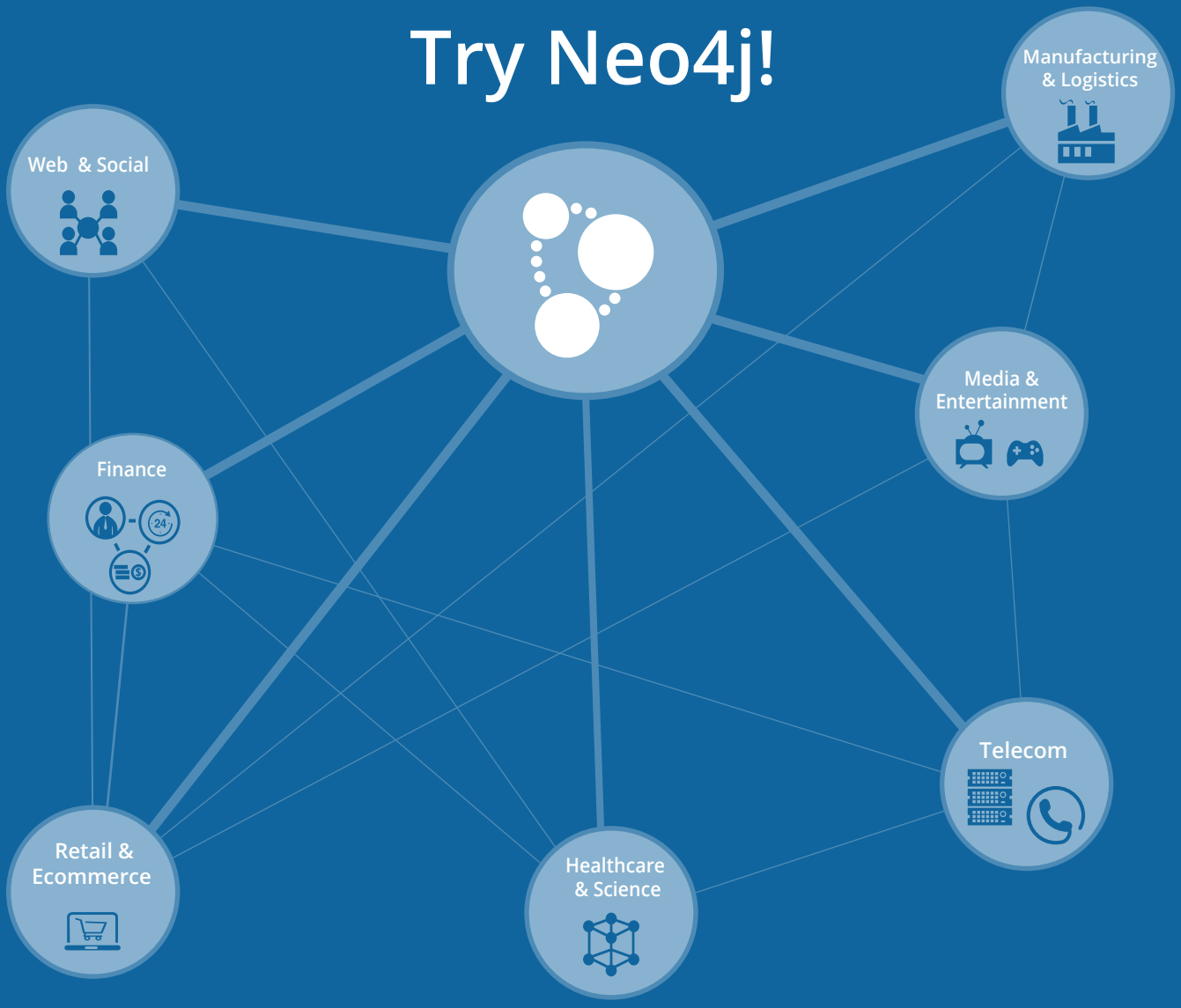
Neo4j

Every industry that has data also has a graph.

Connect the Data. Transform Your Business. Try Neo4j!



Download Now.

Thousands of organizations, from startups to Fortune 500 companies, are using Neo4j to build new and innovative applications that leverage connections in data.

Learn more: neo4j.com
Follow us: @neo4j

Querying Graphs with Neo4j

By Michael Hunger

WHAT IS A GRAPH DATABASE?

