csv' AS line CREATE (:Artist {name: line[1], year: toInt(line[2])})</pre> <p>Load data from a CSV file and create nodes.</p>
<pre>LOAD CSV WITH HEADERS FROM 'http://neo4j.com/docs/2.3.3/cypher-refcard/csv/artists-with-headers.csv' AS line CREATE (:Artist {name: line.Name, year: toInt(line.Year)})</pre> <p>Load CSV data which has headers.</p>
<pre>LOAD CSV FROM 'http://neo4j.com/docs/2.3.3/cypher-refcard/csv/artists-fieldterminator.csv' AS line FIELDTERMINATOR ',' CREATE (:Artist {name: line[1], year: toInt(line[2])})</pre> <p>Use a different field terminator, not the default which is a comma (with no whitespace around it).</p>

Import
<pre>LOAD CSV FROM 'http://neo4j.com/docs/2.3.3/cypher-refcard/csv/artists-fieldterminator.csv' AS line FIELDTERMINATOR ',' CREATE (:Artist {name: line[1], year: toInt(line[2])})</pre> <p>Use a different field terminator, not the default which is a comma (with no whitespace around it).</p>

NULL
<ul style="list-style-type: none"> NULL is used to represent missing/undefined values. NULL is not equal to NULL. Not knowing two values does not imply that they are the same value. So the expression <code>NULL = NULL</code> yields <code>NULL</code> and not <code>TRUE</code>. To check if an expression is NULL, use <code>IS NULL</code>. Arithmetic expressions, comparisons and function calls (except <code>coalesce</code>) will return <code>NULL</code> if any argument is <code>NULL</code>. An attempt to access a missing element in a collection or a property that doesn't exist yields <code>NULL</code>. In <code>OPTIONAL MATCH</code> clauses, <code>NULLS</code> will be used for missing parts of the pattern.

Labels
<pre>CREATE (n:Person {name: {value}})</pre> <p>Create a node with label and property.</p>
<pre>MERGE (n:Person {name: {value}})</pre> <p>Matches or creates unique node(s) with label and property.</p>
<pre>SET n:Spouse:Parent:Employee</pre> <p>Add label(s) to a node.</p>
<pre>MATCH (n:Person)</pre> <p>Matches nodes labeled Person.</p>
<pre>MATCH (n:Person) WHERE n.name = {value}</pre> <p>Matches nodes labeled Person with the given name.</p>
<pre>WHERE (n:Person)</pre> <p>Checks existence of label on node.</p>
<pre>Labels(n)</pre> <p>Labels of the node.</p>
<pre>REMOVE n:Person</pre> <p>Remove label from node.</p>

Neo4j Cypher Refcard 2.3.3

Patterns
<pre>(n:Person)</pre> <p>Node with <code>Person</code> label.</p>
<pre>(n:Person:Swedish)</pre> <p>Node with both <code>Person</code> and <code>Swedish</code> labels.</p>
<pre>(n:Person {name: {value}})</pre> <p>Node with the declared properties.</p>
<pre>(n)-->(m)</pre> <p>Relationship from <code>n</code> to <code>m</code>.</p>
<pre>(n)--(m)</pre> <p>Relationship in any direction between <code>n</code> and <code>m</code>.</p>
<pre>(n:Person)-->(m)</pre> <p>Node <code>n</code> labeled <code>Person</code> with relationship to <code>m</code>.</p>
<pre>(m)-[:KNOWS]-(n)</pre> <p>Relationship of type <code>KNOWS</code> from <code>n</code> to <code>m</code>.</p>
<pre>(n)-[:KNOWS LOVES]->(m)</pre> <p>Relationship of type <code>KNOWS</code> or of type <code>LOVES</code> from <code>n</code> to <code>m</code>.</p>
<pre>(n)-[r]->(m)</pre> <p>Bind the relationship to identifier <code>r</code>.</p>
<pre>(n)-[*1..5]->(m)</pre> <p>Variable length path of between 1 and 5 relationships from <code>n</code> to <code>m</code>.</p>
<pre>(n)-[*]->(m)</pre> <p>Variable length path of any number of relationships from <code>n</code> to <code>m</code>. (Please see the performance tips.)</p>
<pre>(n)-[:KNOWS]->(m {property: {value}})</pre> <p>A relationship of type <code>KNOWS</code> from a node <code>n</code> to a node <code>m</code> with the declared property.</p>
<pre>shortestPath((n1:Person)-[*..6]-(n2:Person))</pre> <p>Find a single shortest path.</p>
<pre>allShortestPaths((n1:Person)-[*..6]->(n2:Person))</pre> <p>Find all shortest paths.</p>
<pre>size((n)-->()->())</pre> <p>Count the paths matching the pattern.</p>

