



# BASI DI DATI 2

*DATABASE NOSQL: NEO4J*

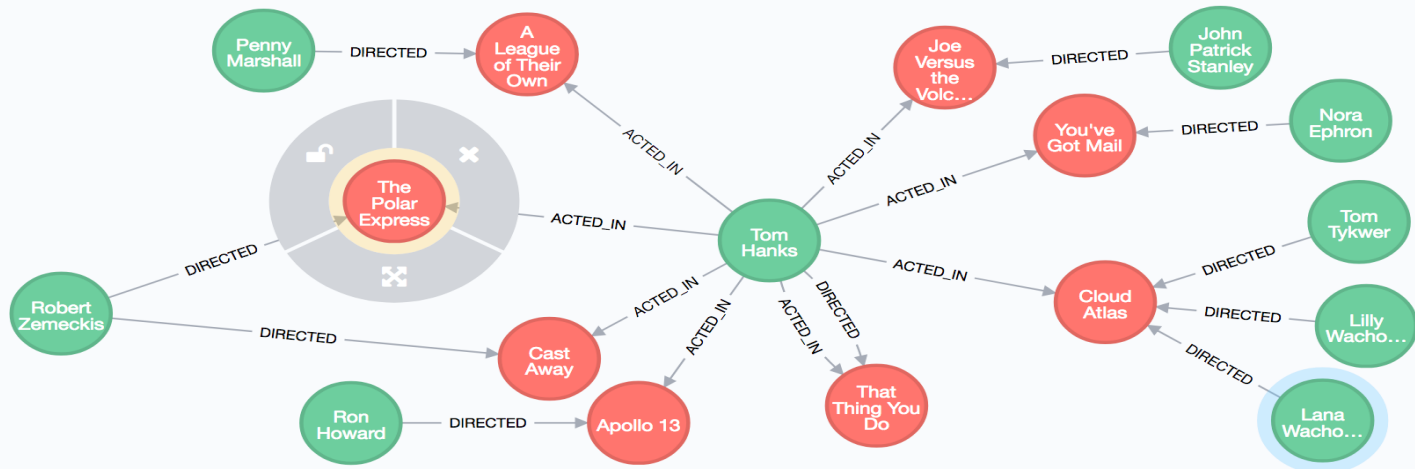
# Neo4j

- Neo4j is one of the popular Graph Databases and Cypher Query Language (CQL).

- Neo4j is written in Java Language.

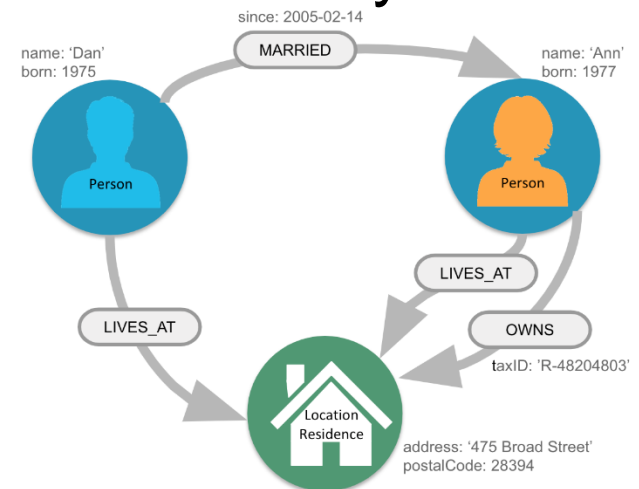


- It is highly scalable and schema free (NoSQL).



# What is a Graph Database?

- A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links.
  - ▣ It is composed of two elements:
    - nodes (vertices)
    - relationships (edges)
- Graph database is a database used to model the data in the form of graph.
  - ▣ The nodes of a graph depict the entities while the relationships depict the associations of these nodes.



# Why Graph Databases?

- Most of the data exists in the form of the relationship between different objects:
  - ▣ The relationship between the data is more valuable than the data itself.
- Relational databases store highly structured data which have several records storing the same type of data:
  - ▣ they can be used to store structured data
  - ▣ they do not store the relationships between the data.
- The data model for graph databases is simpler compared to other databases and, they can be used with OLTP systems.
  - ▣ They provide features like transactional integrity and operational availability.

# RDBMS Vs Graph Database

<b>RDBMS</b>	<b>Graph Database</b>
Tables	Graphs
Rows	Nodes
Columns and Data	Properties and its values
Constraints	Relationships
Joins	Traversal

# Advantages of Neo4j

- ❑ **Flexible data model** – Neo4j provides a flexible simple and yet powerful data model, which can be easily changed according to the applications and industries.
- ❑ **Real-time insights** – Neo4j provides results based on real-time data.
- ❑ **High availability** – Neo4j is highly available for large enterprise real-time applications with transactional guarantees.
- ❑ **Connected and semi structures data** – you can easily represent connected and semi-structured data.
- ❑ **Easy retrieval** – you can not only represent but also easily retrieve (traverse/navigate) connected data faster when compared to other databases.
- ❑ **Cypher query language** – Neo4j provides a declarative query language to represent the graph visually, using an ascii-art syntax. The commands of this language are in human readable format and very easy to learn.
- ❑ **No joins** – it does not require complex joins to retrieve connected/related data as it is very easy to retrieve its adjacent node or relationship details without joins or indexes.

# Features of Neo4j

- **Data model (flexible schema)** – Neo4j follows a data model named native **property graph** model.
  - ▣ the graph contains nodes (entities) and these nodes are connected with each other (depicted by relationships).
  - ▣ Nodes and relationships store data in key-value pairs known as properties.
- There is no need to follow a fixed schema.
  - ▣ You can add or remove properties.
  - ▣ It provides schema constraints.
- **ACID properties** – Neo4j supports full ACID (Atomicity, Consistency, Isolation, and Durability) rules.
- **Scalability and reliability** – You can scale the database by increasing the number of reads/writes, and the volume without effecting the query processing speed and data integrity.
  - ▣ Neo4j also provides support for **replication** for data safety and reliability.

# Features of Neo4j (2)

- **Cypher Query Language** – Neo4j provides a powerful declarative query language known as Cypher.
  - ▣ It uses ASCII-art for depicting graphs.
  - ▣ Cypher is easy to learn and can be used to create and retrieve relations between data without using the complex queries like joins.
- **Built-in web application** – Neo4j provides a built-in **Neo4j Browser** web application.
  - ▣ Using this, you can create and query your graph data.
- **Drivers** – Neo4j can work with –
  - ▣ REST API to work with programming languages such as Java, Spring, Scala etc.
  - ▣ Javascript to work with UI MVC frameworks such as Node JS.
  - ▣ It supports two kinds of Java API: Cypher API and Native Java API to develop Java applications.
- **Indexing** – Neo4j supports indexes by using Apache Lucence.