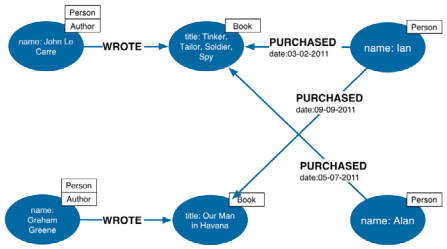
We live in a connected world. There are no isolated pieces of information, but rich, connected domains all around us. Only a database that embraces relationships as a core aspect of its data model is able to store, process, and query connections efficiently. While other databases compute joins expensively at query time, a graph database stores connections as first class citizens, readily available for any "join-like" navigation operation. Accessing those already persisted connections is an efficient, constant-time operation and allows you to traverse millions of relationships per second.

Independent of the size of the total dataset, graph databases excel at managing highly connected data and complex queries. Armed only with a pattern and a set of starting points, graph databases explore the larger neighborhoods around these initial entities — collecting and aggregating information from millions of nodes and relationships — but leaving the billions outside the search perimeter untouched.

THE PROPERTY GRAPH MODEL

If you've ever worked with an object model or entity relationship diagram, the labeled property graph model will be familiar.

Labeled Property Graph Data Model



The property graph contains connected entities (the *nodes*) which can hold any number of properties (key-value-pairs). Nodes can be tagged with several labels representing different roles in the domain. Besides putting a subset of node properties and relationships into a certain context, labels also allow you to attach metadata — like index or constraint information — to nodes.

Relationships provide directed, named semantic connections between two nodes. A relationship always has a *direction*, a *type*, a *start node*, and an *end node*. Like nodes, relationships can have arbitrary properties. Usually, relationships contain quantitative properties, such as weights, distances, ratings, time intervals, or strengths. As relationships are stored efficiently in the graph database, two nodes can have as many different or similar relationships connecting them without sacrificing performance. Note that although they are directed, relationships can always be navigated in both directions.

There is only one consistency rule in a graph database: "No broken links". Since a relationship always has to have a start and end node, you can only delete a node by also removing its associated relationships.

WHAT IS NEO4J?

Neo4j is an open-source, NoSQL graph database implemented mainly in Java and Scala. Its development started in 2003 and it has been sponsored by Neo Technology, Inc. since 2011. The source code and issue tracking is available on github.com/neo4j, with an active community supporting users on Stack Overflow and the Neo4j Google Group.

Neo4j is used by hundreds of thousands of users in almost all industries. Use cases include matchmaking, network management, impact analysis, software analytics, scientific research, routing, organizational and project management, content management, recommendations, social networks, and more.

NEO4J EDITIONS

Neo4j's free *Community Edition* is a high-performance, fully ACID-transactional database. It includes (but is not limited to) all the functionality described in this Refcard.

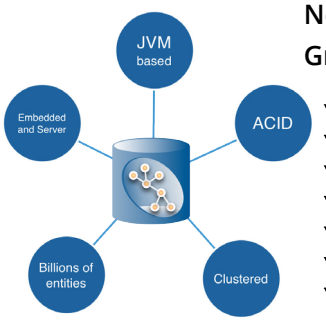
Neo4j's *Enterprise Edition* adds scalable clustering, fail-over, live backups, and comprehensive monitoring for production use.

More information about the Community and Enterprise editions is available at neo4j.com/product.

NEO4J SERVER

You can download Neo4j from neo4j.com/download. Neo4j Server can be installed and run on all operating systems. It provides an easy-to-use web interface at localhost:7474

The simplest way of getting started is to use Neo4j's database browser to execute your graph queries (written in Cypher, the graph query language described in this Refcard) in a workbench-like fashion. Results are rendered as either intuitive graph visualizations or as easy-to-read exportable tables.



Neo4j is a NoSQL Graph Database

- open source
- welcoming UI
- easy data modeling
- readable queries
- active community
- high performance
- optional schema

A remote Neo4j Server can be accessed via its Cypher HTTP API, either directly or through one of the many available language drivers. For especially high performance use cases, you can add Neo4j Server extensions in any JVM language to access Neo4j's internal database engine directly without network overhead.