Collections
<pre>["a", "b", "c"] AS coll</pre> <p>Literal collections are declared in square brackets.</p>
<pre>size({coll}) AS len, {coll}[0] AS value</pre> <p>Collections can be passed in as parameters.</p>
<pre>range({firstNum}, {lastNum}, {step}) AS coll</pre> <p>Range creates a collection of numbers (<code>step</code> is optional), other functions returning collections are: <code>labels</code>, <code>nodes</code>, <code>relationships</code>, <code>rels</code>, <code>filter</code>, <code>extract</code>.</p>
<pre>MATCH (a)-[r:KNOWS*]->() RETURN r AS rels</pre> <p>Relationship identifiers of a variable length path contain a collection of relationships.</p>
<pre>RETURN matchedNode.coll[0] AS value, size(matchedNode.coll) AS len</pre> <p>Properties can be arrays/collections of strings, numbers or booleans.</p>
<pre>coll[{idx}] AS value, coll[{startIdx}..{endIdx}] AS slice</pre> <p>Collection elements can be accessed with <code>idx</code> subscripts in square brackets. Invalid indexes return <code>NULL</code>. Slices can be retrieved with intervals from <code>start_idx</code> to <code>end_idx</code> each of which can be omitted or negative. Out of range elements are ignored.</p>
<pre>UNWIND {names} AS name MATCH (n {name: name}) RETURN avg(n.age)</pre> <p>With <code>UNWIND</code>, you can transform any collection back into individual rows. The example matches all names from a list of names.</p>

Maps
<pre>{name: "Alice", age: 38, address: {city: 'London', residential: true}}</pre> <p>Literal maps are declared in curly braces much like property maps. Nested maps and collections are supported.</p>
<pre>MERGE (p:Person {name: {map}.name}) ON CREATE SET p = {map}</pre> <p>Maps can be passed in as parameters and used as <code>map</code> or by accessing keys.</p>
<pre>MATCH (matchedNode:Person) RETURN matchedNode</pre> <p>Nodes and relationships are returned as maps of their data.</p>
<pre>map.name, map.age, map.children[0]</pre> <p>Map entries can be accessed by their keys. Invalid keys result in an error.</p>

CASE
<pre>CASE n.eyes WHEN "blue" THEN 1 WHEN "brown" THEN 2 ELSE 3 END</pre> <p>Return <code>THEN</code> value from the matching <code>WHEN</code> value. The <code>ELSE</code> value is optional, and substituted for <code>NULL</code> if missing.</p>
<pre>CASE WHEN n.eyes = "blue" THEN 1 WHEN n.age < 40 THEN 2 ELSE 3 END</pre> <p>Return <code>THEN</code> value from the first <code>WHEN</code> predicate evaluating to <code>TRUE</code>. Predicates are evaluated in order.</p>

Relationship Functions
<pre>type(a_relationship)</pre> <p>String representation of the relationship type.</p>
<pre>startNode(a_relationship)</pre> <p>Start node of the relationship.</p>
<pre>endNode(a_relationship)</pre> <p>End node of the relationship.</p>
<pre>id(a_relationship)</pre> <p>The internal id of the relationship.</p>

