**Commands For Neo4J:**

To Create a Node:

1. create(e:employee) 🡺 This will create a new node in database
2. create(e:employee) return e 🡺 This will return the node created now.
3. create(e:employee:proctor) return e 🡺Node which holds the different labels, like

an employee is also a proctor.

Creating Multiple Nodes in single command:

1. create(e:employee),(p:proctor) return e,p 🡺 Create and return two different node of different “Labels”.
2. create(e:employee{name:"Haseeb Ahmad"}),(p:proctor) return e,p 🡺 This will create node with specific properties.
3. CREATE(e:employee{name:"ram", branch:"cse"}),(f:employee{name:"sam", branch:"ece"}),(g:employee{name:"tom", branch:"eee"}) return e,f,g 🡺 Creating multiple nodes of same Label/type having different properties.
4. CREATE(e:employee{name:"ram", branch:"cse"}),(f:employee{name:"sam", branch:"ece"}),(g:employee{name:"tom", branch:"eee"}),(h:professor{name:"satish",designation:"ap1"}),(j:professor{name:"ramesh",designation:{"ap2"}}) return e,f,g, h, j

🡺 Creating multiple nodes of same and different Label/type having different properties.

To Find Nodes:

1. MATCH (n)  🡺 match all nodes inside database but returns none.
2. MATCH (n) return n 🡺 match all nodes inside database and returns all
3. MATCH(e:employee),(d:department) return e.name, d.name 🡺 This will return “names” of both the labels ‘department’ and ‘employee’.
4. MATCH(e:employee) where  e.name="Haseeb Ahmad" return e 🡺 Use of where clause.
5. MATCH(e:employee{name:"Haseeb Ahmad"}) return e 🡺 Without where clause matching for specific property.

To Update Node Information:

1. match(e:employee{name:’satish’}) set e.salary = 20000 🡺 This will set the salary property to 20000, this property was not included before, it is added now.
2. match(e:employee{name:’satish’}) remove e.salary 🡺 This will remove the salary property from node.
3. match(e:employee{name:’satish’}) remove e.name set e.firstName=’satish’ return e 🡺 This will return node of satish with ‘name’ property being removed and firstName is added.

Using ‘Set’ and ‘Remove’ clauses to change the ‘label’/’Type’ of a node.

1. match(e:employee{name:”satish”}) remove e:employee set e:clerk return e 🡺 This will remove ‘e’ from employee and set it to ‘clerk’ label.

Using ‘Delete’ clause to delete the node.

1. match(e:employee) delete e.
2. match(n) delete n 🡺 Delete all nodes in database.

Using Arithmetic Operators:

1. match(e:employee) where e.name in ['satihs', 'Haseeb'] return e 🡺 This “IN” keyword search for value to fall in specific set of values like in above array.
2. match(e:employee) where e.name='Haseeb' and e.salary=20000 return e 🡺 Use of ‘AND’ operator.
3. match(e:employee) where e.name='Haseeb' or e.name='satish' return e 🡺 Use of ‘OR’ operator.

Creating Relationship between Nodes:

1. match(e:employee{name:’satish’}), (d:department{name:’scope’}) create (e) –[w:worksFor] -> (d) return e,w,d 🡺 In this we are first matching/finding the required nodes, then we are creating a link between two nodes by specifying the “Label” of relationship and then returning all ‘nodes’ and their ‘relation’.
2. match(e:employee)-[w:worksfor]->(d:department{name:’scope’}) return e,w,d 🡺 This will find the all employees which have ‘workfor’ relationship with department named as ‘scope’.
3. match(e:employee{name:’satish’})-[w:worksfor]->(d:department) return e,w,d 🡺 It will return all departments in which ‘satish’ works.

Updating Relations between Nodes.