GETTING STARTED WITH NEO4J

LOCAL INSTALLATION

Prerequisites: Java 7 or greater

- Visit neo4j.com/download and download Neo4j for your platform
- On Windows, run the installer, choose a directory and start the server
- On the other platforms, unzip the file and change to the directory in a terminal, e.g. `~/Download/neo4j-community-2.1.5`
- From the extracted directory, run `bin/neo4j` to start the server, which should then be up and listening on http://localhost:7474

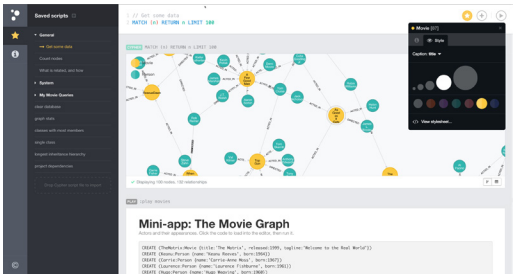
NEO4J IN THE CLOUD

In order to get started with Neo4j as a cloud hosted service, visit neo4j.com/developer/guide-cloud-deployment and choose a Neo4j Cloud Hosting provider that meets your needs. Most have free offerings to get started and provide provisioned Neo4j instances within minutes.

THE NEO4J BROWSER: GETTING STARTED

Once you have Neo4j running (accessible at localhost:7474 or on a remote server), open the Neo4j Browser and check out the Cypher Workbench. It provides guides to get you started, links to the manual and a sample movie database.

Neo4j Browser



your graph-query workbench

Let's add some data!

In the sidebar on the left, open the *Information tab* (the "i"), then choose the "Movie Graph App". You'll see a slide-show explaining the dataset of *Movie* and *Person* nodes connected via the *ACTED_IN* and *DIRECTED* relationships.

On the second slide, a large chunk of Cypher code contains the statements to create the dataset. Click on it to transfer the whole statement to the command line above.

Run the statement by hitting the "Execute" button on the top right. This inserts data into the database and renders a lonely node as a result.

Double click that lonely node and note how it expands into a star of other nodes, connected to the initial node by relationships.

Select any node and check the black property pop-up. Select its "eye" tab to customize node and relationship sizes, captions and colors. Select the *title* property as the caption for the movie nodes and *name* for the people. Continue to explore the graph visualization.

For a simple query that returns a *Person* node with its associated *Movie*s, run:

```
MATCH (p:Person {name:"Tom Hanks"})-->(m:Movie)
RETURN p,m
```

In the next step we're going to connect to the database.

CONNECTING TO NEO4J

Now that you have your server up and running, you can access it from any driver or just plain http. Enter `:GET /db/data/` into the browser console to see a representation of the management API. You can see a "transaction" entry representing the transactional HTTP endpoint that you can POST Cypher statements to. Assuming you have loaded the sample Movie dataset, try to execute:

```
:POST /db/data/transaction/commit
{"statements":[{"statement" :
"MATCH (m:Movie) RETURN m.title LIMIT {limit}",
"parameters":{"limit":10}}]}
```

With that simple request, you have effectively utilized the http endpoint used by many Neo4j drivers!

Head over to neo4j.com/developer/language-guides to find a suitable development guide and driver for your language-stack.

You've taken a small yet vital step on the path to your own Neo4j-powered application. With a basic understanding of Cypher, you can start to build your applications.

The next section introduces Neo4j's query language and provides a reference for the most frequently used clauses and operations.

CYPHER — NEO4J'S QUERY LANGUAGE

INTRODUCTION

When representing graphs on a whiteboard, we draw nodes as circles and relationships as arrows. Cypher was born out of graph patterns transformed from these two-dimensional whiteboard drawings into one-dimensional ASCII art. Cypher denotes nodes with round parentheses: `(a:Node)`, and relationships with labeled arrows: `-[:A_RELATIONSHIP]->`. Additional `{key:value}` structures represent properties for both.

Patterns of nodes and relationships are the building blocks for all Cypher queries.

As in most query languages, a Cypher statement is a series of clauses. The simplest statement consists of a `MATCH` or `CREATE` clause followed by a `RETURN` clause. Other clauses are borrowed from other query languages (like SQL): `WHERE`, `ORDER BY`, and `LIMIT SKIP`.