# Data Model

- Neo4j Graph Database follows the Property Graph Model to store and manage its data.
  - ▣ The model represents data in Nodes, Relationships and Properties
  - ▣ Properties are key-value pairs
  - ▣ Nodes are represented using circle and Relationships are represented using arrow keys
  - ▣ Relationships have directions: Unidirectional and Bidirectional
  - ▣ Each Relationship contains "Start Node" or "From Node" and "To Node" or "End Node"
  - ▣ Both Nodes and Relationships contain properties
  - ▣ Relationships connects nodes

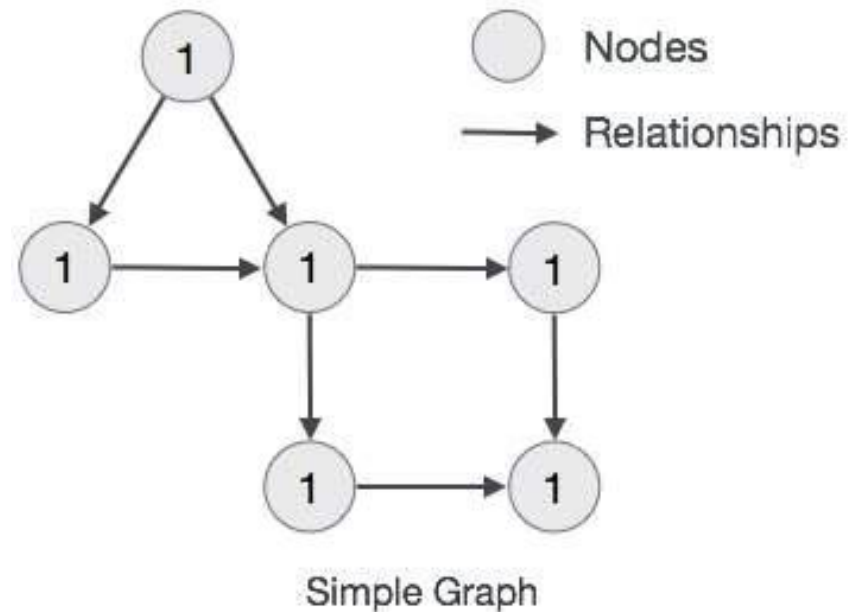
## Data Model (2)

- In Property Graph Data Model, relationships should be directional (--> or <--).
  - ▣ If we try to create relationships without direction, then it will throw an error message.
- Neo4j stores its data in terms of Graphs in its native format.
- Neo4j uses Native GPE (Graph Processing Engine) to work with its Native graph storage format.

# Data Model (3)

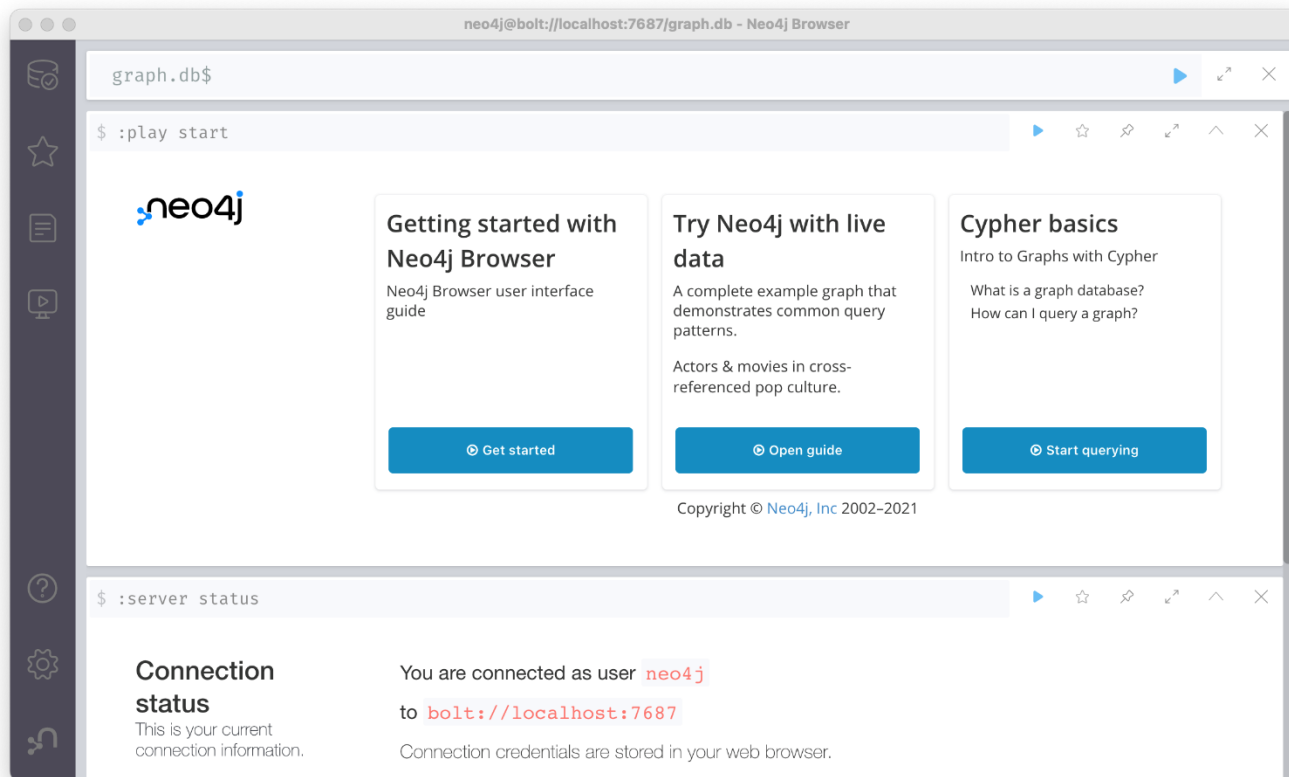
- The main building blocks of Graph DB Data Model are:

- ▣ Nodes
- ▣ Relationships
- ▣ Properties



# Neo4j Database Server Setup

- Neo4j official site is <https://neo4j.com>.
- Start the Neo4j server.
  - ▣ You can access Neo4j using the URL **<http://localhost:7474/>**



# Building Blocks

- Neo4j Graph Database has the following building blocks:
  - ▣ Nodes
  - ▣ Properties
  - ▣ Relationships
  - ▣ Labels
  - ▣ Data Browser (color, size, caption)

# Node

- Node is a fundamental unit of a Graph.
- It contains properties with key-value pairs.
  - ▣ Node Name = "Employee" contains a set of properties as key-value pairs.