1. match(e:employee{name:"Haseeb"})-[s:studiesIn]->(d:department{name:"SEECS"}) set s.studyingDepartment="DOC" return e,s,d 🡺 This will add more values/information in label attaching the two nodes.
2. match(e:employee{name:"Haseeb"})-[s:studiesIn]->(d:department{name:"SEECS"}) remove s.studyingDepartment return e,s,d 🡺 This will remove the new value added on label as “studyingDepartment”.
3. match(e:employee{name:"Haseeb"})-[s:studiesIn]->(d:department{name:"SEECS"}) delete s return e,d 🡺 This will delete the relationship between the nodes and set them free of each other.

Creating Relationships While Creating Nodes and More on Node Relationships:

1. When we have relationship between two nodes, we can’t delete those nodes. The command for this will be, 🡺 match(n) detach delete n. For deleting a specific node we can first “select it using match() clause and then detach it and at the end delete it”.
2. create(d:driver{name:'Haseeb'})-[r:drives]->(c:car{name:"Kia"}) return d,r,c 🡺 This will create both nodes and their relationship at the same time.
3. match(e)-[r:drives]->(d) return e,r,d 🡺 This will select all nodes irrespective of their labels, which have incoming/outgoing “drives” relationship attached with them.
4. create(e:employee{name:"Waleed"})-[s:studiesIn]->(d:department{name:"SEECS"}) <-[s1:studiesIn]-(e1:employee{name:"Kalasra"}) return e, e1, s, s1, d 🡺 This query will add two nodes in the same query time to a single department.
5. CREATE(e:employee{name:"satish"})-[w:worksFor]->(d:department{name:"SMME"})<-[w1:workFor]-(e1:employee{name:"sam"})-[m:manages]->(d1:department{name:"sbst"})<-[m1:manages]-(e2:employee{name:"tom"}) return e, w, d, w1, e1, m,d1, m1,e2  🡺 This will create three in which one is working in two departments and other two are working in same department along with the departments in which they are working. This operation is completed using a single command.

Finding all In-coming and Out-Going Relationships:

1. match(d:department{name:"SMME"})<--(e) return d,e 🡺 This will return all incoming relationships from to “SMME” department, from any other node.
2. match(d:employee)-->(e) return d,e 🡺 This return all out-going nodes of all employees, like to which other node they are attached with.
3. match(e)-[m:manages|worksFor]->(d) return m,d,e 🡺 This query will return nodes which have “manages”, ”workFor” relationships between. Each node can have both or any one relationship.

Order By Clause:

1. match(e:employee) return e.name order by e.name 🡺 This will return employee names in ascending order.
2. match(e:employee) return e.name, e.salary order by e.name, e.salary 🡺 This will return names of employees ordered by multiple properties.

Skip Clause:

1. match(e:employee) return e.name skip {Any Integer Value} 🡺 This command will skip the first “Integer” value from result list and return the remaining list.

