

## Legend

Read
Write
General
Functions
Schema
Performance

## Syntax

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

MATCH
<code>MATCH (n:Person)-[:KNOWS]-&gt;(m:Person)</code> <code>WHERE n.name = 'Alice'</code> Node patterns can contain labels and properties.
<code>MATCH (n)--&gt;(m)</code> Any pattern can be used in MATCH.
<code>MATCH (n {name: 'Alice'})--&gt;(m)</code> Patterns with node properties.
<code>MATCH p = (n)--&gt;(m)</code> Assign a path to p.
<code>OPTIONAL MATCH (n)-[r]-&gt;(m)</code> Optional pattern: nulls will be used for missing parts.

WHERE
<code>WHERE n.property &lt;&gt; \$value</code> 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.

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

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

CREATE
<code>CREATE (n {name: \$value})</code> Create a node with the given properties.
<code>CREATE (n \$map)</code> Create a node with the given properties.
<code>UNWIND \$listOfMaps AS properties</code> <code>CREATE (n) SET n = properties</code> Create nodes with the given properties.
<code>CREATE (n)-[r:KNOWS]-&gt;(m)</code> Create a relationship with the given type and direction; bind a variable to it.
<code>CREATE (n)-[:LOVES {since: \$value}]-&gt;(m)</code> Create a relationship with the given type, direction, and properties.

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

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

WITH
<code>MATCH (user)-[:FRIEND]- (friend)</code> <code>WHERE user.name = \$name</code> <code>WITH user, count(friend) AS friends</code> <code>WHERE friends &gt; 10</code> <code>RETURN user</code> The WITH syntax is similar to RETURN. It separates query parts explicitly, allowing you to declare which variables to carry over to the next part.
<code>MATCH (user)-[:FRIEND]- (friend)</code> <code>WITH user, count(friend) AS friends</code> <code>ORDER BY friends DESC</code> <code>SKIP 1</code> <code>LIMIT 3</code> <code>RETURN user</code> ORDER BY, SKIP, and LIMIT can also be used with WITH.

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

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

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

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

FOREACH
<code>FOREACH (r IN relationships(path)  </code> <code>SET r.marked = true)</code> Execute a mutating operation for each relationship in a path.
<code>FOREACH (value IN coll  </code> <code>CREATE (:Person {name: value}))</code> Execute a mutating operation for each element in a list.

CALL
<code>CALL db.labels() YIELD label</code> This shows a standalone call to the built-in procedure <code>db.labels</code> to list all labels used in the database. Note that required procedure arguments are given explicitly in brackets after the procedure name.
<code>CALL java.stored.procedureWithArgs</code> Standalone calls may omit YIELD and also provide arguments implicitly via statement parameters, e.g. a standalone call requiring one argument <code>input</code> may be run by passing the parameter map <code>{input: 'foo'}</code> .
<code>CALL db.labels() YIELD label</code> <code>RETURN count(label) AS count</code> Calls the built-in procedure <code>db.labels</code> 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.

Import
<code>LOAD CSV FROM</code> <code>'https://neo4j.com/docs/cypher-refcard/3.4/csv/artists.csv' AS line</code> <code>CREATE (:Artist {name: line[1], year: toInteger(line[2])})</code> Load data from a CSV file and create nodes.
<code>LOAD CSV WITH HEADERS FROM</code> <code>'https://neo4j.com/docs/cypher-refcard/3.4/csv/artists-with-headers.csv' AS line</code> <code>CREATE (:Artist {name: line.Name, year: toInteger(line.Year)})</code> Load CSV data which has headers.

<code>USING PERIODIC COMMIT 500</code> <code>LOAD CSV WITH HEADERS FROM</code> <code>'https://neo4j.com/docs/cypher-refcard/3.4/csv/artists-with-headers.csv' AS line</code> <code>CREATE (:Artist {name: line.Name, year: toInteger(line.Year)})</code> Commit the current transaction after every 500 rows when importing large amounts of data.
<code>LOAD CSV FROM</code> <code>'https://neo4j.com/docs/cypher-refcard/3.4/csv/artists-fieldterminator.csv'</code> <code>AS line FIELDTERMINATOR ','</code> <code>CREATE (:Artist {name: line[1], year: toInteger(line[2])})</code> Use a different field terminator, not the default which is a comma (with no whitespace around it).