Employee Node

# Properties

- Property is a key-value pair to describe Graph Nodes and Relationships.
  - ▣ Key = Value
- where Key is a String and Value may be represented using any Neo4j **data types**.
- Cypher provides support for a number of data types:
  - ▣ Property types
  - ▣ Structural types
  - ▣ Composite types

# Property types

- **Property types** comprise:
  - ▣ Number, an abstract type, which has the subtypes Integer (byte, short, int, long) and Float (float, double)
  - ▣ String (string, char)
  - ▣ Boolean
  - ▣ The spatial type *Point*
  - ▣ Temporal types: *Date*, *Time*, *LocalTime*, *DateTime*, *LocalDateTime* and *Duration*



# Structural types

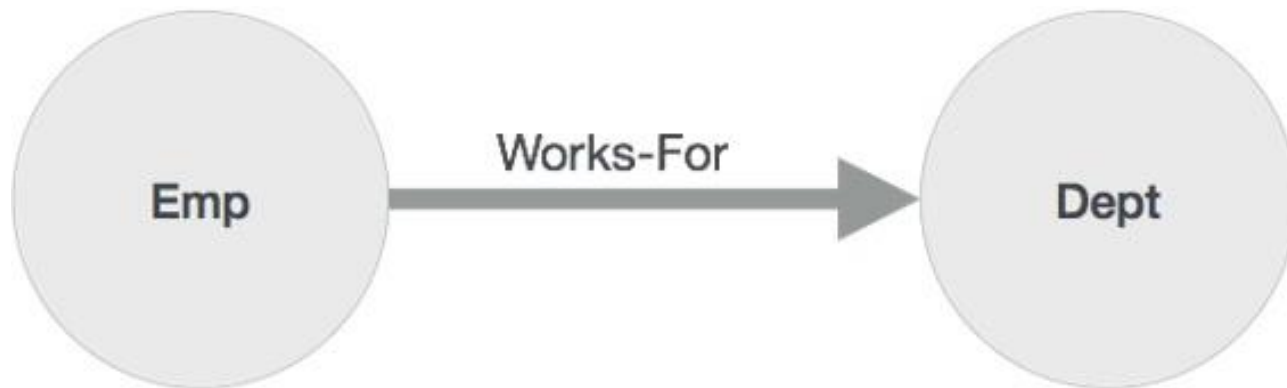
- **Structural types** comprise:
  - ▣ Nodes, comprising:
    - Id (*we can identify nodes as entities with a unique conceptual identity*)
    - Label(s)
    - Map (of properties)
  - ▣ Relationships, comprising:
    - Id
    - Type
    - Map (of properties)
    - Id of the start and end nodes
  - ▣ Paths
    - An alternating sequence of nodes and relationships

# Composite types

- **Composite types** comprise:
  - ▣ **Lists** are heterogeneous, ordered collections of values, each of which has any property, structural or composite type.
  - ▣ **Maps** are heterogeneous, unordered collections of (key, value) pairs, where:
    - the key is a String
    - the value has any property, structural or composite type
- Composite values can also contain **null**.

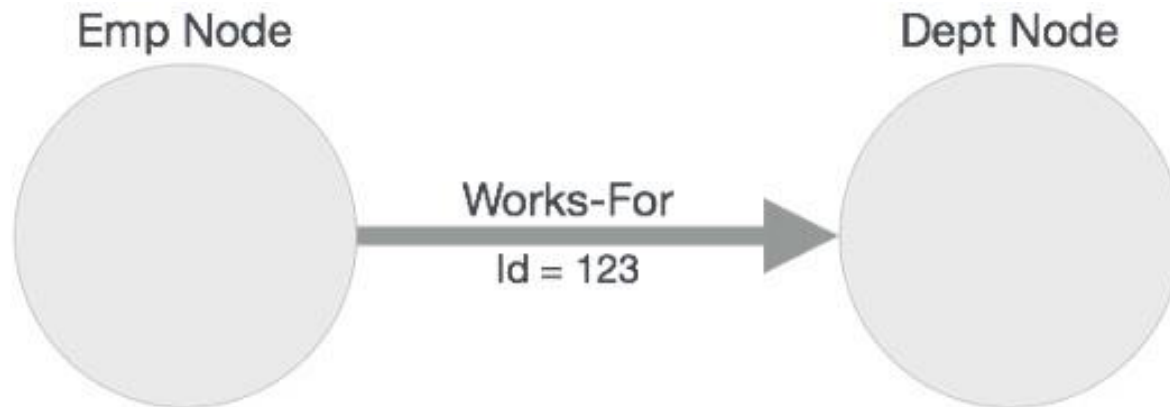
# Relationships

- Relationships are another major building block of a Graph Database.
  - It connects two nodes.
  - Emp (*start node*) and Dept (*end node*) are two different nodes.
  - "WORKS\_FOR" is a relationship between Emp and Dept nodes.



## Relationships (2)

- Like nodes, relationships also can contain properties as key-value pairs.
  - ▣ "WORKS\_FOR" relationship has one property as key-value pair.



# Labels

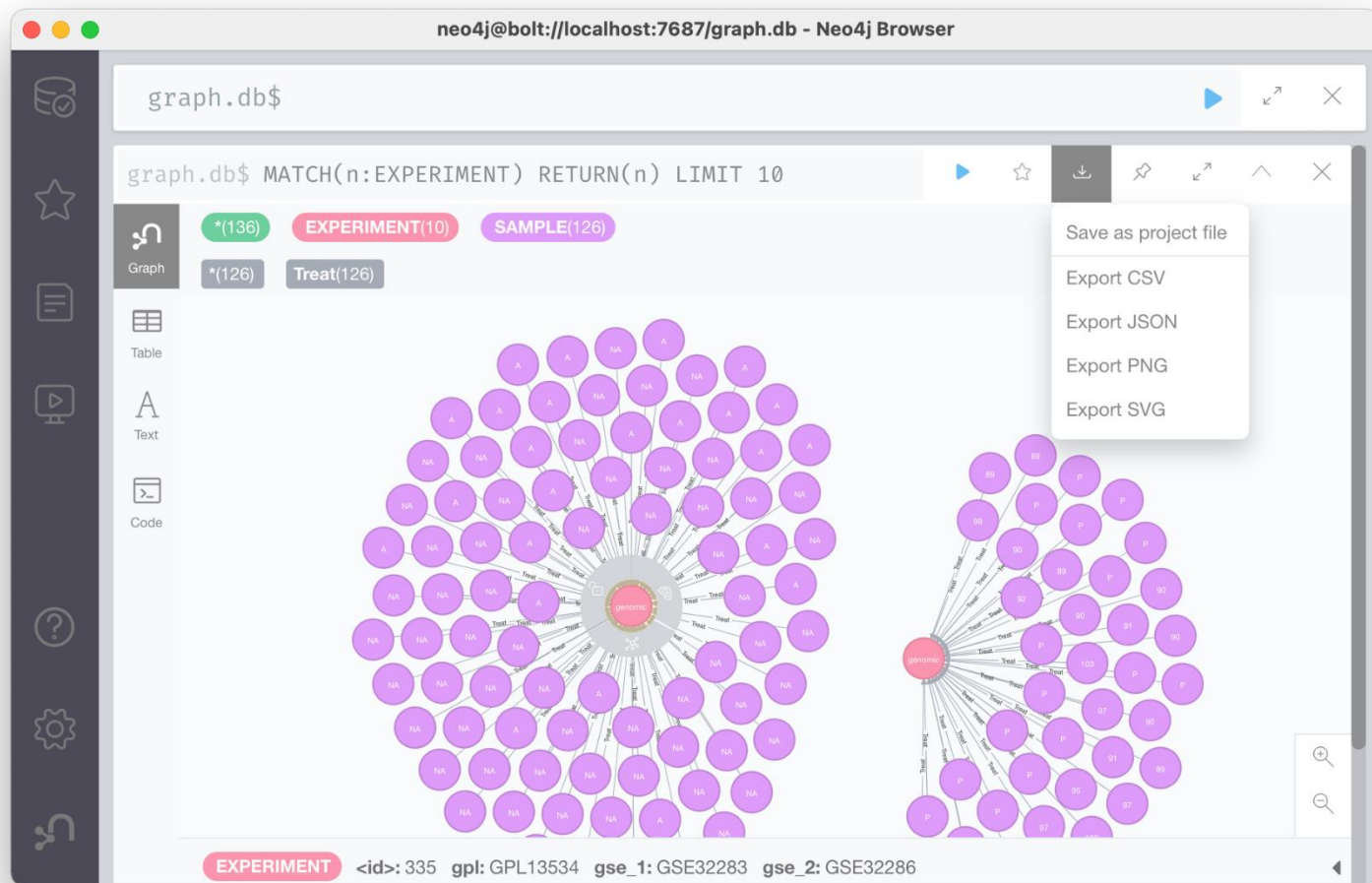
- Label associates a common name to a set of nodes or relationships.
  - ▣ A node or relationship can contain one or more labels.
  - ▣ We can create new labels to existing nodes or relationships.
  - ▣ We can remove the existing labels from the existing nodes or relationships.



