**CYPHER CODE  
  
Cypher Reference Card:** [**Neo4j Cypher Refcard 4.4**](https://neo4j.com/docs/cypher-refcard/current/)

**Cypher Manual:** [**The Neo4j Cypher Manual v4.4 - Neo4j Cypher Manual**](https://neo4j.com/docs/cypher-manual/current/) **Retrieve Nodes**

// Return all Person node types

MATCH (p:Person)

RETURN p  
  
// Return node for Tom  
MATCH (p:Person {name: 'Tom Hanks'})

RETURN p  
  
// Return nodes properties born for Tom  
MATCH (p:Person {name: 'Tom Hanks'})

RETURN p.born  
  
// Return nodes properties name and born for Tom or Rita  
MATCH (p:Person)

WHERE p.name = 'Tom Hanks' OR p.name = 'Rita Wilson'

RETURN p.name, p.born  
  
**Traverse Relationships**  
// This code returns the titles of all moves that Tom Hanks acted in.   
MATCH (p:Person {name: 'Tom Hanks'})-[:ACTED\_IN]->(m:Movie)

RETURN m.title  
  
// This query retrieves the Person nodes and Movie nodes where the person acted in a movie that was released in 2008 or 2009  
MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE m.released = 2008 OR m.released = 2009

RETURN p, m  
  
**Filtering Queries**

// We want to retrieve Person nodes of people who acted in movies released between 2000 and 2003  
MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE 2000 <= m.released <= 2003

RETURN p.name, m.title, m.released  
  
// Return Movie nodes where Jack Nicholson acted in the movie, and the movie has the tagline property  
MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name='Jack Nicholson' AND m.tagline IS NOT NULL

RETURN m.title, m.tagline  
  
// Find all actors in the graph whose first name is Michael  
MATCH (p:Person)-[:ACTED\_IN]->()

WHERE p.name STARTS WITH 'Michael'

RETURN p.name  
  
// String tests are case-sensitive so you may need to use the toLower() or toUpper() functions to ensure the test yields the correct results  
MATCH (p:Person)-[:ACTED\_IN]->()

WHERE toLower(p.name) STARTS WITH 'michael'

RETURN p.name  
  
// find all people who wrote a movie but did not direct that same movie  
MATCH (p:Person)-[:WROTE]->(m:Movie)

WHERE NOT exists( (p)-[:DIRECTED]->(m) )

RETURN p.name, m.title  
  
// retrieve Person nodes of people born in 1965, 1970, or 1975  
MATCH (p:Person)

WHERE p.born IN [1965, 1970, 1975]

RETURN p.name, p.born  
  
// return the name of the actor who played Neo in the movie The Matrix  
MATCH (p:Person)-[r:ACTED\_IN]->(m:Movie)

WHERE 'Neo' IN r.roles AND m.title='The Matrix'

RETURN p.name, r.roles  
  
// Discover the properties for the Person nodes in the graph  
MATCH (p:Person)

RETURN p.name, keys(p)  
  
// Discover the properties defined in the graph  
CALL db.propertyKeys()  
  
**Creating nodes**// create a node to represent Michael Cain  
MERGE (p:Person {name: 'Michael Cain'})

// Verify the node was created  
MATCH (p:Person {name: 'Michael Cain'})

RETURN p  
  
// Create multiple merge clauses  
MERGE (p:Person {name: 'Katie Holmes'})

MERGE (m:Movie {title: 'The Dark Knight'})

RETURN p, m  
  
**Creating relationship between two nodes**  
// if the Person and Movie nodes both already exist, we can find them using a MATCH clause before creating the relationship between them.  
  
MATCH (p:Person {name: 'Michael Cain'})

MATCH (m:Movie {title: 'The Dark Knight'})

MERGE (p)-[:ACTED\_IN]->(m)  
  
// We can confirm that this relationship exists, you don’t need to worry about specifying the direction of a match. It will look for all nodes that are connected.  
  
MATCH (p:Person {name: 'Michael Cain'})-[:ACTED\_IN]-(m:Movie {title: 'The Dark Knight'})

RETURN p, m  
  
**Creating nodes and relationships using multiple clauses**

// Chaining multiple MERGE clauses together. If you don’t define the direction of the relationship, Neo4j will assume a left-to-right relationship.

MERGE (p:Person {name: 'Chadwick Boseman'})

MERGE (m:Movie {title: 'Black Panther'})

MERGE (p)-[:ACTED\_IN]-(m)

**Creating nodes and relationships in single clause**

MERGE (p:Person {name: 'Emily Blunt'})-[:ACTED\_IN]->(m:Movie {title: 'A Quiet Place'})

RETURN p, m

**Adding properties to a node or relationship**

// Inline as part of a merge clause  
MERGE (p:Person {name: 'Michael Cain'})

MERGE (m:Movie {title: 'Batman Begins'})

MERGE (p)-[:ACTED\_IN {roles: ['Alfred Penny']}]->(m)

RETURN p,m  
  
// Using set keyword as a reference to a node or relationship  
MATCH (p:Person)-[r:ACTED\_IN]->(m:Movie)