More advanced statements use node-relationship patterns as predicates or expressions. Try the following two Cypher statements in your Neo4j Browser:

```
CREATE (who:Person {name:"Me"})-[likes:LIKE]->
(what:Graph:Database {name:"Neo4j"})
RETURN who, likes, what;
```


```
MATCH (:Person {name:"Me"})-[:LIKE]->(what)
RETURN what.name AS What,
       count(*) as times AS Times;
```

WHAT	TIMES
Neo4j	1

If these clauses look familiar — especially if you're a SQL developer — that's great! Cypher is intended to be familiar enough to help you move rapidly along the learning curve. At the same time, it's tailored to query connected data in an expressive but still easy-to-understand fashion.

For live graph models using Cypher, check the Graph-Gists contributed by Neo4j users.

The following Cypher language reference details everything you need to know about the language and should equip you to write Cypher queries that help you express your intriguing business questions.

CYPHER REFERENCE

Note: `{value}` denotes either literals, for *ad hoc* Cypher queries; or parameters, for applications. Neo4j properties can be strings, numbers, or booleans, or arrays thereof.

READ QUERY STRUCTURE

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

MATCH

COMMAND	DESCRIPTION
`MATCH (n:Person)-[:KNOWS]->(m:Person) WHERE n.name="Alice"`	Node patterns can contain labels and are connected by relationships.
`MATCH (n)-->(m)`	Any pattern can be used in `MATCH`.
`MATCH (n:Person {name:"Alice"})-->(m)`	Patterns with node properties.
`MATCH p = (n)-->(m)`	Assign a path to p.
`OPTIONAL MATCH (n)-[r]->(m)`	Optional pattern, `NULL`s will be used for missing parts.

WHERE

COMMAND	DESCRIPTION
`WHERE node.property = {value}`	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.

RETURN

COMMAND	DESCRIPTION
`RETURN *`	Return the value of all named identifiers.
`RETURN n AS columnName`	Use alias for result column.
`RETURN DISTINCT n`	Return unique rows.
`ORDER BY n.property`	Sort the result.
`ORDER BY n.property DESC`	Sort the result in descending order.
`SKIP {skip_number}`	Skip a number of results.
`LIMIT {limit_number}`	Limit the number of results.
`SKIP {skip} LIMIT {limit}`	Skip results at the top and limit the number of results.
`RETURN count(*)`	The number of matching rows. See Aggregation for more.

WITH

COMMAND	DESCRIPTION
`MATCH (user)-[:KNOWS]-(friend) WHERE user.name = {name}`	`WITH` chains query parts. It allows you to specify which projection of your data is available after `WITH`.
`WITH user, count(*) AS friends`	You can also use `ORDER BY`, `SKIP`, `LIMIT` and aggregation with `WITH`.
`WHERE friends > 10 RETURN user`	You might have to alias expressions to give them a name.

UNION

COMMAND	DESCRIPTION

`MATCH (a)-[:KNOWS]->(b)`
`RETURN b.name`
`UNION`
`MATCH (a)-[:LOVES]->(b)`
`RETURN b.name`
	Returns the distinct union of all query results.
	Result column types and names have to match.

`MATCH (a)-[:KNOWS]->(b)`
`RETURN b.name`
`UNION ALL`
`MATCH (a)-[:LOVES]->(b)`
`RETURN b.name`
	Returns the union of all query results, including duplicated rows.

WRITE-ONLY QUERY STRUCTURE

```
(CREATE | MERGE)*
[SET|DELETE|REMOVE|FOREACH]*
[RETURN [ORDER BY] [SKIP] [LIMIT]]
```

READ-WRITE QUERY STRUCTURE

```
[[OPTIONAL] MATCH WHERE]
[WITH [ORDER BY] [SKIP] [LIMIT]]
(CREATE | MERGE)*
[SET|DELETE|REMOVE|FOREACH]*
[RETURN [ORDER BY] [SKIP] [LIMIT]]
```

CREATE

COMMAND	DESCRIPTION
`CREATE` `(n:Person {name: {value}})`	Create a node with the given properties.
`CREATE (n:Person {map})`	Create a node with the given properties.
`CREATE` `(n:Person {collOfMaps})`	Create nodes with the given properties.
`CREATE (n)-[r:KNOWS]->(m)`	Create a relationship with the given type and direction; bind an identifier to it.