- Left side node has a label: "Emp" and the right side node has a label: "Dept".
- Relationship between those two nodes also has a label: "WORKS\_FOR".

# Neo4j Data Browser

- We can access Neo4j Data Browser using the following URL **<http://localhost:7474/browser/>**



# Cypher Query Language (CQL)

- Neo4j CQL:
  - ▣ Is a **query language** for Neo4j Graph Database.
  - ▣ Is a declarative pattern-matching language.
  - ▣ Follows SQL like syntax.
  - ▣ Syntax is very simple and in human readable format.
  - ▣ Has commands to perform database operations.
  - ▣ Supports many clauses such as WHERE, ORDER BY, etc., to write very complex queries in an easy manner.
  - ▣ Supports some functions such as String, Aggregation.
    - it also supports some Relationship functions.

# Neo4j CQL (read) Clauses

Read Clause	Usage
MATCH	This clause is used to search the data with a specified pattern.
OPTIONAL MATCH	This is the same as match, the only difference being it can use nulls in case of missing parts of the pattern.
WHERE	This clause is used to add contents to the CQL queries.
LOAD CSV	This clause is used to import data from CSV files.



# Neo4j CQL (write) Clauses

Write Clause	Usage
CREATE	This clause is used to create nodes, relationships, and properties.
MERGE	This clause verifies whether the specified pattern exists in the graph. If not, it creates the pattern.
SET	This clause is used to update labels on nodes, properties on nodes and relationships.
DELETE	This clause is used to delete nodes and relationships or paths etc. from the graph.
REMOVE	This clause is used to remove properties and elements from nodes and relationships.
FOREACH	This class is used to update the data within a list.
CREATE UNIQUE	Using the clauses CREATE and MATCH, you can get a unique pattern by matching the existing pattern and creating the missing one.

# Neo4j CQL (general) Clauses

General Clause	Usage
RETURN	This clause is used to define what to include in the query result set.
ORDER BY	This clause is used to arrange the output of a query in order. It is used along with the clauses <b>RETURN</b> or <b>WITH</b> .
LIMIT	This clause is used to limit the rows in the result to a specific value.
SKIP	This clause is used to define from which row to start including the rows in the output.
WITH	This clause is used to chain the query parts together.
UNWIND	This clause is used to expand a list into a sequence of rows.
UNION	This clause is used to combine the result of multiple queries.
CALL	This clause is used to invoke a procedure deployed in the database.

# CQL Operators

Type	Operators
Mathematical	+, -, *, /, %, ^
Comparison	+, <>, <, >, <=, >=
Boolean	AND, OR, XOR, NOT
String	+
List	+, IN, [X], [X...Y]
Regular Expression	=~ (e.g., a.name =~ 'Tim.*')
String matching	STARTS WITH, ENDS WITH, CONSTRAINTS

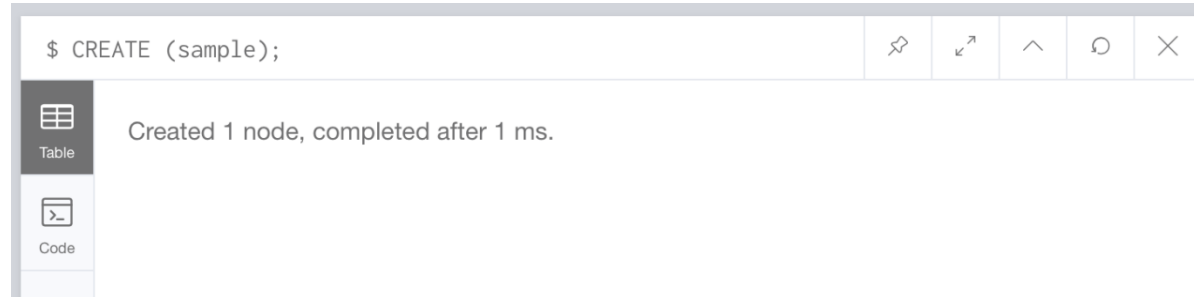
# Neo4j CQL Functions

CQL Function	Usage
String	They are used to work with String literals.
Aggregation	They are used to perform some aggregation operations on CQL Query results.
Relationship	They are used to get details of relationships such as startnode, endnode, etc.

# Creating a Single node

- The syntax for creating a node using CQL.