WHERE p.name = 'Michael Cain' AND m.title = 'The Dark Knight'

SET r.roles = ['Alfred Penny']

RETURN p, r, m

// Setting multiple properties  
MATCH (p:Person)-[r:ACTED\_IN]->(m:Movie)

WHERE p.name = 'Michael Cain' AND m.title = 'The Dark Knight'

SET r.roles = ['Alfred Penny'], r.year = 2008

RETURN p, r, m  
  
  
// Updating properties – changing Alfred Penny to Mr. Alfred Penny

MATCH (p:Person)-[r:ACTED\_IN]->(m:Movie)

WHERE p.name = 'Michael Cain' AND m.title = 'The Dark Knight'

SET r.roles = ['Mr. Alfred Penny']

RETURN p, r, m

// Removing properties  
MATCH (p:Person)-[r:ACTED\_IN]->(m:Movie)

WHERE p.name = 'Michael Cain' AND m.title = 'The Dark Knight'

REMOVE r.roles

RETURN p, r, m

**Merging with relationships**

// Checks if person node for a value exists and if not it creates it and the createdAt property is set. If the node is found, then the updatedAt property is set. In both cases, the born property is set.  
  
// Find or create a person with this name

MERGE (p:Person {name: 'McKenna Grace'})

// Only set the `createdAt` property if the node is created during this query

ON CREATE SET p.createdAt = datetime()

// Only set the `updatedAt` property if the node was created previously

ON MATCH SET p.updatedAt = datetime()

// Set the `born` property regardless

SET p.born = 2006

RETURN p  
  
// Setting multiple properties ON CREATE SET  
ON CREATE SET m.released = 2020, m.tagline = `A great ride!'  
  
// Using MERGE to create nodes or relationships  
MERGE (p:Person {name: 'Michael Cain'})-[:ACTED\_IN]->(m:Movie {title: 'The Cider House Rules'})

RETURN p, m

**Deleting Data**// Delete a node for Person Jane Doe  
MATCH (p:Person)

WHERE p.name = 'Jane Doe'

DELETE p  
  
// Delete a relationship – remove relationship of Jane Doe acting in The Matrix, but do not delete Jane Doe or The Matrix as nodes in the graph. If we attempted to delete the Jane Doe node, we will receive an error because it has relationships in the graph.  
  
MATCH (p:Person {name: 'Jane Doe'})-[r:ACTED\_IN]->(m:Movie {title: 'The Matrix'})

DELETE r

RETURN p, m

// Deleting a node and its relationships

MATCH (p:Person {name: 'Jane Doe'})

DETACH DELETE p  
  
// Deleting all nodes and relationships in a graph  
MATCH (n)

DETACH DELETE n  
  
// Deleting labels – this removes the label called Developer for the node Jane Doe.  
MATCH (p:Person {name: 'Jane Doe'})

REMOVE p:Developer

RETURN p  
  
// Return all node lables defined in the graph  
CALL db.labels()  
  
MATCH (n) DETACH DELETE n;

// Create nodes and properties

MERGE (:Movie {title: 'Apollo 13', tmdbId: 568, released: '1995-06-30', imdbRating: 7.6, genres: ['Drama', 'Adventure', 'IMAX']})

MERGE (:Person {name: 'Tom Hanks', tmdbId: 31, born: '1956-07-09'})

MERGE (:Person {name: 'Meg Ryan', tmdbId: 5344, born: '1961-11-19'})

MERGE (:Person {name: 'Danny DeVito', tmdbId: 518, born: '1944-11-17'})

MERGE (:Person {name: 'Jack Nicholson', tmdbId: 514, born: '1937-04-22'})

MERGE (:Movie {title: 'Sleepless in Seattle', tmdbId: 858, released: '1993-06-25', imdbRating: 6.8, genres: ['Comedy', 'Drama', 'Romance']})

MERGE (:Movie {title: 'Hoffa', tmdbId: 10410, released: '1992-12-25', imdbRating: 6.6, genres: ['Crime', 'Drama']})  
  
MATCH (n) RETURN n  
  
// Adding new nodes and properties  
MERGE (u:User {userId: 534, name: 'Sandy Jones'})

MERGE (u:User {userId: 105, name: 'Clinton Spencer'})  
  
// Creating relationship properties  
MATCH (apollo:Movie {title: 'Apollo 13'})

MATCH (tom:Person {name: 'Tom Hanks'})

MATCH (meg:Person {name: 'Meg Ryan'})

MATCH (danny:Person {name: 'Danny DeVito'})

MATCH (sleep:Movie {title: 'Sleepless in Seattle'})

MATCH (hoffa:Movie {title: 'Hoffa'})

MATCH (jack:Person {name: 'Jack Nicholson'})

// create the relationships between nodes

MERGE (tom)-[:ACTED\_IN {role: 'Jim Lovell'}]->(apollo)

MERGE (tom)-[:ACTED\_IN {role: 'Sam Baldwin'}]->(sleep)