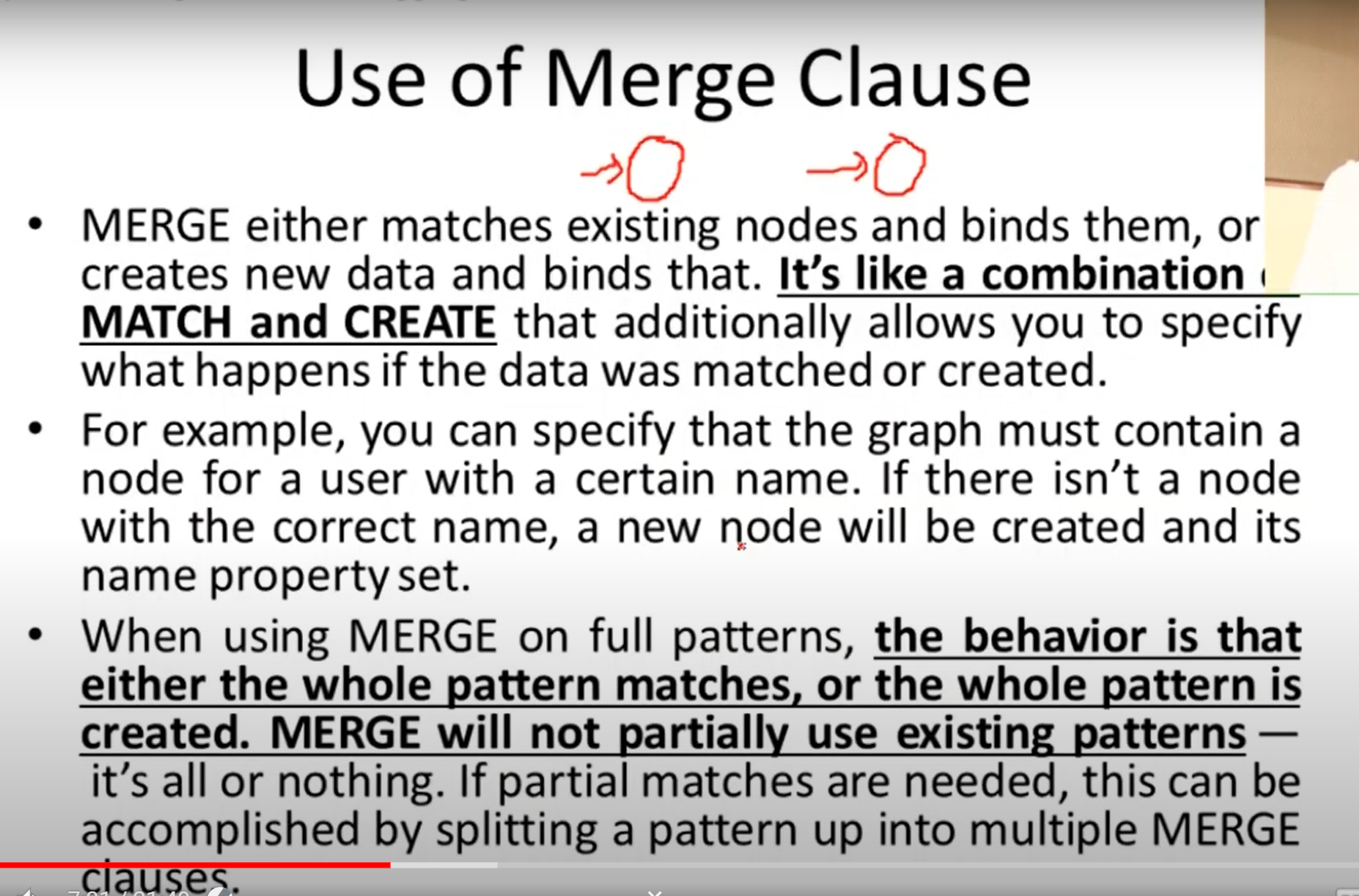
Limit Clause:

1. match(e:employee) return e.name limit 3 🡺 This returns only top 3 results from the result list.

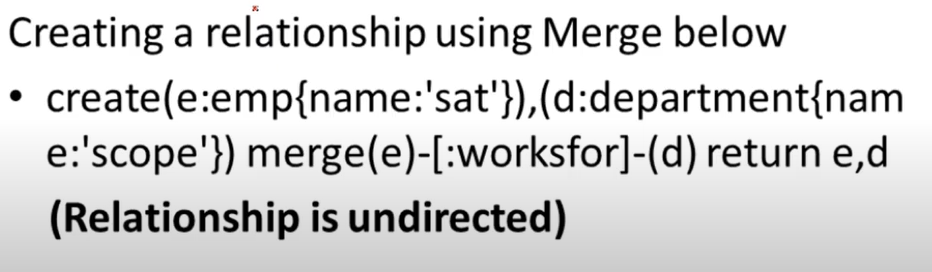
Union and Union All Clause:

1. match(e:employee) return e.name UNION match (e:employee) return e.name 🡺 This command will return unique result by joining two sets described in our query labels.
2. match(e:employee) return e.name UNION ALL match (e:employee) return e.name 🡺 This command will return all values irrespective of rather they occur single time or twice or more, that is the main difference between “UNION” and “UNION ALL” clause.