Performance
<ul style="list-style-type: none"> <li>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.</li> <li>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.</li> <li>Return only the data you need. Avoid returning whole nodes and relationships — instead, pick the data you need and return only that.</li> <li>Use PROFILE / EXPLAIN to analyze the performance of your queries. See <a href="#">Query Tuning</a> for more information on these and other topics, such as planner hints.</li> </ul>

## DELETE

`DELETE n, r`  
Delete a node and a relationship.

`DETACH DELETE n`  
Delete a node and all relationships connected to it.

`MATCH (n)`  
`DETACH DELETE n`  
Delete all nodes and relationships from the database.

## FOREACH

`FOREACH (r IN relationships(path) | SET r.marked = true)`  
Execute a mutating operation for each relationship in a path.

`FOREACH (value IN coll | CREATE (:Person {name: value}))`  
Execute a mutating operation for each element in a list.

## CALL

`CALL db.labels() YIELD label`  
This shows a standalone call to the built-in 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.

`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 count`  
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`.

## Import

`LOAD CSV FROM 'https://neo4j.com/docs/cypher-refcard/3.4/csv/artists.csv' AS line`  
`CREATE (:Artist {name: line[1], year: toInteger(line[2])})`  
Load data from a CSV file and create nodes.

`LOAD CSV WITH HEADERS FROM 'https://neo4j.com/docs/cypher-refcard/3.4/csv/artists-with-headers.csv' AS line`  
`CREATE (:Artist {name: line.Name, year: toInteger(line.Year)})`  
Load CSV data which has headers.

`USING PERIODIC COMMIT 500`  
`LOAD CSV WITH HEADERS FROM 'https://neo4j.com/docs/cypher-refcard/3.4/csv/artists-with-headers.csv' AS line`  
`CREATE (:Artist {name: line.Name, year: toInteger(line.Year)})`  
Commit the current transaction after every 500 rows when importing large amounts of data.

`LOAD CSV FROM 'https://neo4j.com/docs/cypher-refcard/3.4/csv/artists-fieldterminator.csv' AS line FIELDTERMINATOR ';'`   
`CREATE (:Artist {name: line[1], year: toInteger(line[2])})`  
Use a different field terminator, not the default which is a comma (with no whitespace around it).

## Performance

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

## Operators

General	<code>DISTINCT</code> , <code>.</code> , <code>[]</code>
Mathematical	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>%</code> , <code>^</code>
Comparison	<code>=</code> , <code>&lt;&gt;</code> , <code>&lt;</code> , <code>&gt;</code> , <code>&lt;=</code> , <code>&gt;=</code> , <code>IS NULL</code> , <code>IS NOT NULL</code>
Boolean	<code>AND</code> , <code>OR</code> , <code>XOR</code> , <code>NOT</code>
String	<code>+</code>
List	<code>+</code> , <code>IN</code> , <code>[x]</code> , <code>[x .. y]</code>
Regular Expression	<code>=~</code>
String matching	<code>STARTS WITH</code> , <code>ENDS WITH</code> , <code>CONTAINS</code>

## null

- `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 `null = null` yields `null` and not `true`. To check if an expression is `null`, use `IS NULL`.
- Arithmetic expressions, comparisons and function calls (except `coalesce`) will return `null` if any argument is `null`.
- An attempt to access a missing element in a list or a property that doesn't exist yields `null`.
- In `OPTIONAL MATCH` clauses, `nulls` will be used for missing parts of the pattern.

## Predicates

`n.property <> $value`

Use comparison operators.

`exists(n.property)`

Use functions.

`n.number >= 1 AND n.number <= 10`

Use boolean operators to combine predicates.

`1 <= n.number <= 10`

Use chained operators to combine predicates.

`n:Person`

Check for node labels.

`variable IS NULL`

Check if something is `null`.

`NOT exists(n.property) OR n.property = $value`

Either the property does not exist or the predicate is `true`.

`n.property = $value`

Non-existing property returns `null`, which is not equal to anything.

`n["property"] = $value`

Properties may also be accessed using a dynamically computed property name.

`n.property STARTS WITH 'Tob' OR`  
`n.property ENDS WITH 'n' OR`  
`n.property CONTAINS 'goodie'`

String matching.