`CREATE`
`(n)-[:LOVES since:{value}}]`
`->(m)`
	Create a relationship with the given type, direction, and properties.

MERGE

COMMAND	DESCRIPTION

`MERGE`
`(n:Person {name: {value}})`
`ON CREATE`
`SET n.created=timestamp()`
`ON MATCH`
`SET n.counter=`
`coalesce(n.counter,0)+1,`
`n.accessed=timestamp()`
	Match pattern and 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`
`(a)-[r:KNOWS]->(b:Person`
`{name: {value2}})`
	`MERGE` matches or creates whole subgraphs attached to the node.

SET

COMMAND	DESCRIPTION

`SET`
`n.property1 = {value1},`
`n.property2 = {value2}`
	Update or create a property

`SET n = {map}`	Set all properties. This will remove any existing properties.
`SET n += {map}`	Add and update properties, while keeping existing ones.
`SET n:Person`	Adds a label `Person` to a node.

DELETE

COMMAND	DESCRIPTION
`DELETE n, r`	Delete nodes and relationships.

REMOVE

COMMAND	DESCRIPTION
`REMOVE n:Person`	Remove a label from `n`.
`REMOVE n.property`	Remove a property.

INDEX

COMMAND	DESCRIPTION
`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 equality comparison. Note that for example `lower(n.name) = {value}` will not use an index.
`MATCH (n:Person)` `WHERE n.name IN {values}`	An index can be automatically used for the `IN` collection checks.

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

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

CONSTRAINT

COMMAND	DESCRIPTION

`CREATE CONSTRAINT`
`ON (p:Person)`
`ASSERT p.name IS UNIQUE`
	Create a unique constraint on the label `Person` and property `name`.
	
	If any other node with the label `Person` is updated or created with a value for `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`.

OPERATORS

TYPE	OPERATOR
Mathematical	`+, -, *, /, %, ^`
Comparison	`=, <>, <, >, <=, >=`
Boolean	`AND, OR, XOR, NOT`
String	`+`
Collection	`+, IN, [x], [x .. y]`
Regular Expression	`=~`

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`.
- Missing elements like a property that doesn't exist or accessing elements that don't exist in a collection yields `NULL`.
- In `OPTIONAL MATCH` clauses, `NULL`s will be used for missing parts of the pattern.

PATTERNS

COMMAND	DESCRIPTION
`(n)-->(m)`	A relationship from `n` to `m` exists.
`(n:Person)`	Matches nodes with the label `Person`.
`(n:Person:Swedish)`	Matches nodes that have both `Person` and `Swedish` labels.
`(n:Person {name: {value}})`	Matches nodes with the declared properties.
`(n:Person)-->(m)`	Node `n`, labeled `Person`, has a relationship to node `m`.
`(n)--(m)`	A relationship in any direction between `n` and `m`.
`(m)<-[:KNOWS]-(n)`	A relationship from `n` to `m` of type `KNOWS` exists.
`(n)-[:KNOWS|:LOVES]->(m)`	A relationship from `n` to `m` of type `KNOWS` or `LOVES` exists.
`(n)-[r]->(m)`	Bind an identifier to the relationship.
`(n)-[*1..5]->(m)`	Variable length paths can span 1 to 5 hops.
`(n)-[*]->(m)`	Variable length path of any depth. See performance tips.
`(n)-[:KNOWS]->(m:Label {property: {value}})`	Match or set properties in `MATCH`, `CREATE`, `CREATE UNIQUE` or `MERGE` clauses.
`shortestPath( (n1)-[*..6]-(n2))`	Find a single shortest path for previously matched nodes.
`allShortestPaths( (n1)-[*..6]-(n2))`	Find all shortest paths.