MERGE (meg)-[:ACTED\_IN {role: 'Annie Reed'}]->(sleep)

MERGE (danny)-[:ACTED\_IN {role: 'Bobby Ciaro'}]->(hoffa)

MERGE (danny)-[:DIRECTED]->(hoffa)

MERGE (jack)-[:ACTED\_IN {role: 'Jimmy Hoffa'}]->(hoffa)  
  
// Adding more properties to relationships between nodes  
MATCH (sandy:User {name: 'Sandy Jones'})

MATCH (clinton:User {name: 'Clinton Spencer'})

MATCH (apollo:Movie {title: 'Apollo 13'})

MATCH (sleep:Movie {title: 'Sleepless in Seattle'})

MATCH (hoffa:Movie {title: 'Hoffa'})

MERGE (sandy)-[:RATED {rating:5}]->(apollo)

MERGE (sandy)-[:RATED {rating:4}]->(sleep)

MERGE (clinton)-[:RATED {rating:3}]->(apollo)

MERGE (clinton)-[:RATED {rating:3}]->(sleep)

MERGE (clinton)-[:RATED {rating:3}]->(hoffa)

// How many people rated a movie test case  
MATCH (u:User)-[:RATED]-(m:Movie)

WHERE m.title = 'Apollo 13'

RETURN count(\*) AS `Number of reviewers`  
  
// Youngest actor test case  
MATCH (p:Person)-[:ACTED\_IN]-(m:Movie)

WHERE m.title = 'Hoffa'

RETURN p.name AS Actor, p.born as `Year Born` ORDER BY p.born DESC LIMIT 1  
  
// Highest rated movie in a particular year test case  
MATCH (m:Movie)

WHERE m.released STARTS WITH '1995'

RETURN m.title as Movie, m.imdbRating as Rating ORDER BY m.imdbRating DESC LIMIT 1  
  
// Test case – what drama movies did an actor act in   
MATCH (p:Person)-[:ACTED\_IN]-(m:Movie)

WHERE p.name = 'Tom Hanks' AND

'Drama' IN m.genres

RETURN m.title AS Movie

// Profiling a query for performance  
PROFILE MATCH (p:Person)-[:ACTED\_IN]-()

WHERE p.born < '1950'

RETURN p.name  
  
// Adding an additional label to nodes  
MATCH (p:Person)

WHERE exists ((p)-[:ACTED\_IN]-())

SET p:Actor

// Turning language properties into Language nodes  
MATCH (m:Movie)

UNWIND m.languages AS language

WITH language, collect(m) AS movies

MERGE (l:Language {name:language})

WITH l, movies

UNWIND movies AS m

WITH l,m

MERGE (m)-[:IN\_LANGUAGE]->(l);

MATCH (m:Movie)

SET m.languages = null

// Refactoring to create specialised relationships – eg ACTED\_IN\_1995  
MATCH (n:Actor)-[:ACTED\_IN]->(m:Movie)

CALL apoc.merge.relationship(n,

'ACTED\_IN\_' + left(m.released,4),

{},

{},

m ,

{}

) YIELD rel

RETURN count(\*) AS `Number of relationships merged`;  
  
// Return actor or director from specialised relationships  
MATCH (p:Person)-[:ACTED\_IN\_1995|DIRECTED\_1995]->()

RETURN p.name as `Actor or Director`  
  
**Adding Intermediate Nodes**// Find an actor that acted in a Movie

MATCH (a:Actor)-[r:ACTED\_IN]->(m:Movie)

// Create a Role node

MERGE (x:Role {name: r.role})

// Create the PLAYED relationship

// relationship between the Actor and the Role nodes.

MERGE (a)-[:PLAYED]->(x)

// Create the IN\_MOVIE relationship between

// the Role and the Movie nodes.

MERGE (x)-[:IN\_MOVIE]->(m)

**Reading in CSV data**LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing/ratings.csv'

AS row

RETURN count(row)  
  
// Check how many nodes generated  
MATCH (n) RETURN count(n)  
  
// Check how many relationships created  
MATCH ()-[r]->() RETURN count(r)  
  
**Converting from String to Date**MATCH (p:Person)

SET p.born = CASE p.born WHEN "" THEN null ELSE date(p.born) END

WITH p

SET p.died = CASE p.died WHEN "" THEN null ELSE date(p.died) END  
  
**Confirm Node Properties**CALL apoc.meta.nodeTypeProperties()

YIELD nodeType, propertyName, propertyTypes  
  
**Confirm Relationship Properties**CALL apoc.meta.relTypeProperties()

YIELD relType, propertyName, propertyTypes

**Transforming List Properties**MATCH (m:Movie)

SET m.countries = split(coalesce(m.countries,""), "|"),

m.languages = split(coalesce(m.languages,""), "|"),

m.genres = split(coalesce(m.genres,""), "|")  
  
**Adding Labels to Nodes**// Add Actor label to all nodes that have the ACTED\_IN relationshipMATCH (p:Person)-[:ACTED\_IN]->()