Merge Clause:



1. merge(e:driver) return e 🡺 This query creates a new node if it does not exist otherwise return it as it is, with its value.
2. merge(e:clerk) on create set e.Joined="Today" return e 🡺 This command will create a new node if it does not exist and if it is created by this command then it will set its property “Joined” on creating this node.
3. merge(e:clerk) on match set e.Joined="Yesterday" return e 🡺 This command will create a new node if it does not exist and if it existed then this command will change the “Joined” property to a new value.
4. merge(e:clerk) on match set e.Joined="Yesterday" on create set e.Joined="Today" return e 🡺 Single command to run above both cases.
5. Text

   Description automatically generatedMerge Clause on Relationship between Nodes.

Aggregate Functions

1. match(e:employee) return count(\*) as totalEmployees 🡺 This command will calculate total employees in employees Label.
2. match(e:employee) return min(e.salary), max(e.salary) 🡺 Min, Max aggregate functions.
3. match(e:employee) return sum(e.salary) 🡺 Sum Aggregate Function.
4. match(e:employee{gender:'Male'}) return count(\*) 🡺 This will total count of ‘Male’ employees inside department.
5. match(e:employee)-[w:worksFor]->(d:department) return d.name, count(\*) 🡺 This command will return total number of employees in each department.
6. match(e:employee)-[w:worksFor]->(d:department) return d.name, count(\*) order by count(\*) desc 🡺 This command returns total employees in each department in descending order.
7. Avg 🡺 This is another aggregate function.

Constraints in Neo4J

Constraints are Rules we apply in our database, like length of string should be 12 characters.

Unique Constraint

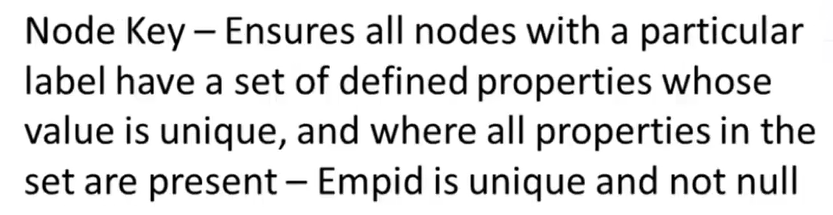
1. CREATE constraint on (e:employee) assert e.phone is unique 🡺 This command will create a constraint of unique phone number on all employees.
2. call db.constraints 🡺 To see all constraints on database.

Property Existence Constraint.

Ensures that a Property Exists for all nodes with a specific label or for all relationships with specific type.

1. create constraint on (e:clerk) assert exists(e.Joined) 🡺 Make sure the above property.
2. create constraint on () –[m:manages] -> () assert exists(m.joined) 🡺 This command make sure that when ever a “manages” relationship is created, it should have “joined” value in it also.

Node Key Constraint



1. create constraint on (s:something) assert (s.name) is node key 🡺 Above command will create “node key” constraint on s.name field on all something label values.
2. For creating composite, we can add multiple fields of specific label, that will be 🡺

create constraint on (s:something) assert (s.name,s.phone, …...) is node key

Removing Unique Constraint

1. drop constraint on (c:employee) assert (c.phone) is unique 🡺 This command will remove constraint on employee phone number.
2. drop constraint on (c:clerk) assert exists(c.Joined) 🡺 This command will remove exists constraint on any value.

String Matching Expressions

1. match p=(e:employee)-[:workFor]->(d:department) where e.name starts with 's' return p 🡺 In this command first we are fetching the data based on relationship between nodes and storing it, then we are filtering the data based on some specific pattern we want to search for.
2. match p=(e:employee)-[:studiesIn]->(d:department) where e.name ends with 'd' return p 🡺 This query search for employees whose name ends with word ‘d’.
3. match p=(e:employee)-[:studiesIn]->(d:department) where e.name contains 'lee' return p 🡺 This command search for specific pattern inside employees.

Case Expression

Case Expression is just like switch case in C/C++/Java or any other programming language. Its pattern is:

CASE test

WHEN value THEN result

[WHEN….]