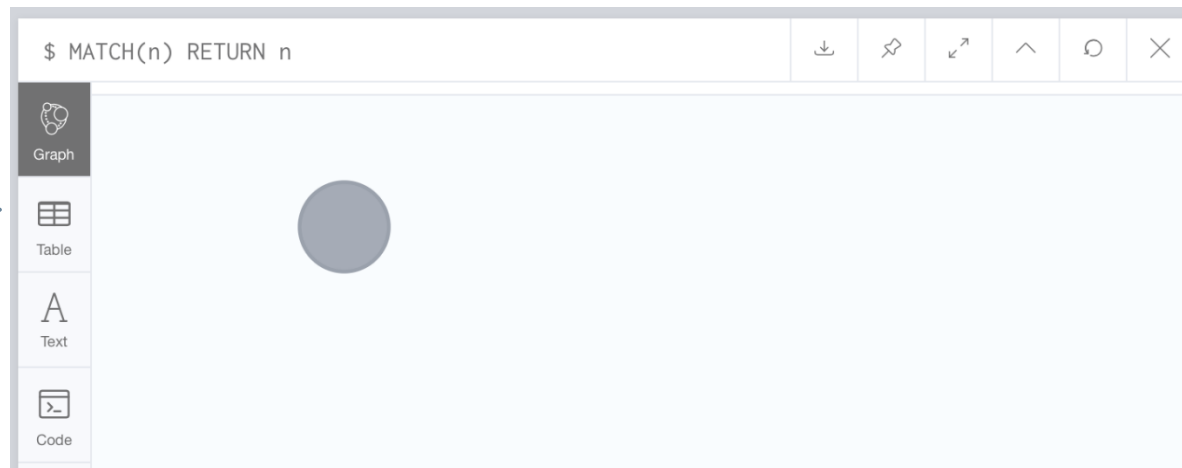
**\$ CREATE (sample);**



- To verify the creation of the node type.

**\$ MATCH (n) RETURN n;**

```
{  
  "identity": 0,  
  "labels": [],  
  "properties": {  
  
  }  
}
```



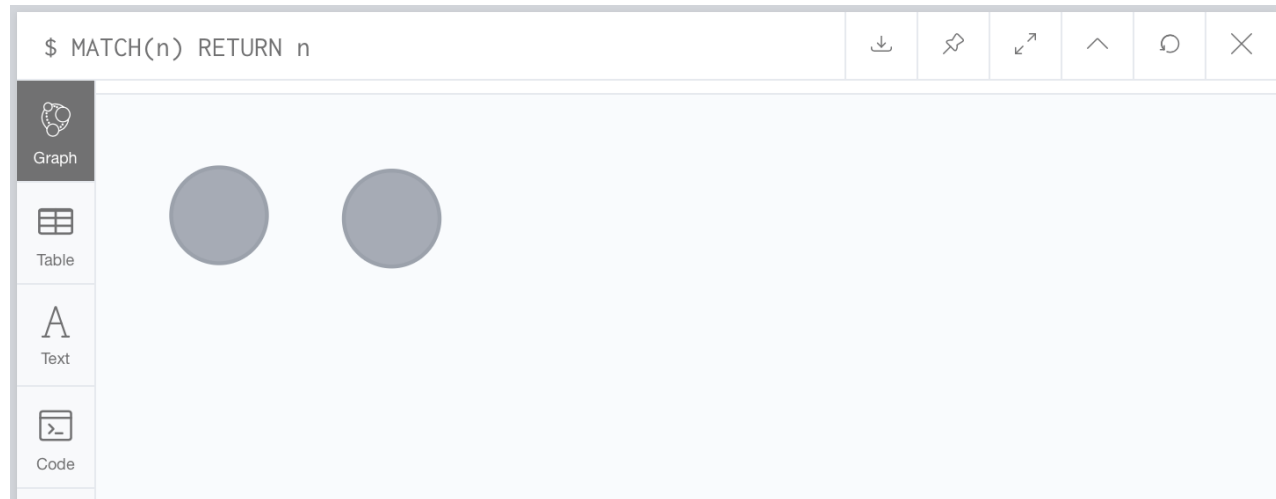
# Creating Multiple Nodes

- To create multiple nodes in Neo4j:

```
$ CREATE (sample1), (sample2);
```

```
// Return all nodes
```

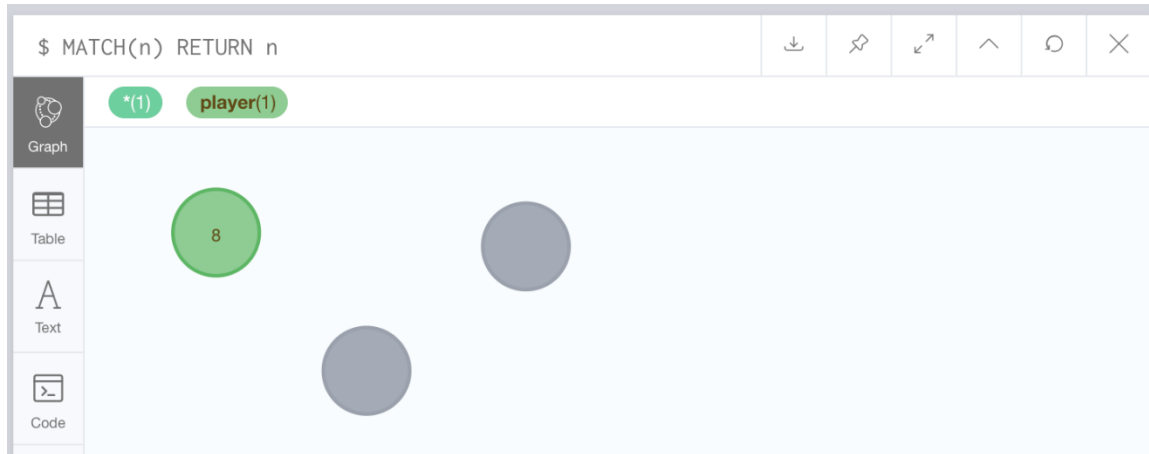
```
$ MATCH (n) RETURN n;
```



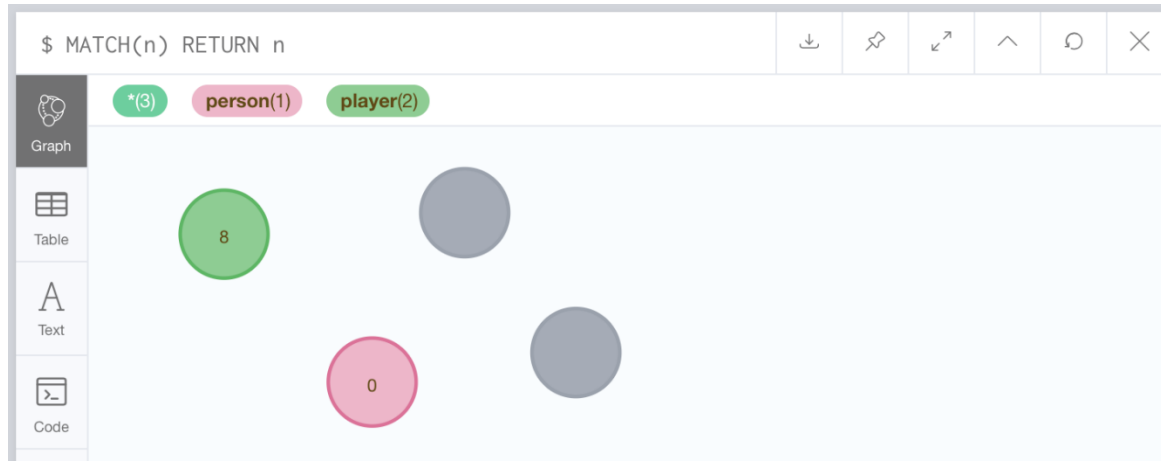
# Creating a Node with a (multiple) Label(s)