Collection Predicates
<pre>all(x IN coll WHERE exists(x.property))</pre> <p>Returns <code>true</code> if the predicate is <code>TRUE</code> for all elements of the collection.</p>
<pre>any(x IN coll WHERE exists(x.property))</pre> <p>Returns <code>true</code> if the predicate is <code>TRUE</code> for at least one element of the collection.</p>
<pre>none(x IN coll WHERE exists(x.property))</pre> <p>Returns <code>TRUE</code> if the predicate is <code>FALSE</code> for all elements of the collection.</p>
<pre>single(x IN coll WHERE exists(x.property))</pre> <p>Returns <code>TRUE</code> if the predicate is <code>TRUE</code> for exactly one element in the collection.</p>

Functions
<pre>coalesce(n.property, {defaultValue})</pre> <p>The first non-<code>NULL</code> expression.</p>
<pre>timestamp()</pre> <p>Milliseconds since midnight, January 1, 1970 UTC.</p>
<pre>id(nodeOrRelationship)</pre> <p>The internal id of the relationship or node.</p>
<pre>toInt({expr})</pre> <p>Converts the given input into an integer if possible; otherwise it returns <code>NULL</code>.</p>
<pre>toFloat({expr})</pre> <p>Converts the given input into a floating point number if possible; otherwise it returns <code>NULL</code>.</p>
<pre>keys({expr})</pre> <p>Returns a collection of string representations for the property names of a node, relationship, or map.</p>

Path Functions
<pre>length(path)</pre> <p>The number of relationships in the path.</p>
<pre>nodes(path)</pre> <p>The nodes in the path as a collection.</p>
<pre>relationships(path)</pre> <p>The relationships in the path as a collection.</p>
<pre>extract(x IN nodes(path) x.prop)</pre> <p>Extract properties from the nodes in a path.</p>

Mathematical Functions
<pre>abs({expr})</pre> <p>The absolute value.</p>
<pre>rand()</pre> <p>A random number between 0 and 1. Returns a new value for each call. Also useful for selecting subset or random ordering.</p>
<pre>round({expr})</pre> <p>Round to the nearest integer, <code>ceil</code> and <code>floor</code> find the next integer up or down.</p>
<pre>sqrt({expr})</pre> <p>The square root.</p>
<pre>sign({expr})</pre> <p>0 if zero, -1 if negative, 1 if positive.</p>
<pre>sin({expr})</pre> <p>Trigonometric functions, also <code>cos</code>, <code>tan</code>, <code>cot</code>, <code>asin</code>, <code>acos</code>, <code>atan</code>, <code>atan2</code>, <code>haversin</code>.</p>
<pre>degrees({expr}), radians({expr}), pi()</pre> <p>Converts radians into degrees, use <code>radians</code> for the reverse. <code>pi</code> for π.</p>
<pre>log10({expr}), log({expr}), exp({expr}), e()</pre> <p>Logarithm base 10, natural logarithm, <code>e</code> to the power of the parameter. Value of <code>e</code>.</p>

String Functions
<pre>toString({expression})</pre> <p>String representation of the expression.</p>
<pre>replace({original}, {search}, {replacement})</pre> <p>Replace all occurrences of <code>search</code> with <code>replacement</code>. All arguments are be expressions.</p>
<pre>substring({original}, {begin}, {subLength})</pre> <p>Get part of a string. The <code>subLength</code> argument is optional.</p>
<pre>left({original}, {subLength}), right({original}, {subLength})</pre> <p>The first part of a string. The last part of the string.</p>
<pre>trim({original}), ltrim({original}), rtrim({original})</pre> <p>Trim all whitespace, or on left or right side.</p>
<pre>upper({original}), lower({original})</pre> <p>UPPERCASE and lowercase.</p>
<pre>split({original}, {delimiter})</pre> <p>Split a string into a collection of strings.</p>
<pre>reverse({original})</pre> <p>Reverse a string.</p>
<pre>length({string})</pre> <p>Calculate the number of characters in the string.</p>