WITH DISTINCT p SET p:Actor  
  
**Create a uniqueness constraint**// Ensure Name property in Genre nodes are unique  
CREATE CONSTRAINT Genre\_name ON (g:Genre) ASSERT g.name IS UNIQUE  
  
**Creating new nodes from the property of current nodes**MATCH (m:Movie)

UNWIND m.genres AS genre

WITH m, genre

MERGE (g:Genre {name:genre})

MERGE (m)-[:IN\_GENRE]->(g)  
  
**View Graph Schema**CALL db.schema.visualization  
  
**Delete all nodes and relatiomships in a graph**MATCH (u:User) DETACH DELETE u;

MATCH (p:Person) DETACH DELETE p;

MATCH (m:Movie) DETACH DELETE m;

MATCH (n) DETACH DELETE n

**Importing Movie and Genre data**// Only load first 10 rows  
LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing/2-movieData.csv'

AS row

//process only Movie rows

WITH row WHERE row.Entity = "Movie"

RETURN

row.tmdbId,

row.imdbId,

toFloat(row.imdbRating),

row.released,

row.title,

toInteger(row.year),

row.poster,

toInteger(row.runtime),

split(coalesce(row.countries,""), "|"),

toInteger(row.imdbVotes),

toInteger(row.revenue),

row.plot,

row.url,

toInteger(row.budget),

split(coalesce(row.languages,""), "|"),

split(coalesce(row.genres,""), "|")

LIMIT 10  
  
**Importing Movie and Genre data with :auto periodic commit**:auto USING PERIODIC COMMIT

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing/2-movieData.csv'

AS row

//process only Movie rows

WITH row WHERE row.Entity = "Movie"

MERGE (m:Movie {movieId: toInteger(row.movieId)})

ON CREATE SET

m.tmdbId = toInteger(row.tmdbId),

m.imdbId = toInteger(row.imdbId),

m.imdbRating = toFloat(row.imdbRating),

m.released = row.released,

m.title = row.title,

m.year = toInteger(row.year),

m.poster = row.poster,

m.runtime = toInteger(row.runtime),

m.countries = split(coalesce(row.countries,""), "|"),

m.imdbVotes = toInteger(row.imdbVotes),

m.revenue = toInteger(row.revenue),

m.plot = row.plot,

m.url = row.url,

m.budget = toInteger(row.budget),

m.languages = split(coalesce(row.languages,""), "|")

WITH m,split(coalesce(row.genres,""), "|") AS genres

UNWIND genres AS genre

WITH m, genre

MERGE (g:Genre {name:genre})

MERGE (m)-[:IN\_GENRE]->(g)  
**Importing to create relationship in the graph between Person and Movie**// Create ACTED\_IN relationship  
:auto USING PERIODIC COMMIT

LOAD CSV WITH HEADERS

FROM 'https://data.neo4j.com/importing/2-movieData.csv'

AS row

WITH row WHERE row.Entity = "Join" AND row.Work = "Acting"

MATCH (p:Person {tmdbId: toInteger(row.tmdbId)})

MATCH (m:Movie {movieId: toInteger(row.movieId)})

MERGE (p)-[r:ACTED\_IN]->(m)

ON CREATE

SET r.role = row.role

SET p:Actor  
  
**View property types for nodes**CALL db.schema.nodeTypeProperties()  
  
**View property types for relationships**CALL db.schema.relTypeProperties()  
  
**Show uniqueness constraint indexes in the graph**SHOW CONSTRAINTS  
  
**Testing equality**MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name = 'Tom Hanks'

AND m.year = 2013

RETURN m.title  
  
**Testing Inequality**MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name <> 'Tom Hanks'

AND m.title = 'Captain Phillips'

RETURN p.name  
  
**Testing less than or greater than**MATCH (m:Movie) WHERE m.title = 'Toy Story'

RETURN

m.year < 1995 AS lessThan, // Less than (false)

m.year <= 1995 AS lessThanOrEqual, // Less than or equal(true)

m.year > 1995 AS moreThan, // More than (false)

m.year >= 1995 AS moreThanOrEqual // More than or equal (true)  
  
**Testing ranges**MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name = 'Tom Hanks'

AND 2005 <= m.year <= 2010

RETURN m.title, m.released

**Or condition**MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name = 'Tom Hanks'

OR m.title = 'Captain Phillips'

RETURN p.name, m.title  
  
**Testing null conditions**MATCH (p:Person)

WHERE p.died IS NOT NULL

AND p.born.year >= 1985

RETURN p.name, p.born, p.died  
  
**Test to see if node has a label**MATCH (p:Person)

WHERE p.born.year > 1960

AND p:Actor

AND p:Director

RETURN p.name, p.born, labels(p)  
  
// Variation of above query  
MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)<-[:DIRECTED]-(p)

WHERE p.born.year > 1960

RETURN p.name, p.born, labels(p), m.title  
  
**Discovering relationship types**MATCH (p:Person)-[r]->(m:Movie)

WHERE p.name = 'Tom Hanks'