- To create a node with a label.

```
$ CREATE (sample:player);
```



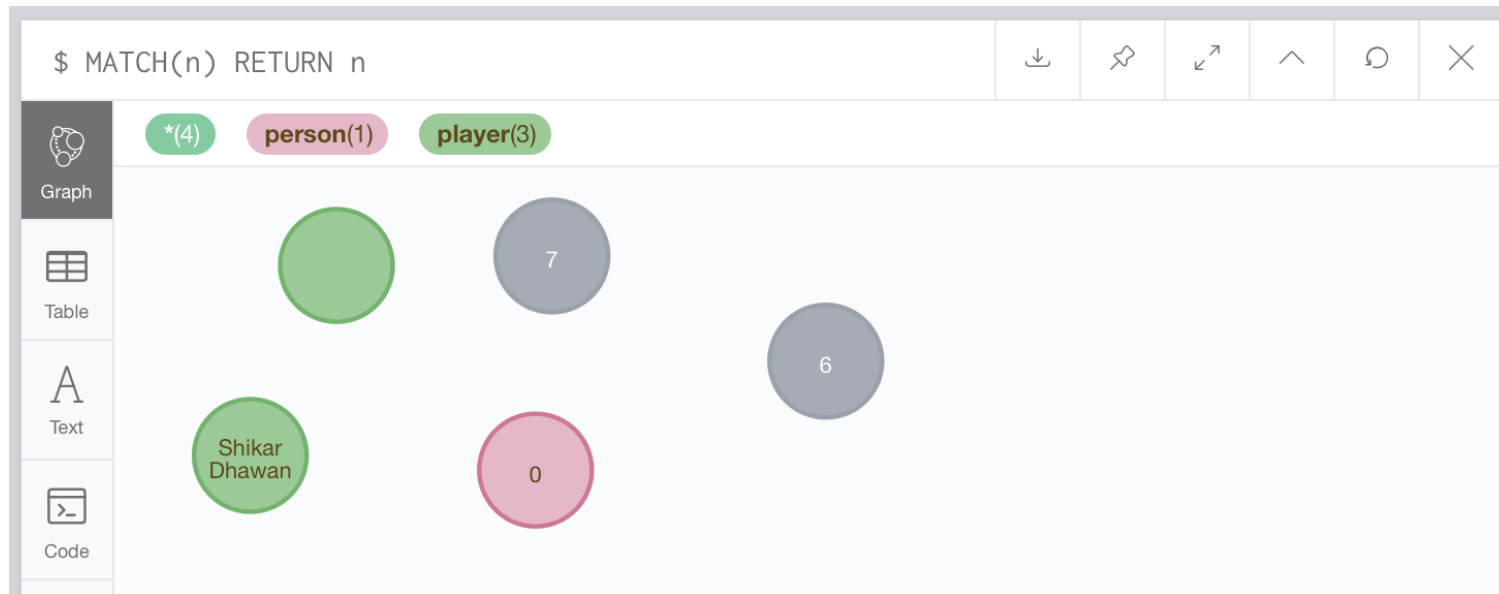
```
$ CREATE (sample:person:player);
```



# Create Node with Properties

- To create a node with properties:

```
$ CREATE (Dhawan:player {name: "Shikar Dhawan", YOB: 1985, POB: "Delhi"} );
```



- To delete nodes (without relationships):

```
$ MATCH (n) DELETE n;
```

in case of relationships

**ERROR** Neo.ClientError.Schema.ConstraintValidationFailed

Cannot delete node<0>, because it still has relationships. To delete this node, you must first delete its relationships.



# Creating a Relationship

- To create a relationship using the **create** clause:

```
$ CREATE (Dhawan:player {name: "Shikar Dhawan", YOB: 1985, POB: "Delhi"})  
CREATE (Ind:country {name: "India"})  
CREATE (Dhawan)-[r:BATSMAN_OF]->(Ind)  
RETURN Dhawan, Ind;
```

- To create a relationship between the existing nodes using the match clause:

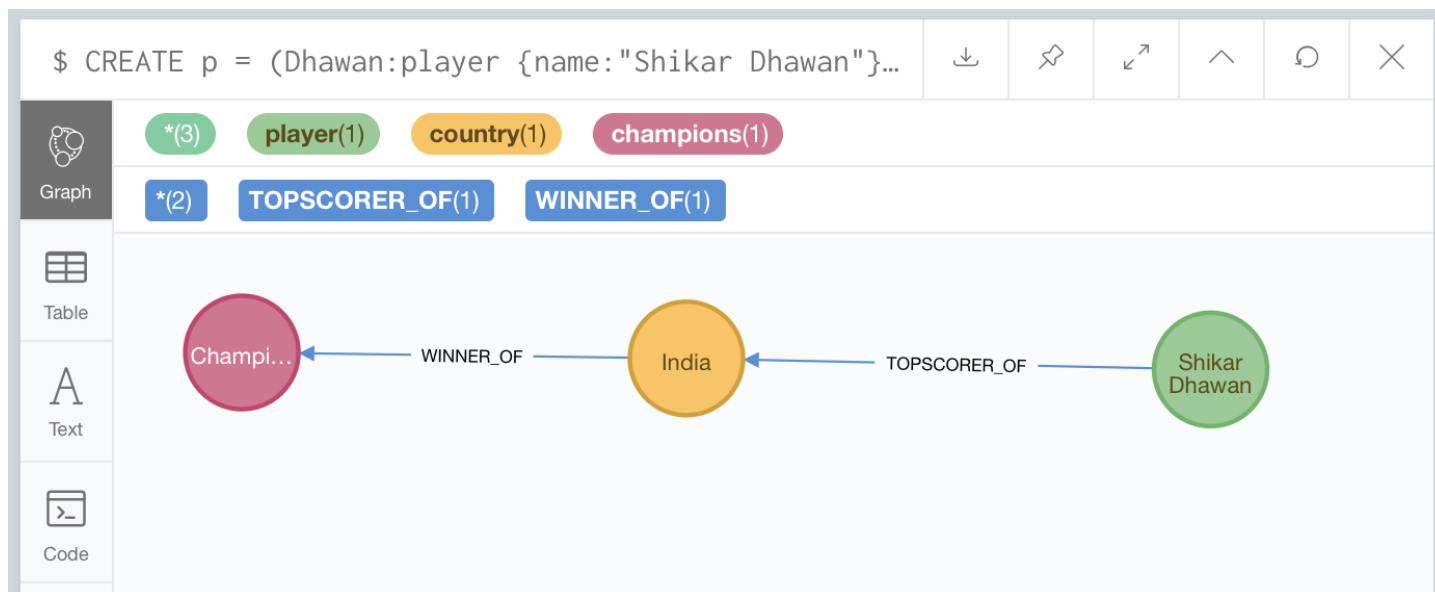
```
$ MATCH (a:player), (b:country) WHERE a.name = "Shikar Dhawan" AND b.name = "India"  
CREATE (a)-[r: BATSMAN_OF]->(b) RETURN a, b;
```

