Neo4j Cypher Cheat Sheet

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

- 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 mutate graph data by creating, updating, and removing nodes, relationships, and properties.

You can try cypher snippets live in the Neo4j Console at
http://console.neo4j.org

Read-Only Query Structure

```
START me=node:people(name='Andres')
[MATCH me-[:FRIEND]->friend ]
WHERE friend.age > 18
RETURN me, friend.name
ORDER BY friend.age asc
SKIP 5 LIMIT 10
```

START	*meaning*
START n=node(*id,[id2, id3]*)	Load the node with id id into *n*
START n=node:indexName (*key*="value")	Query the index with an exact query and put the result into *n* Use node_auto_index for the auto-index
START n=node:indexName ("lucene query")	Query the index using a full Lucene query and put the result in *n*
START n=node(*)	Load all nodes
START m=node(1), n=node(2)	Multiple start points

RETURN	*meaning*
RETURN *	Return all named nodes, relationships and identifiers
RETURN *expr* AS *alias*	Set result column name as alias
RETURN distinct *expr*	Return unique values for *expr*

MATCH	*meaning*
MATCH n-->m	A pattern where *n* has outgoing relationships to another node, no matter relationship-type
MATCH n--m	*n* has relationship in either direction to *m*
MATCH n-[:KNOWS]->m	The outgoing relationship between n and m has to be of *KNOWS* relationship type
MATCH n-[:KNOWS|LOVES]-m	*n* has *KNOWS* or *LOVES* relationship to *m*
MATCH n-[r]->m	An outgoing relationship from *n* to *m*, and store the relationship in *r*
MATCH n-[r?]->m	The relationship is optional
MATCH n-[*1..5]->m	A multi step relationship between between *n* and *m*, one and five steps away
MATCH n-[*]->m	A pattern where *n* has a relationship to *m* unbound number of steps away
MATCH n-[?:KNOWS*..5]->m	An optional relationship between *n* and *m* that is of *KNOWS* relationship type, and between one and five steps long.
MATCH n-->m<--o	A pattern with *n* having an outgoing relationship to *m*, and *m* having incoming relationship from *o*
MATCH p=n-->m<--o	Store the path going from *n* to *o* over *m* into the path identifier *p*
MATCH p = shortestPath(n-[:KNOWS*3]->m)	Find the shortest path between *n* and *m* of type *KNOWS* of at most length 3

Read-Write-Return Query Structure

```
START emil=node:people(name='Emil')
MATCH emil-[:MARRIED_TO]-madde
CREATE/CREATE UNIQUE
emil-[:DAD]->(noomi {name:"Noomi"})<-[:MOM]-madde
DELETE emil.spare_time
SET emil.happy=true
RETURN noomi
```

CREATE	*meaning*
CREATE (n { name :"Name" })	Creates the node with the given properties
CREATE n = {map}	Create node from *map* parameter
CREATE n = {manyMaps}	Create many nodes from parameter with *coll* of maps
CREATE n-[:KNOWS]->m	Creates the relationship with the given type and dir
CREATE n-[:LOVES {since: 2007}] ->m	Creates the relationship with the given type, dir, and properties

DELETE	*meaning*
DELETE n, DELETE rel	Deletes the node, relationship
DELETE n.prop	Removes the property

CREATE UNIQUE	*meaning*
CREATE UNIQUE n-[:KNOWS]->m	Tries to match the pattern. Creates the missing pieces if the match fails
CREATE UNIQUE n-[:KNOWS]->(m {name:"Name"})	Tries to match a node with the property *name* set to "Name". Creates the node and sets the property if it can't be found.
CREATE UNIQUE n-[:LOVES {since: 2007}] ->m	Tries to find the relationship with the given type, direction, and attributes. Creates it if not found.

SET	*meaning*
SET n.prop = value	Updates or creates the property prop with the given *value*
SET n = {map}	Updates the properties with the given *map* parameter
SET n.prop = null	Deletes the property *prop*

Predicates	*meaning*
NOT *pred1* AND/OR *pred2*	Boolean operators for predicates
ALL(*x in coll*: *pred*)	TRUE if *pred* is TRUE for all values in *coll*
ANY(*x in coll* : *pred*)	TRUE if *pred* is TRUE for at least one value in *coll*
NONE(*x in coll* : *pred*)	TRUE if *pred* returns FALSE for all values in *coll*
SINGLE(*x in coll* : *pred*)	TRUE if *pred* returns TRUE for a single value in *coll*
identifier IS NULL	TRUE if *identifier* is <NULL>
n.prop? = value	TRUE if *n.prop = value* or n is NULL or n.prop does not exist
n.prop! = value	TRUE if *n.prop = value*, FALSE if *n* is NULL or *n.prop* does not exist
n =~ /regexp/	Regular expression
e1 <> e2 e1 < e2 e1 = e2	Comparison operators
has(n.prop)	Checks if property exists
n-[:TYPE]->m	Filter on existence of relationship
expr IN *coll*	Checks for existence of *expr* in *coll*