RETURN m.title AS movie, type(r) AS relationshipType  
  
**Test list inclusion**MATCH (m:Movie)

WHERE "Israel" IN m.countries

RETURN m.title, m.languages, m.countries  
 **Show query history**:history

**String comparison – starts with**MATCH (m:Movie)

WHERE m.title STARTS WITH 'Toy Story'

RETURN m.title, m.released  
  
**String comparison – ends with**MATCH (m:Movie)

WHERE m.title ENDS WITH ' I'

RETURN m.title, m.released  
  
**String comparison – contains**MATCH (m:Movie)

WHERE m.title CONTAINS 'River'

RETURN m.title, m.released  
  
**Case sensitive strings**MATCH (p:Person)

WHERE toLower(p.name) ENDS WITH 'demille'

RETURN p.name

MATCH (p:Person)

WHERE toUpper(p.name) ENDS WITH 'DEMILLE'

RETURN p.name  
  
**Patterns in the graph**

// Test to see if a pattern exists in a graph

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name = 'Tom Hanks'

AND exists {(p)-[:DIRECTED]->(m)}

RETURN p.name, labels(p), m.title  
  
**Profiling queries**// Shows total number of rows retrieved from the graph in each step in the query. PROFILE gives exact amount whereas EXPLAIN gives estimates.  
  
PROFILE MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)<-[:DIRECTED]-(p)

WHERE p.name = 'Tom Hanks'

RETURN m.title

**Finding non-patterns**MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name = 'Tom Hanks'

AND NOT exists {(p)-[:DIRECTED]->(m)}

RETURN m.title  
  
**Using multiple match clauses**// Retrieve actor, movie and director. However using a single MATCH pattern will yield best performance.  
MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)

WHERE m.year > 2000

MATCH (m)<-[:DIRECTED]-(d:Person)

RETURN a.name, m.title, d.name

// Better performing query  
MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)<-[:DIRECTED]-(d:Person)

WHERE m.year > 2000

RETURN a.name, m.title, d.name

**Optionally matching rows**

// Query that allows you to return rows that contain null values for some properties  
MATCH (m:Movie) WHERE m.title = "Kiss Me Deadly"

MATCH (m)-[:IN\_GENRE]->(g:Genre)<-[:IN\_GENRE]-(rec:Movie)

OPTIONAL MATCH (m)<-[:ACTED\_IN]-(a:Actor)-[:ACTED\_IN]->(rec)

RETURN rec.title, a.name

**Ordering results**MATCH (p:Person)

WHERE p.born.year = 1980

RETURN p.name AS name,

p.born AS birthDate

ORDER BY p.born  
  
MATCH (p:Person)

WHERE p.born.year = 1980

RETURN p.name AS name,

p.born AS birthDate

ORDER BY p.born DESC

**Eliminating null values from results**MATCH (p:Person)

WHERE p.born IS NOT NULL

RETURN p.name AS name, p.born AS birthDate

ORDER BY p.born DESC

**Multiple sort order of results**MATCH (p:Person)-[:DIRECTED | ACTED\_IN]->(m:Movie)

WHERE p.name = 'Tom Hanks'

OR p.name = 'Keanu Reeves'

RETURN m.year, m.title

ORDER BY m.year DESC , m.title  
  
  
**Eliminating duplicate results**MATCH (p:Person)-[:DIRECTED | ACTED\_IN]->(m:Movie)

WHERE p.name = 'Tom Hanks'

RETURN DISTINCT m.title, m.released

ORDER BY m.title  
  
**Return property values in JSON format only**MATCH (p:Person)

WHERE p.name CONTAINS "Thomas"

RETURN p { .\* } AS person

ORDER BY p.name ASC  
  
MATCH (p:Person)

WHERE p.name CONTAINS "Thomas"

RETURN p { .name, .born } AS person

ORDER BY p.name  
  
// Add new property called favourite to the JSON output  
MATCH (m:Movie)<-[:DIRECTED]-(d:Director)

WHERE d.name = 'Woody Allen'

RETURN m {.\*, favorite: true} AS movie

**Using CASE WHEN statements**

MATCH (m:Movie)<-[:ACTED\_IN]-(p:Person)

WHERE p.name = 'Henry Fonda'

RETURN m.title AS movie,

CASE

WHEN m.year < 1940 THEN 'oldies'

WHEN 1940 <= m.year < 1950 THEN 'forties'

WHEN 1950 <= m.year < 1960 THEN 'fifties'

WHEN 1960 <= m.year < 1970 THEN 'sixties'

WHEN 1970 <= m.year < 1980 THEN 'seventies'

WHEN 1980 <= m.year < 1990 THEN 'eighties'

WHEN 1990 <= m.year < 2000 THEN 'nineties'

ELSE 'two-thousands'

END

AS timeframe  
  
**Using COUNT to aggregate data**MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)

WHERE a.name = 'Tom Hanks'

RETURN a.name AS actorName,

count(\*) AS numMovies  
  
// How many times an actor and director worked together. You do not need a grouping key as all non-aggregated result columns act as grouping keys.  
MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)<-[:DIRECTED]-(d:Person)