The screenshot displays two panels of a Cypher query editor. The left panel shows a query to create nodes and a relationship: `$ CREATE (Dhawan:player {name: "Shikar Dhawan", YOB: 1985, POB: "Delhi"}) CREATE (Ind:country {name: "India"}) CREATE (Dhawan)-[r:BATSMAN_OF]->(Ind) RETURN Dhawan, Ind;`. The right panel shows a query to match existing nodes and create a relationship: `$ MATCH (a:player), (b:Country) WHERE a.name = "Shikar..." CREATE (a)-[r: BATSMAN_OF]->(b) RETURN a, b;`. Both panels include a sidebar with icons for Graph, Table, Text, and Code. The graph visualizations show a blue node labeled 'India' and a green node labeled 'Shikar Dhawan' connected by a relationship labeled 'BATSMAN\_OF'.

# Creating a Complete Path

- To create a path in Neo4j using the CREATE clause:

```
$ CREATE p =  
    (Dhawan:player {name:"Shikar Dhawan"})-[:TOPSCORER_OF]->  
    (Ind:country {name:"India"})-[:WINNER_OF]->  
    (CT2019:tournament {name:"Champions Trophy 2019"})  
RETURN p;
```



# Merge Command

- MERGE command is a combination of CREATE command and MATCH command.
  - ▣ MERGE command searches for a given pattern in the graph.
    - If it exists, then it returns the results.
    - If it does not exist in the graph, then it creates a new node/relationship and returns the results.

# Example

```
$ CREATE (Dhawan:player {name: "Shikar Dhawan", YOB: 1985, POB: "Delhi"})  
CREATE (Ind:country {name: "India"})  
CREATE (Dhawan)-[r:BATSMAN_OF]->(Ind);
```

// Merging a node with a label

```
$ MERGE (Jadeja:player)  
RETURN Jadeja; // Return the existing node
```

```
$ MERGE (CT2019:tournament {name: "Champions Trophy 2019"})  
RETURN CT2019, labels(CT2019); // Return a new node
```

// Merging a node with properties

```
$ MERGE (Jadeja:player {name: "Ravindra Jadeja", YOB: 1988, POB: "NavagamGhed"})  
RETURN Jadeja; // Returns a new node
```

# OnCreate and OnMatch

- Using on-create and on-match, you can set properties for indicating whether the node is created or matched.

```
$ MERGE (Jadeja:player {name: "Ravindra Jadeja", YOB: 1988, POB: "NavagamGhed"})  
ON CREATE SET Jadeja.isCreated = "true"  
ON MATCH SET Jadeja.isFound = "true"  
RETURN Jadeja;
```

The screenshot displays the Neo4j Cypher query editor. The query entered is:

```
$ MERGE (Jadeja:player {name: "Ravindra Jadeja", YOB: 1988, POB: "NavagamGhed"}) ON CREATE SET ...
```

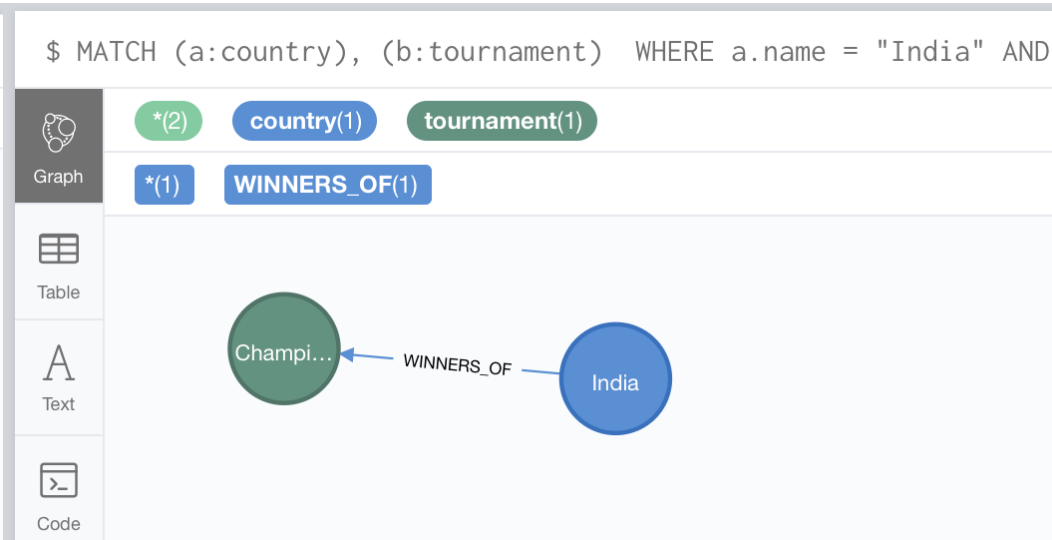
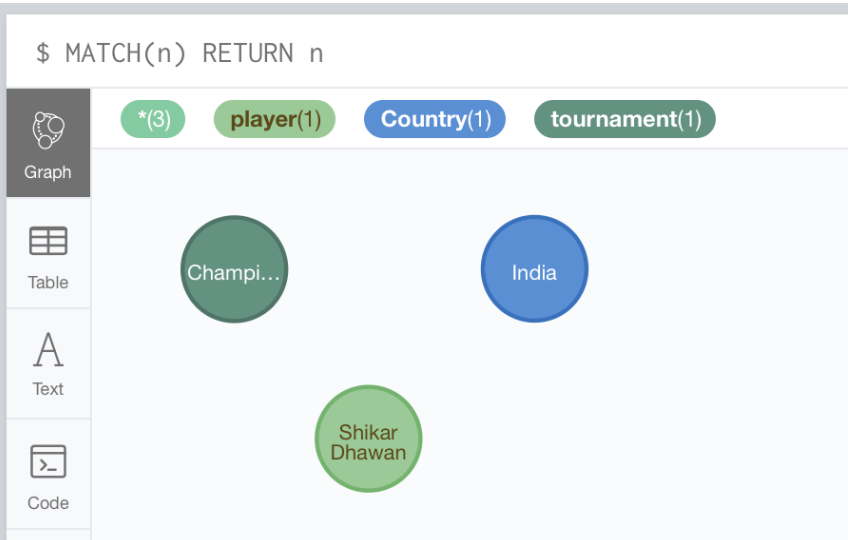
The interface shows a graph view with a single node labeled "Ravindra Jadeja". The node is represented by a green circle with a blue border. The node's properties are displayed in a table at the bottom:

player	<id>	POB	YOB	isCreated	name
player	28	NavagamGhed	1988	true	Ravindra Jadeja