LABELS

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

COLLECTIONS

COMMAND	DESCRIPTION
`['a','b','c'] AS coll`	Literal collections are declared in square brackets.
`length({coll}) AS len, {coll}[0] AS value`	Collections can be passed in as parameters.
`range({from},{to},{step}) AS coll`	Range creates a collection of numbers (step is optional).
`MATCH (a)-[r:KNOWS*..5]->() RETURN r AS rels`	Relationship identifiers of a variable length path is a collection of relationships.
`RETURN node.coll[0] AS value, length(node.coll) AS len`	Properties can be arrays/ collections of strings, numbers or booleans.
`coll[{idx}] AS value, coll[{start}..{end}] AS slice`	Collection elements can be accessed with `idx` subscripts in square brackets. Invalid indexes return `NULL`. Slices can be retrieved with intervals from `start` to `end`, each of which can be omitted or negative. Out of range elements are ignored.
`UNWIND {names} AS name MATCH (n:Person {name:name}) RETURN avg(n.age)`	With `UNWIND`, you can transform any collection back into individual rows. The example matches all names from a list of names.

MAPS

COMMAND	DESCRIPTION
`{name:"Alice", age:38, address:{city:"London", residential:true}}`	Literal maps are declared in curly braces much like property maps. Nested maps and collections are supported.
`MERGE (p:Person{name:{map}.name}) ON CREATE SET p={map}`	Maps can be passed in as parameters and used as map or by accessing keys.
`range({start},{end},{step}) AS coll`	Range creates a collection of numbers (step is optional).
`MATCH (n:Person)-[r]->(m) RETURN n,r,m`	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.

RELATIONSHIP FUNCTIONS

COMMAND	DESCRIPTION
`type(rel)`	String representation of the relationship type.
`startNode(rel)`	Start node of the relationship.
`endNode(rel)`	End node of the relationship.
`id(rel)`	The internal id of the relationship.

PREDICATES

COMMAND	DESCRIPTION
`n.property <> {value}`	Comparison operators.
`has(n.property)`	Functions.
`n.number >= 1 AND n.number <= 10`	Boolean operators combine predicates.
`n:Person`	Check for node labels.
`identifier IS NULL`	Check if something is `NULL`.
`NOT has(n.property) OR n.property = {value}`	Either property does not exist or predicate is `TRUE`.
`n.property = {value}`	Non-existing property returns `NULL`, which is not equal to anything.
`n.property =~ "Tob.*"`	Regular expression.
`(n)-[:KNOWS]->(m)`	Make sure the pattern has at least one match.
`NOT (n)-[:KNOWS]->(m)`	Exclude nodes with certain pattern matches from the result.
`n.property IN [{value1}, {value2}]`	Check if an element exists in a collection.

COLLECTION PREDICATES

COMMAND	DESCRIPTION
`ALL(x IN coll WHERE has(x.property))`	Returns `TRUE` if the predicate is `TRUE` for all elements of the collection.
`ANY(x IN coll WHERE has(x.property))`	Returns `TRUE` if the predicate is `TRUE` for at least one element of the collection.
`NONE(x IN coll WHERE has(x.property))`	Returns `TRUE` if the predicate is `FALSE` for all elements of the collection.
`SINGLE(x IN coll WHERE has(x.property))`	Returns `TRUE` if the predicate is `TRUE` for exactly one element in the collection.

FUNCTIONS

COMMAND	DESCRIPTION
`coalesce(n.property,..., {defaultValue})`	The first non-`NULL` expression.
`timestamp()`	Milliseconds since midnight, January 1, 1970 UTC.
`id(node_or_relationship)`	The internal id of the relationship or node.
`toInt({expr})`	Converts the given input in an integer if possible; otherwise it returns `NULL`.
`toFloat({expr})`	Converts the given input in a floating point number if possible; otherwise it returns `NULL`.

PATH FUNCTIONS

COMMAND	DESCRIPTION
`length(path)`	The length of the path.
`nodes(path)`	The nodes in the path as a collection.
`relationships(path), rels(path)`	The relationships in the path as a collection.
`MATCH path=(n)-->(m) RETURN extract( x IN nodes(path) | x.prop)`	Assign a path and process its nodes.
`MATCH path=(n1)-[*]->(n2) FOREACH (n IN rels(path) | SET n.marked = TRUE)`	Execute an update operation for each relationship of a path.