`n.property =~ 'Tob.*'`

String regular expression matching.

`(n)-[:KNOWS]->(m)`

Ensure the pattern has at least one match.

`NOT (n)-[:KNOWS]->(m)`

Exclude matches to `(n)-[:KNOWS]->(m)` from the result.

`n.property IN [$value1, $value2]`

Check if an element exists in a list.

## CASE

`CASE n.eyes`  
`WHEN 'blue' THEN 1`  
`WHEN 'brown' THEN 2`  
`ELSE 3`  
`END`

Return `THEN` value from the matching `WHEN` value. The `ELSE` value is optional, and substituted for `null` if missing.

`CASE`  
`WHEN n.eyes = 'blue' THEN 1`  
`WHEN n.age < 40 THEN 2`  
`ELSE 3`  
`END`

Return `THEN` value from the first `WHEN` predicate evaluating to `true`. Predicates are evaluated in order.

## Patterns

`(n:Person)`

Node with `Person` label.

`(n:Person:Swedish)`

Node with both `Person` and `Swedish` labels.

`(n:Person {name: $value})`

Node with the declared properties.

`()-[r {name: $value}]-()`

Matches relationships with the declared properties.

`(n)-->(m)`

Relationship from `n` to `m`.

`(n)--(m)`

Relationship in any direction between `n` and `m`.

`(n:Person)-->(m)`

Node `n` labeled `Person` with relationship to `m`.

`(m)-[:KNOWS]-(n)`

Relationship of type `KNOWS` from `n` to `m`.

`(n)-[:KNOWS|:LOVES]->(m)`

Relationship of type `KNOWS` or of type `LOVES` from `n` to `m`.

`(n)-[r]->(m)`

Bind the relationship to variable `r`.

`(n)-[*1..5]->(m)`

Variable length path of between 1 and 5 relationships from `n` to `m`.

`(n)-[*]->(m)`

Variable length path of any number of relationships from `n` to `m`. (See Performance section.)

`(n)-[:KNOWS]->(m {property: $value})`

A relationship of type `KNOWS` from a node `n` to a node `m` with the declared property.

`shortestPath((n1:Person)-[*..6]-(n2:Person))`

Find a single shortest path.

`allShortestPaths((n1:Person)-[*..6]->(n2:Person))`

Find all shortest paths.

`size((n)-->()->())`

Count the paths matching the pattern.

## Labels

`CREATE (n:Person {name: $value})`

Create a node with label and property.

`MERGE (n:Person {name: $value})`

Matches or creates unique node(s) with the label and property.

`SET n:Spouse:Parent:Employee`

Add label(s) to a node.

`MATCH (n:Person)`

Matches nodes labeled `Person`.

`MATCH (n:Person)`

`WHERE n.name = $value`

Matches nodes labeled `Person` with the given `name`.

`WHERE (n:Person)`

Checks the existence of the label on the node.

`labels(n)`

Labels of the node.

`REMOVE n:Person`

Remove the label from the node.

## Maps

`{name: 'Alice', age: 38,`  
`address: {city: 'London', residential: true}}`

Literal maps are declared in curly braces much like property maps. Lists are supported.

`WITH {person: {name: 'Anne', age: 25}} AS p`

`RETURN p.person.name`

Access the property of a nested map.

`MERGE (p:Person {name: $map.name})`

`ON CREATE SET p = $map`

Maps can be passed in as parameters and used either as a map or by accessing keys.

`MATCH (matchedNode:Person)`

`RETURN matchedNode`

Nodes and relationships are returned as maps of their data.

`map.name`, `map.age`, `map.children[0]`

Map entries can be accessed by their keys. Invalid keys result in an error.

# Neo4j Cypher Refcard 3.4

## Functions

`coalesce(n.property, $defaultValue)`  
The first non-null expression.

`timestamp()`  
Milliseconds since midnight, January 1, 1970 UTC.

`id(nodeOrRelationship)`  
The internal id of the relationship or node.

`toInteger($expr)`  
Converts the given input into an integer if possible; otherwise it returns `null`.

`toFloat($expr)`  
Converts the given input into a floating point number if possible; otherwise it returns `null`.

`toBoolean($expr)`  
Converts the given input into a boolean if possible; otherwise it returns `null`.