RETURN a.name AS actorName,

d.name AS directorName,

count(\*) AS numMovies  
ORDER BY numMovies DESC  
  
**Using LIMIT**MATCH (m:Movie)

RETURN m.languages AS languages,

m.countries AS countries

LIMIT 10  
  
**Returning a list – inside square brackets**MATCH (p:Person)

RETURN p.name, [p.born, p.died] AS lifeTime

LIMIT 10  
  
**Using collect() to aggregate values in a list**MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)

RETURN a.name AS actor,

count(\*) AS total,

collect(m.title) AS movies

ORDER BY total DESC LIMIT 10  
  
**Eliminating duplication in lists**

MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)

WHERE m.year = 1920

RETURN collect( DISTINCT m.title) AS movies,

collect( a.name) AS actors  
  
**Collecting nodes**// Returns list of all Movie nodes for Tom CruiseMATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name ='Tom Cruise'

RETURN collect(m) AS tomCruiseMovies  
  
**Accessing elements of a list**// Access elements in a list. List begins with index 0. This returns the first name in the list.

MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)

RETURN m.title AS movie,

collect(a.name)[0] AS castMember,

// This returns the second and subsequent names in the list  
MATCH (a:Person)-[:ACTED\_IN]->(m:Movie)

RETURN m.title AS movie,

collect(a.name)[2..] AS castMember,

size(collect(a.name)) as castSize

size(collect(a.name)) as castSize  
  
**Size returns the number of elements in a list**MATCH (actor:Person)-[:ACTED\_IN]->(m:Movie)<-[:DIRECTED]-(director:Person)

RETURN actor.name, director.name,

size(collect(m)) AS collaborations,

collect(m.title) AS movies  
  
**List comprehension**// Return movies where countries either USA or GermanyMATCH (m:Movie)

RETURN m.title as movie,

[x IN m.countries WHERE x = 'USA' OR x = 'Germany']

AS country LIMIT 500  
  
**Pattern comprehension**// Create lists without changing the cardinality of the query  
MATCH (m:Movie)

WHERE m.year = 2015

RETURN m.title,

[(dir:Person)-[:DIRECTED]->(m) | dir.name] AS directors,

[(actor:Person)-[:ACTED\_IN]->(m) | actor.name] AS actors  
  
// Create a format for the list pattern  
MATCH (a:Person {name: 'Tom Hanks'})

RETURN [(a)-->(b:Movie)

WHERE b.title CONTAINS "Toy" | b.title + ": " + b.year]

AS movies  
  
**Working with maps**// Create key-value pairs that can be used as a property in a graph  
  
RETURN {Jan: 31, Feb: 28, Mar: 31, Apr: 30 ,

May: 31, Jun: 30 , Jul: 31, Aug: 31, Sep: 30,

Oct: 31, Nov: 30, Dec: 31}.Feb AS daysInFeb  
  
// Return list of keys of a map  
RETURN keys({Jan: 31, Feb: 28, Mar: 31, Apr: 30 ,

May: 31, Jun: 30 ,Jul: 31, Aug: 31, Sep: 30,

Oct: 31, Nov: 30, Dec: 31}) AS months  
  
**Map projections**// Map projections are when you can use retrieved nodes to create or return some of the information in the nodes.  
MATCH (m:Movie)

WHERE m.title CONTAINS 'Matrix'

RETURN m { .title, .released } AS movie  
  
**Code returns duration between Date1 and Date2**  
MATCH (x:Test {id: 1})

RETURN duration.between(x.date1,x.date2)  
  
// Calculate number of days between date1 and date 2  
MATCH (x:Test {id: 1})

RETURN duration.between(x.datetime1,x.datetime2).days  
  
// Add duration of 6 months  
MATCH (x:Test {id: 1})

RETURN x.date1 + duration({months: 6})  
  
**Formatting date and time**MATCH (x:Test {id: 1})

RETURN x.datetime as Datetime,

apoc.temporal.format( x.datetime, 'HH:mm:ss.SSSS')

AS formattedDateTime

// ISO8601 datetime format

MATCH (x:Test {id: 1})

RETURN apoc.date.toISO8601(x.datetime.epochMillis, "ms")

AS iso8601

**Anchor of a query**// Movie nodes are the anchor for this query because there are 9k of them compared to 19k people. Anchor to entity with fewest nodes.  
PROFILE MATCH (p:Person)-[:ACTED\_IN]->(m: Movie)

RETURN p.name, m.title LIMIT 100  
  
// This will anchor to the filtered person node. If the Person nodes has an index on name, it only retrieves one record. If there is no index, it needs to scan/filter all Person nodes for the name property.  
PROFILE MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name = 'Eminem'

RETURN p.name, m.title

**Multiple anchors**// This query has two sets of anchor nodes – p1 and p2. It retrieves anchor nodes before the equality filter is applied. Note: having a label on non-anchor nodes such as m1 and m2 are not necessary and will add more db hits if they are labelled.  
  