Predicates
<pre>n.property <> {value}</pre> <p>Use comparison operators.</p>
<pre>exists(n.property)</pre> <p>Use functions.</p>
<pre>n.number >= 1 AND n.number <= 10</pre> <p>Use boolean operators to combine predicates.</p>
<pre>1 <= n.number <= 10</pre> <p>Use chained operators to combine predicates.</p>
<pre>n:Person</pre> <p>Check for node labels.</p>
<pre>identifier IS NULL</pre> <p>Check if something is <code>NULL</code>.</p>
<pre>NOT exists(n.property) OR n.property = {value}</pre> <p>Either property does not exist or predicate is <code>TRUE</code>.</p>
<pre>n.property = {value}</pre> <p>Non-existing property returns <code>NULL</code>, which is not equal to anything.</p>
<pre>n["property"] = {value}</pre> <p>Properties may also be accessed using a dynamically computed property name.</p>
<pre>n.property STARTS WITH "Tob" OR n.property ENDS WITH "n" OR n.property CONTAINS "goodie"</pre> <p>String matching.</p>
<pre>n.property =~ "Tob.*"</pre> <p>String regular expression matching.</p>
<pre>(n)-[:KNOWS]->(m)</pre> <p>Make sure the pattern has at least one match.</p>
<pre>NOT (n)-[:KNOWS]->(m)</pre> <p>Exclude matches to <code>(n)-[:KNOWS]->(m)</code> from the result.</p>
<pre>n.property IN [{value1}, {value2}]</pre> <p>Check if an element exists in a collection.</p>

Collection Expressions
<pre>size({coll})</pre> <p>Number of elements in the collection.</p>
<pre>head({coll}), last({coll}), tail({coll})</pre> <p><code>head</code> returns the first, <code>last</code> the last element of the collection. <code>tail</code> returns all but the first element. All return <code>NULL</code> for an empty collection.</p>
<pre>[x IN coll WHERE x.prop <> {value} x.prop]</pre> <p>Combination of filter and <code>extract</code> in a concise notation.</p>
<pre>extract(x IN coll x.prop)</pre> <p>A collection of the value of the expression for each element in the original collection.</p>
<pre>filter(x IN coll WHERE x.prop <> {value})</pre> <p>A filtered collection of the elements where the predicate is <code>TRUE</code>.</p>
<pre>reduce(s = "", x IN coll s + x.prop)</pre> <p>Evaluate expression for each element in the collection, accumulate the results.</p>

Aggregation
<pre>count(*)</pre> <p>The number of matching rows.</p>
<pre>count(identifier)</pre> <p>The number of non-<code>NULL</code> values.</p>
<pre>count(DISTINCT identifier)</pre> <p>All aggregation functions also take the <code>DISTINCT</code> modifier, which removes duplicates from the values.</p>
<pre>collect(n.property)</pre> <p>Collection from the values, ignores <code>NULL</code>.</p>
<pre>sum(n.property)</pre> <p>Sum numerical values. Similar functions are <code>avg</code>, <code>min</code>, <code>max</code>.</p>
<pre>percentileDisc(n.property, {percentile})</pre> <p>Discrete percentile. Continuous percentile is <code>percentileCont</code>. The <code>percentile</code> argument is from 0.0 to 1.0.</p>
<pre>stdev(n.property)</pre> <p>Standard deviation for a sample of a population. For an entire population use <code>stdevp</code>.</p>

START
<pre>START n = node:nodeIndexName(key = {value})</pre> <p>Query the index named <code>nodeIndexName</code> with an exact query. Use <code>node_auto_index</code> for the automatic index. Note that other uses of <code>START</code> have been removed as of Cypher 2.2.</p>

CREATE UNIQUE
<pre>CREATE UNIQUE (n)-[:KNOWS]->(m {property: {value}})</pre> <p>Match pattern or create it if it does not exist. The pattern can not include any optional parts.</p>

Performance
<ul style="list-style-type: none">Use parameters instead of literals when possible. This allows Cypher to re-use your queries instead of having to parse and build new execution plans.Always set an upper limit for your variable length patterns. It's easy 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 <code>PROFILE</code> / <code>EXPLAIN</code> to analyze the performance of your queries. See Query Tuning for more information.