`keys($expr)`  
Returns a list of string representations for the property names of a node, relationship, or map.

`properties({expr})`  
Returns a map containing all the properties of a node or relationship.

## Temporal Functions

`date("2018-04-05")`  
Returns a date parsed from a string.

`localtime("12:45:30.25")`  
Returns a time with no time zone.

`time("12:45:30.25+01:00")`  
Returns a time in a specified time zone.

`localdatetime("2018-04-05T12:34:00")`  
Returns a datetime with no time zone.

`datetime("2018-04-05T12:34:00[Europe/Berlin]")`  
Returns a datetime in the specified time zone.

`date({year: {year}, month: {month}, day: {day}})`  
All of the temporal functions can also be called with a map of named components. This example returns a date from `year`, `month` and `day` components. Each function supports a different set of possible components.

`datetime({date: {date}, time: {time}})`  
Temporal types can be created by combining other types. This example creates a `datetime` from a `date` and a `time`.

`date({date: {datetime}, day: 5})`  
Temporal types can be created by selecting from more complex types, as well as overriding individual components. This example creates a `date` by selecting from a `datetime`, as well as overriding the `day` component.

`WITH date("2018-04-05") AS d  
RETURN d.year, d.month, d.day, d.week, d.dayOfWeek`  
Accessors allow extracting components of temporal types.

## Mathematical Functions

`abs($expr)`  
The absolute value.

`rand()`  
Returns a random number in the range from 0 (inclusive) to 1 (exclusive), `[0,1)`. Returns a new value for each call. Also useful for selecting a subset or random ordering.

`round($expr)`  
Round to the nearest integer; `ceil()` and `floor()` find the next integer up or down.

`sqrt($expr)`  
The square root.

`sign($expr)`  
0 if zero, -1 if negative, 1 if positive.

`sin($expr)`  
Trigonometric functions also include `cos()`, `tan()`, `cot()`, `asin()`, `acos()`, `atan()`, `atan2()`, and `haversin()`. All arguments for the trigonometric functions should be in radians, if not otherwise specified.

`degrees($expr)`, `radians($expr)`, `pi()`  
Converts radians into degrees; use `radians()` for the reverse, and `pi()` for  $\pi$ .

`log10($expr)`, `log($expr)`, `exp($expr)`, `e()`  
Logarithm base 10, natural logarithm, e to the power of the parameter, and the value of e.

## Spatial Functions

`point({x: $x, y: $y})`  
Returns a point in a 2D cartesian coordinate system.

`point({latitude: $y, longitude: $x})`  
Returns a point in a 2D geographic coordinate system, with coordinates specified in decimal degrees.

`point({x: $x, y: $y, z: $z})`  
Returns a point in a 3D cartesian coordinate system.

`point({latitude: $y, longitude: $x, height: $z})`  
Returns a point in a 3D geographic coordinate system, with latitude and longitude in decimal degrees, and height in meters.

`distance(point({x: $x1, y: $y1}), point({x: $x2, y: $y2}))`  
Returns a floating point number representing the linear distance between two points. The returned units will be the same as those of the point coordinates, and it will work for both 2D and 3D cartesian points.

`distance(point({latitude: $y1, longitude: $x1}), point({latitude: $y2, longitude: $x2}))`  
Returns the geodesic distance between two points in meters. It can be used for 3D geographic points as well.

## Duration Functions

`duration("P1Y2M10DT12H45M30.25S")`  
Returns a duration of 1 year, 2 months, 10 days, 12 hours, 45 minutes and 30.25 seconds.

`duration.between($date1,$date2)`  
Returns a duration between two temporal instances.

`WITH duration("P1Y2M10DT12H45M") AS d  
RETURN d.years, d.months, d.days, d.hours, d.minutes`  
Returns 1 year, 14 months, 10 days, 12 hours and 765 minutes.

`WITH duration("P1Y2M10DT12H45M") AS d  
RETURN d.years, d.monthsOfYear, d.days, d.hours, d.minutesOfHour`  
Returns 1 year, 2 months, 10 days, 12 hours and 45 minutes.

`date("2015-01-01") + duration("P1Y1M1D")`  
Returns a date of 2016-02-02. It is also possible to subtract durations from temporal instances.