Expressions	*meaning*
a-zA-Z0-9_ or 'some na-me'	Allowed identifier (or quoted)
n + / - * % m	Arithmetic operators "+" also works on strings and collections
n.prop, n.prop?	Property on node, property on node, or NULL if missing
[42,"Hello",'World',{p}]	A collection
{param}	Parameter value, passed into the query execution as map { param : "value",... }
a-->()<--b	A path-pattern

Functions	*meaning*
HEAD(*coll*)	First element of *coll*
TAIL(*coll*)	*coll* except first element
LAST(*coll*)	Last element of *coll*
TYPE(*rel*)	Relationship type of *rel*
ID(*node*) ID(*relationship*)	Id of *node* or *relationship*
COALESCE(*expr*,*default*)	Returns *default* if *expr* is NULL otherwise *expr*
RANGE(*start*,*end*[,*step*])	Creates a range from *start* to *end* (inclusive) with a optional *step*
ABS(*v*) ROUND(*v*) SQRT(*v*) SIGN(*v*)	Math functions

Path Functions	*meaning*
NODES(*path*)	Returns the nodes in *path*
RELS(*path*)	Returns the relationships in *path*
LENGTH(*path*)	Returns the length of *path*

Aggregate Functions	*meaning*
	NULL values in *expr*
COUNT(*)	Returns the number of values aggregated over
SUM(*expr*)	Returns the sum of all values in *expr* Throws exception for non-numeric values
AVG(*expr*)	Returns the average of all values in *expr*
MAX(*expr*)	Returns the largest value in *expr*
MIN(*expr*)	Returns the smallest values in *expr*
COLLECT(*expr*)	Returns an *coll* containing all values in *expr*
FILTER(*x* in *coll* : *predicate*)	Returns a all the elements in *coll* that match the given *predicate*
EXTRACT(*x* in *coll* : *expr*)	Applies the *expr* once for every element in *coll*

FOREACH

FOREACH is used to execute a mutating operation for each element of a collection, e.g. creating a node for each element
using the element as an attribute value.

```
START user=node:users("name:A*"),
promotion=node(...)
MATCH user-[:FRIEND]-friend-[:FRIEND]-foaf
WITH user, collect(distinct foaf) as new_friends
```

WITH

WITH syntax is similar to RETURN. It separates query parts explicitly, allowing you to declare which identifiers to carry over to the next part. This can be used to limit the visible identifiers but mostly for creating aggregate values that can be used in the next query part either for filtering (implementing HAVING) or for the creation of new structures in the graph.

WITH also creates a boundary between reading and updating query parts so that they don't interfere.

```
START user=node:users("name:A*")
MATCH user-[:FRIEND]-friend
WITH user, count(friend) as friends
WHERE friends > 10
RETURN user

START user=node:users("name:A*")
MATCH user-[:FRIEND]-friend
WITH user, count(friend) as friends
SET user.numberOfFriends = friends
```

Transactions

The Neo4j-Shell supports commands to begin transactions, which allows you issue multiple commands and then only commit them when you're satisfied and rollback if you ran into an issue or don't want your changes to happen.

```
neo4j-sh (0)$ begin
==> Transaction started
neo4j-sh (0)$ rollback
==> Transaction rolled back
neo4j-sh (0)$ commit
==> Transaction committed
```

Useful Snippets

`START n=node(...)`
`MATCH n-->m-->o`
`WHERE not ( n-->o )`
`RETURN o`
	Not already connected to
	
	This returns nodes that *m* is connected to, that *n* is not already connected to.

`START n=node(...)`
`MATCH path = n-[*]-n`
`RETURN n, length(path)`
	Find cycles
	
	This returns nodes that *m* is connected to, that *n* is not already connected to.

`START n=node(...)`
`MATCH n-[r]-m`
`RETURN type(r), count(*)`
	Group count relationship types
	
	Returns a count of each of the relationship-types.

`START n=node(...)`
`MATCH n-[r?]-()`
`DELETE n,r`
	Delete node with relationships
	
	Finds the node and all relationships (if any) and deletes node and relationships.

`START n = node(1), m = node(2) RETURN n.name +" and "+ m.name`	***String concat on expressions***

Useful Links

Cypher Screencast
http://bit.ly/cypher-stanley

Cypher Reference Manual
http://bit.ly/cypher-reference

Cypher Presentation
http://bit.ly/cypher-slide