PROFILE

MATCH (p1:Person)-[:ACTED\_IN]->(m1)

MATCH (m2)<-[:ACTED\_IN]-(p2:Person)

WHERE p1.name = 'Tom Hanks'

AND p2.name = 'Meg Ryan'

AND m1 = m2

RETURN m1.title  
  
**Expand to follow path**// Specify the nodes and relationship to eliminate paths from the nodes to memory to nodes that will need to be retrieved. This is a good example to sufficiently narrowing the query.  
  
PROFILE MATCH (m:Movie)<-[:DIRECTED]-(p:Person)

WHERE p.name = 'Clint Eastwood'

RETURN m.title

**Shortest path**// You can calculate the shortest path between any two nodesMATCH p = shortestPath((p1:Person)-[\*]-(p2:Person))

WHERE p1.name = "Eminem"

AND p2.name = "Charlton Heston"

RETURN p  
  
// Shortest path where relationship is specified.  
MATCH p = shortestPath((p1:Person)-[:ACTED\_IN\*]-(p2:Person))

WHERE p1.name = "Eminem"

AND p2.name = "Charlton Heston"

RETURN p  
  
**Varying length traversal**// Retrieve all person nodes that are exactly two hops away  
MATCH (p:Person {name: 'Eminem'})-[:ACTED\_IN\*2]-(others:Person)

RETURN others.name  
  
// Up to 4 hops away  
MATCH (p:Person {name: 'Eminem'})-[:ACTED\_IN\*1..4]-(others:Person)

RETURN others.name  
  
**Scoping variables for a query using WITH**// Initialise variable name actorName with a value  
WITH 'Tom Hanks' AS actorName

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name = actorName

RETURN m.title AS movies  
  
// Calculating a value for a variable using WITH  
WITH 'toy story' AS mt, 'Tom Hanks' AS actorName

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WITH m, toLower(m.title) AS movieTitle

WHERE p.name = actorName

AND movieTitle CONTAINS mt

RETURN m.title AS movies, movieTitle  
  
**Using map projections in a WITH clause**// Specify which properties are returned by assigning them to a variable  
WITH 'toy story' AS mt, 'Tom Hanks' AS actorName

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WITH m, toLower(m.title) AS movieTitle

WHERE p.name = actorName

AND movieTitle CONTAINS mt

RETURN m.title AS movies, movieTitle  
  
// Return highest revenue movie for Tom Hanks  
WITH 'Tom Hanks' AS theActor

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name = theActor

AND m.revenue IS NOT NULL

WITH m ORDER BY m.revenue DESC LIMIT 1

RETURN m.revenue AS revenue, m.title AS title  
  
// Return a collection of properties for the 4 highest rating movies  
MATCH (n:Movie)

WHERE n.imdbRating IS NOT NULL AND n.poster IS NOT NULL

WITH n {

.title,

.imdbRating,

actors: [ (n)<-[:ACTED\_IN]-(p) | p { tmdbId:p.imdbId, .name } ],

genres: [ (n)-[:IN\_GENRE]->(g) | g {.name}]

}

ORDER BY n.imdbRating DESC

LIMIT 4

RETURN collect(n)  
  
**Pipeline results from first query to second query**// The WITH clause enables us to pipeline the results of the first query into the second query. With this simple query, m need not be redefined or scoped.  
WITH 'Tom Hanks' AS theActor

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name = theActor

WITH m LIMIT 5

MATCH (d:Person)-[:DIRECTED]->(m)

RETURN d.name AS director,

m.title AS movies  
  
**Using WITH for aggregation**MATCH (:Movie {title: 'Toy Story'})-[:IN\_GENRE]->(g:Genre)<-[:IN\_GENRE]-(m)

WHERE m.imdbRating IS NOT NULL

WITH g.name AS genre,

count(m) AS moviesInCommon,

sum(m.imdbRating) AS total

RETURN genre, moviesInCommon,

total/moviesInCommon AS score

ORDER By score DESC  
  
**Aggregation and pipelining**MATCH (u:User {name: "Misty Williams"})-[r:RATED]->(:Movie)

WITH u, avg(r.rating) AS average

MATCH (u)-[r:RATED]->(m:Movie)

WHERE r.rating > average

RETURN average , m.title AS movie,

r.rating as rating

ORDER BY rating DESC  
  
**Using WITH for collecting**// Collects names and numbers of actorsMATCH (m:Movie)--(a:Actor)

WHERE m.title CONTAINS 'New York'

WITH m, collect (a.name) AS actors,

count(\*) AS numActors

RETURN m.title AS movieTitle, actors

ORDER BY numActors DESC  
  
// Another example  
MATCH (m:Movie)<-[:ACTED\_IN]-(a:Actor)

WHERE m.title CONTAINS 'New York'

WITH m, collect (a.name) AS actors,

count(\*) AS numActors

ORDER BY numActors DESC

RETURN collect(m { .title, actors, numActors }) AS movies

**Use LIMIT early in WITH statement to improve query performance**PROFILE MATCH (p:Actor)

WHERE p.born.year = 1980

WITH p LIMIT 3

MATCH (p)-[:ACTED\_IN]->(m:Movie)