`duration("PT30S") * 10`  
Returns a duration of 5 minutes. It is also possible to divide a duration by a number.

## String Functions

`toString($expression)`  
String representation of the expression.

`replace($original, $search, $replacement)`  
Replace all occurrences of `search` with `replacement`. All arguments must be expressions.

`substring($original, $begin, $subLength)`  
Get part of a string. The `subLength` argument is optional.

`left($original, $subLength),  
right($original, $subLength)`  
The first part of a string. The last part of the string.

`trim($original), lTrim($original),  
rTrim($original)`  
Trim all whitespace, or on the left or right side.

`toUpper($original), toLower($original)`  
UPPERCASE and lowercase.

`split($original, $delimiter)`  
Split a string into a list of strings.

`reverse($original)`  
Reverse a string.

`size($string)`  
Calculate the number of characters in the string.

## Aggregating Functions

`count(*)`  
The number of matching rows.

`count(variable)`  
The number of non-null values.

`count(DISTINCT variable)`  
All aggregating functions also take the `DISTINCT` operator, which removes duplicates from the values.

`collect(n.property)`  
List from the values, ignores `null`.

`sum(n.property)`  
Sum numerical values. Similar functions are `avg()`, `min()`, `max()`.

`percentileDisc(n.property, $percentile)`  
Discrete percentile. Continuous percentile is `percentileCont()`. The `percentile` argument is from 0.0 to 1.0.

`stDev(n.property)`  
Standard deviation for a sample of a population. For an entire population use `stDevP()`.

## Path Functions

`length(path)`  
The number of relationships in the path.

`nodes(path)`  
The nodes in the path as a list.

`relationships(path)`  
The relationships in the path as a list.

`extract(x IN nodes(path) | x.prop)`  
Extract properties from the nodes in a path.

## Relationship Functions

`type(a_relationship)`  
String representation of the relationship type.

`startNode(a_relationship)`  
Start node of the relationship.

`endNode(a_relationship)`  
End node of the relationship.

`id(a_relationship)`  
The internal id of the relationship.

## INDEX

`CREATE INDEX ON :Person(name)`  
Create an index on the label `Person` and property `name`.

`MATCH (n:Person) WHERE n.name = $value`  
An index can be automatically used for the equality comparison. Note that for example `toLower(n.name) = $value` will not use an index.

`MATCH (n:Person)  
WHERE n.name IN [$value]`  
An index can automatically be used for the `IN` list checks.

`MATCH (n:Person)  
USING INDEX n:Person(name)  
WHERE n.name = $value`  
Index usage can be enforced when Cypher uses a suboptimal index, or more than one index should be used.

`DROP INDEX ON :Person(name)`  
Drop the index on the label `Person` and property `name`.

## CONSTRAINT

`CREATE CONSTRAINT ON (p:Person)  
ASSERT p.name IS UNIQUE`

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.

`DROP CONSTRAINT ON (p:Person)  
ASSERT p.name IS UNIQUE`  
Drop the unique constraint and index on the label `Person` and property `name`.

`CREATE CONSTRAINT ON (p:Person)  
ASSERT exists(p.name)`

(★) 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.

`DROP CONSTRAINT ON (p:Person)  
ASSERT exists(p.name)`  
(★) Drop the node property existence constraint on the label `Person` and property `name`.

`CREATE CONSTRAINT ON ()-[l:LIKED]-()  
ASSERT exists(l.when)`

(★) 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.

`DROP CONSTRAINT ON ()-[l:LIKED]-()  
ASSERT exists(l.when)`

(★) Drop the relationship property existence constraint on the type `LIKED` and property `when`.

`CREATE CONSTRAINT ON (p:Person)  
ASSERT (p.firstname, p.surname) IS NODE KEY`

(★) Create a Node Key constraint on the label `Person` and properties `firstname` and `surname`. If a node with that label is created without both `firstname` and `surname` or if the combination of the two is not unique, or if the `firstname` and/or `surname` labels on an existing node with the `Person` label is modified to violate these constraints, the write operation will fail.

`DROP CONSTRAINT ON (p:Person)  
ASSERT (p.firstname, p.surname) IS NODE KEY`

(★) Drop the Node Key constraint on the label `Person` and properties `firstname` and `surname`.