COLLECTION FUNCTIONS

COMMAND	DESCRIPTION
`length({coll})`	Length of the collection.
`head({coll}), last({coll}), tail({coll})`	Head returns the first, last the last element, and tail the remainder of the collection. All return `NULL` for an empty collection.
`[x IN coll WHERE has(x.prop) | x.prop]`	Combination of filter and extract in a concise notation.
`extract(x IN coll | x.prop)`	A collection of the value of the expression for each element in the original collection.
`filter(x IN coll WHERE x.prop <> {value})`	A filtered collection of the elements where the predicate is TRUE.
`reduce(s = 0, x IN coll | s + x.prop)`	Evaluate expression for each element in the collection, accumulate the results.
`FOREACH (value IN coll | CREATE (:Person{name:value}))`	Execute an update operation for each element in a collection.

MATHEMATICAL FUNCTIONS

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

STRING FUNCTIONS

COMMAND	DESCRIPTION
`toString({expression})`	String representation of the expression.
`replace({string}, {search}, {replacement})`	Replace all occurrences of search with replacement. All arguments are be expressions.
`substring({string}, {begin}, {len})`	Get part of a string. The `len` argument is optional.
`left({string}, {len}), right({string}, {len})`	The first part of a string. The last part of the string.
`trim({string}) ltrim({string}) rtrim({string})`	Trim all whitespace, or on left or right side.
`upper({string}), lower({string})`	UPPERCASE and lowercase.
`split({string}, {delim})`	Split a string into a collection of strings.

AGGREGATION

COMMAND	DESCRIPTION
`count(*)`	The number of matching rows.
`count(identifier)`	The number of non-`NULL` values.
`count(DISTINCT identifier)`	All aggregation functions also take the `DISTINCT` modifier, which removes duplicates from the values.
`collect(n.property)`	Value collection, 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`.

CASE

COMMAND	DESCRIPTION
`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.

START

COMMAND	DESCRIPTION
`START n = node:indexName(key={value}) n=node:nodeIndexName(key={value}) n=node:nodeIndexName(key={value})`	Query the index with an exact query. Use node_auto_index for the old automatic index.
`START n = node:indexName({query})`	Query the index by passing the query string directly, can be used with lucene or spatial syntax. E.g.: `"name:Jo*"` or `"withinDistance:[60,15,100]"`

UPGRADING

In Neo4j 2.0, several Cypher features from version 1.9 have been deprecated or removed:

- `START` is optional.
- `MERGE` takes `CREATE UNIQUE`'s role for the unique creation of patterns. Note that they are not the same.
- Optional relationships are handled by `OPTIONAL MATCH`, not question marks.
- Non-existing properties return `NULL`: `n.prop?` and n.prop! have been removed.
- The separator for collection functions changed from ":" to "|".
- Paths are no longer collections, use `nodes(path)` or `rels(path)`.
- Parentheses around nodes in patterns are no longer optional.
- `CREATE a={property:'value'}` has been removed
- Use `REMOVE` to remove properties.
- Parameters for index-keys and nodes in patterns are no longer allowed.
- To use the older syntax, prepend your Cypher statement with `CYPHER 1.9`.

PERFORMANCE TIPS

- Use parameters instead of literals when possible. This allows Cypher to reuse 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 touch all nodes in a graph by mistake.
- Return only the data you need. Avoid returning whole nodes and relationships — instead, return only the properties you need.

USE CASE: RECOMMENDATIONS

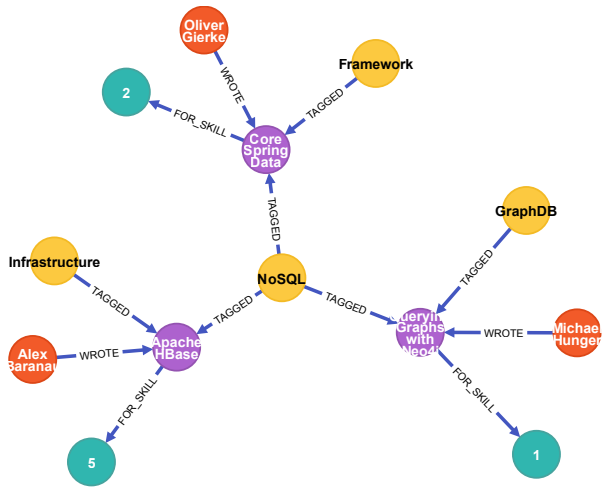
Recommendations in Neo4j are both powerful and easy to implement. You can recommend anything, including friends, music, products, places, books, jobs, travel-connections ... even Refcardz.