[ELSE default]

END

1. match(e:employee) return case e.name when 'Waleed' then 'Hi Waleed' when 'tom' then 'hi tom' else 'bye' end 🡺 This command is practical example of above code pattern.

Nodes Concept

Nodes keyword returns the data in the form of list. Each node values will be returned as list object.

1. match p=(e:employee) return nodes(p)

Foreach Clause

Same as Foreach loop in other programming languages.

1. match p=(e:employee) foreach(emp in nodes(p) | set e.Joined = 'today' ) 🡺 This command first selects all employees, then loop over each employee and then add a new property as ‘Joined’, to each employee node

Finding Nods with Multi-level relationships

1. match(e:employee)-[\*1]-(d:department) return e,d 🡺 This command will return “Employees” along with “department” nodes which are directly connected with “Employees” having any kind of relationship, such nodes are called as “First Level Nodes/Friends”.
2. match(e:employee)-[:studiesIn\*1]-(d:department) return e,d 🡺 This command will again return first level nodes of employee, but they make sure that the relationship type is “studiesIn” nothing else.
3. match(e:employee)-[:studiesIn\*1]->(d:department) return e,d 🡺 This command has only difference that is “->”, this arrow make sure that the relationship is going out of employees not, in-coming relationships should be counted.
4. match(e:employee)-[\*2]-(d:department) return e,d 🡺 This command will show second level nodes which are connect with first level nodes of “employee”.

Getting Multi-Level Connections using same query

1. match(e:employee)-[\*1..3]-(d:department) return e,d 🡺 This is combined query which gives all nodes in level 1,2,3 starting from employee node.

Indexes in Neo4j

1. create index on :employee(name) 🡺 This command will create “index” on “name” property of “Employee”.
2. call db.indexes 🡺 This command will show all indexes of database.
3. create index on :employee(name,salary) 🡺 Composite Index.
4. drop index on :employee(name) 🡺 This command will drop the index on specific property of label.
5. <https://neo4j.com/docs/cypher-manual/current/functions/string/> 🡺 This link describes all string handling functions provided by Neo4J.

Graph Analytics using Neo4J

Call and Yield

Call 🡺 It is used for calling the procedure

Yield 🡺 It is used for specifying the output of procedure

1. call dbms.procedures 🡺 This will return all procedures/functions of dbms with all other properties.
2. call dbms.procedures YIELD name, signature 🡺 This command will return only “name” and “signature” of procedures.
3. match(e) where id(e)=2 return e 🡺 To find element with specific ID.

Shortest Path Query

1. match(e:employee{name:"Waleed"}), (e1:employee{name:'Haseeb'}), p=SHORTESTPATH((e)-[\*1..5]-(e1)) return p 🡺 This command finds the shortest path relationship between “Waleed” and “Haseeb” within the range of “5” hopes and having any kind of relationship. This is shortest Path distance based on number of hopes.
2. match(e:employee{name:"Waleed"}), (e1:employee{name:'Haseeb'}), p=ALLSHORTESTPATHS((e)-[\*1..5]-(e1)) return p 🡺 All shortest paths between two nodes.

**Clustering In Neo4j**

To create cluster of servers of neo4j database, follow this video, for configurations understanding. <https://www.youtube.com/watch?v=1YtvxGVXkeE&list=PLqfPEK2RTgChcOZ6qHgSfwiBPCz2Bzdjh&index=8>

For Understanding through documentation, about how to change configuration file, follow this link:

<https://www.youtube.com/watch?v=1YtvxGVXkeE&list=PLqfPEK2RTgChcOZ6qHgSfwiBPCz2Bzdjh&index=8>

This link will also guid about how we can set up replica servers.

We can only write data through leader node and that data will be replicated to all the followers. In followers we can’t write the data, we can only read it from here.

1. :sysinfo 🡺 This command will show all the information regarding servers inside cluster.
2. neo4j console 🡺 This command will start a single instance of neo4j server. Move to “bin” folder using command line and then run above command to start the server.