WITH p, collect(m.title) AS movies

RETURN p.name AS actor, movies  
  
**Ensures duplicates are not in the collection**MATCH (p:Actor)

WHERE p.born.year = 1980

WITH p LIMIT 3

MATCH (p)-[:ACTED\_IN]->(m:Movie)-[:IN\_GENRE]->(g:Genre)

WITH p, collect(DISTINCT g.name) AS genres

RETURN p.name AS actor, genres  
  
**Unwinding lists**// Unwind languages so there is one movie and one language. For example, Catch Me If You Can, French and Catch Me If You Can, English  
MATCH (m:Movie)-[:ACTED\_IN]-(a:Actor)

WHERE a.name = 'Tom Hanks'

UNWIND m.languages AS lang

RETURN m.title AS movie,

m.languages AS languages,

lang AS language  
  
// Find 10 movies for each language  
MATCH (m:Movie)

UNWIND m.languages AS lang

WITH m, trim(lang) AS language

// this automatically, makes the language distinct because it's a grouping key

WITH language, collect(m.title) AS movies

RETURN language, movies[0..10]  
  
// Only return number of movies released in Taiwan  
MATCH (m:Movie)

UNWIND m.countries AS cntry

WITH m, trim(cntry) AS countries

WITH countries, m.title AS movies WHERE countries = "Taiwan"

RETURN countries, count(movies)  
  
**keys with CALL**// Return 10 movies released in 2000 with highest ratings, then determine average rating for each of the 10 movies  
CALL {

MATCH (m:Movie) WHERE m.year = 2000

RETURN m ORDER BY m.imdbRating DESC LIMIT 10

}

MATCH (:User)-[r:RATED]->(m)

RETURN m.title, avg(r.rating)  
  
**Passing a variable into a CALL subquery**MATCH (m:Movie)

CALL {

WITH m

MATCH (m)<-[r:RATED]-(u:User)

WHERE r.rating = 5

RETURN count(u) AS numReviews

}

RETURN m.title, numReviews

ORDER BY numReviews DESC  
  
**Combining query results with UNION or UNION ALL**// UNION ALL returns all results which is more efficient on memory but can lead to duplicates. UNION returns distinct results.  
MATCH (m:Movie) WHERE m.year = 2000

RETURN {type:"movies", theMovies: collect(m.title)} AS data

UNION ALL

MATCH (a:Actor) WHERE a.born.year > 2000

RETURN { type:"actors", theActors: collect(DISTINCT a.name)} AS data  
  
**Using UNION within a sub-query**MATCH (p:Person)

WITH p LIMIT 100

CALL {

WITH p

OPTIONAL MATCH (p)-[:ACTED\_IN]->(m:Movie)

RETURN m.title + ": " + "Actor" AS work

UNION

WITH p

OPTIONAL MATCH (p)-[:DIRECTED]->(m:Movie)

RETURN m.title+ ": " + "Director" AS work

}

RETURN p.name, collect(work):IN  
  
// Return most popular genre movie released in France  
MATCH (g:Genre)

CALL { WITH g

MATCH (g)<-[:IN\_GENRE]-(m) WHERE 'France' IN m.countries

RETURN count(m) AS numMovies

}

RETURN g.name AS Genre, numMovies ORDER BY numMovies DESC  
  
**UNION ALL Query Example**// All actors and directors in 2015  
MATCH (m:Movie)<-[:ACTED\_IN]-(p:Person)

WHERE m.year = 2015

RETURN "Actor" AS type,

p.name AS workedAs,

collect(m.title) AS movies

UNION ALL

MATCH (m:Movie)<-[:DIRECTED]-(p:Person)

WHERE m.year = 2015

RETURN "Director" AS type,

p.name AS workedAs,

collect(m.title) AS movies  
  
**Parameters**// A parameter name begins with the $ symbol  
:param actorName: 'Tom Hanks'  
MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name = $actorName

RETURN m.released AS releaseDate,

m.title AS title

ORDER BY m.released DESC

**Setting integers**:param number=> 10  
  
**Setting multiple parameters**MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name = $actorName

AND m.title = $movieName

RETURN p, m  
  
**Viewing parameters**:params **Removing parameters**:params {}  
  
**How to use parameters with Python**// In Python, Cypher parameters are passed as named parameters to the tx.run method. In this example, title has been passed as a named parameter.  
def get\_actors(tx, movieTitle): # (1)

result = tx.run("""

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE m.title = $title

RETURN p

""", title=movieTitle)

# Access the `p` value from each record

return [ record["p"] for record in result ]

with driver.session() as session:

result = session.read\_transaction(get\_actors, movieTitle="Toy Story")

**Returning paths**// In some applications, you may need to work with path objects – such as length(p) = length of path, nodes(p) = list of nodes for a path, relationships(p) = returns a list containing the relationships for a path.MATCH p = ((person:Person)-[]->(movie))

WHERE person.name = 'Walt Disney'

RETURN p