As you're reading one, let's take a Refcard collection as an example for a tiny recommendation system. You can find more recommendation examples online.

In our example domain we have :Refcard nodes, which have a title and a lot of content, and connected `:Author`, `:Topic`, and `:Skill` nodes.

To create three entries in our database, we would execute this statement:

```
CREATE (:Author {name:"Michael Hunger"})
       -[:WROTE]->(neo4j:Refcard {title:"Querying Graphs Neo4j"})
       -[:FOR_SKILL]->(:Skill {level:1}),
       (neo4j)<-[:TAGGED]-(nosql:Topic {name:"NoSQL"}),
       (neo4j)<-[:TAGGED]-(:Topic {name:"GraphDB"})
CREATE (:Author {name:"Oliver Gierke"})
       -[:WROTE]->(springData:Refcard {title:"Core Spring Data"})
       -[:FOR_SKILL]->(:Skill {level:2}),
       (springData)<-[:TAGGED]-(:Topic {name:"Framework"}),
       (springData)<-[:TAGGED]-(nosql)
CREATE (:Author {name:"Alex Baranau"})
       -[:WROTE]->(hbase:Refcard {title:"Apache HBase"})
       -[:FOR_SKILL]->(:Skill {level:5}),
       (hbase)<-[:TAGGED]-(:Topic {name:"Infrastructure"}),
       (hbase)<-[:TAGGED]-(nosql)
```



Now we can run a simple query that asks the following question: "I really liked the Core Spring Data Refcard. Matching my reading history and skills, what other Refcardz should I read?"

```
MATCH (ref1:Refcard {title:"Core Spring Data"})
      <-[:TAGGED]-(t)-[:TAGGED]->(ref2:Refcard)<-[:WROTE]-(author)
MATCH (ref1)-[:FOR_SKILL]->(skill1), (ref2)-[:FOR_SKILL]->(skill2)
WHERE abs(skill1.level-skill2.level) < 2
RETURN ref2.title AS Title, author.name AS Author,
       count(*) as Score, collect(DISTINCT t.name) as Topics
ORDER BY Score DESC
```

TITLE	AUTHOR	SCORE	TOPICS
Querying Graphs with Neo4j	Michael Hunger	1	[NoSQL]

That's it! Based on your previous reading habits, the database suggests you read the "Querying Graphs with Neo4j" Refcard if you haven't done so already, because its skill level is similar to that of the Core Spring Data Refcard and it shares a topic: NoSQL.

Explore this example further in one of our live graph gist presentations.

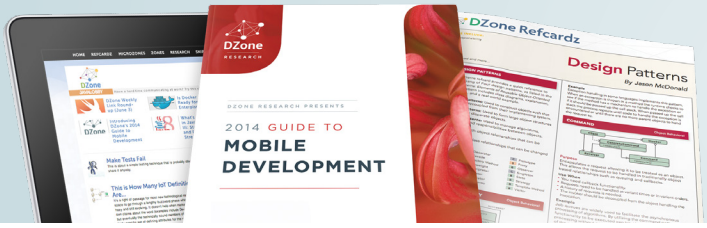
ABOUT THE AUTHOR

Michael Hunger has been passionate about software development for a long time. He is particularly interested in the people who develop software and making them successful by coding, writing, and speaking. For the last few years he has been working at Neo Technology on the open source Neo4j graph database (http://neo4j.com). There, Michael is involved in many things but focuses on developer evangelism in the great Neo4j community and leading the Spring Data Neo4j project (http://projects.spring.io/spring-data-neo4j).

RECOMMENDED BOOK

Graphs model the real world beautifully. But in a graph database, the absence of a natural schema can make both design and implementation difficult. *Graph Databases* helps you solve graph modeling and querying problems by walking through every aspect of graph database development, from basics of graph theory to predictive analysis with graphs to graph database internals.

FREE DOWNLOAD



BROWSE OUR COLLECTION OF **250+ FREE RESOURCES**, INCLUDING:

RESEARCH GUIDES: Unbiased insight from leading tech experts

REFCARDZ: Library of 200+ reference cards covering the latest tech topics

COMMUNITIES: Share links, author articles, and engage with other tech experts

JOIN NOW

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795

Version 1.0 $7.95