# Merge a Relationship

- To merge the relationships using the merge clause:

```
$ CREATE (Dhawan:player {name: "Shikar Dhawan", YOB: 1985, POB: "Delhi"})  
CREATE (Ind:country {name: "India"})  
CREATE (CT2019:tournament {name:"Champions Trophy 2019"});  
  
$ MATCH (a:country), (b:tournament)  
WHERE a.name = "India" AND b.name = "Champions Trophy 2019"  
MERGE (a)-[r:WINNERS_OF]->(b)  
RETURN a, b;
```



# Set Clause

- Using **set** clause, you can add new properties to an existing Node or Relationship, and also add or update existing Properties values.

```
$ CREATE (Dhawan:player {name: "Shikar Dhawan", YOB: 1985, POB: "Delhi"});
```

```
// Add a new property
```

```
$ MATCH (Dhawan:player {name: "Shikar Dhawan", YOB: 1985, POB: "Delhi"})  
  SET Dhawan.highestscore = 187  
  RETURN Dhawan;
```

```
// Modify an existing property
```

```
$ MATCH (Dhawan:player {name: "Shikar Dhawan", YOB: 1985, POB: "Delhi"})  
  SET Dhawan.name = "Ravindra Jadeja"  
  RETURN Dhawan;
```

```
// Remove a property
```

```
$ MATCH (Dhawan:player {name: "Ravindra Jadeja", YOB: 1985, POB: "Delhi"})  
  SET Dhawan.POB = NULL  
  RETURN Dhawan;
```

# Setting Multiple Properties

- To create multiple properties in a node using the **set** clause.
  - ▣ You need to specify these key value pairs with commas.

```
$ MATCH (Jadeja:player {name: "Ravindra Jadeja", YOB: 1985})  
  SET Jadeja.POB = "NavagamGhed", Jadeja.HS = "90"  
  RETURN Jadeja;
```

```
$ CREATE (Anderson {name: "James Anderson", YOB: 1982, POB: "Burnely"});
```

// Setting a label on a node

```
$ MATCH (Anderson {name: "James Anderson", YOB: 1982, POB: "Burnely"})  
  SET Anderson: player  
  RETURN Anderson;
```

// Setting multiple Labels

```
$ MATCH (Anderson {name: "James Anderson", YOB: 1982, POB: "Burnely"})  
  SET Anderson: player: person  
  RETURN Anderson;
```



# Delete Clause

- To delete all the nodes and the relationships in the database using the **delete** clause.

```
$ MATCH (n) DETACH DELETE n;
```

```
$ CREATE (Ishant:player {name: "Ishant Sharma", YOB: 1988, POB: "Delhi"});
```

```
// Delete a specific node
```

```
$ MATCH (Ishant:player {name: "Ishant Sharma", YOB: 1988, POB: "Delhi"})  
  DELETE Ishant;
```

```
// Delete a node with specific id 2
```

```
$ MATCH (n) WHERE id(n) = 2 DELETE n;
```

- To delete a node and any relationship going to or from it, use **detach delete**.

# Remove Clause

- The **remove** clause is used to remove properties and labels from graph elements (Nodes or Relationships).
- The main difference between DELETE and REMOVE:
  - ▣ DELETE operation is used to delete nodes and associated relationships.
  - ▣ REMOVE operation is used to remove labels and properties.

```
$ CREATE (Dhoni:player {name: "Mahendra Dhoni", YOB: 1981, POB: "Ranchi"});
```

```
// Remove a property
```

```
$ MATCH (Dhoni:player {name: "Mahendra Dhoni", YOB: 1981, POB: "Ranchi"})  
  REMOVE Dhoni.POB  
  RETURN Dhoni;
```

```
// Remove a label
```

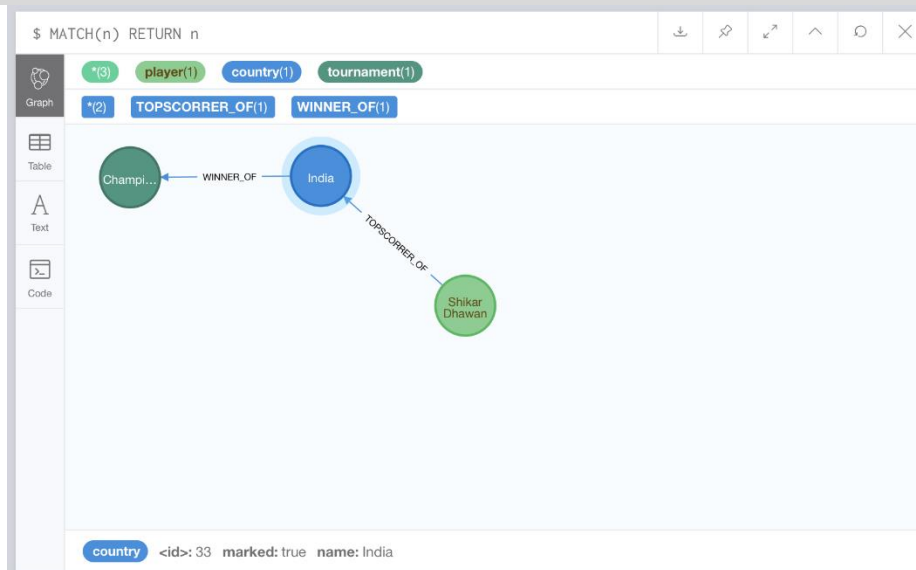
```
$ MATCH (Dhoni:player {name: "Mahendra Dhoni", YOB: 1981})  
  REMOVE Dhoni:player  
  RETURN Dhoni;
```

# Foreach Clause

- To update data within a list whether components of a path, or result of aggregation.

```
$ CREATE p = (Dhawan:player {name:"Shikar Dhawan"})-[:TOPSCORRER_OF]->
(Ind:country {name: "India"})-[:WINNER_OF]->
(CT2019:tournament {name: "Champions Trophy 2019"})
RETURN p;
```

```
$ MATCH p = (Dhawan)-[*]->(CT2019)
WHERE Dhawan.name = "Shikar Dhawan" AND CT2019.name = "Champions Trophy 2019"
FOREACH (n IN nodes(p) | SET n.marked = TRUE);
```



# Match Clause

- To retrieve nodes in the Neo4j database.

```
$ CREATE (Dhoni:player {name: "Mahendra Dhoni", YOB: 1981, POB: "Ranchi"})
CREATE (Ind:country {name: "India", result: "Winners"})
CREATE (CT2019:tournament {name: "Champions Trophy 2019"})

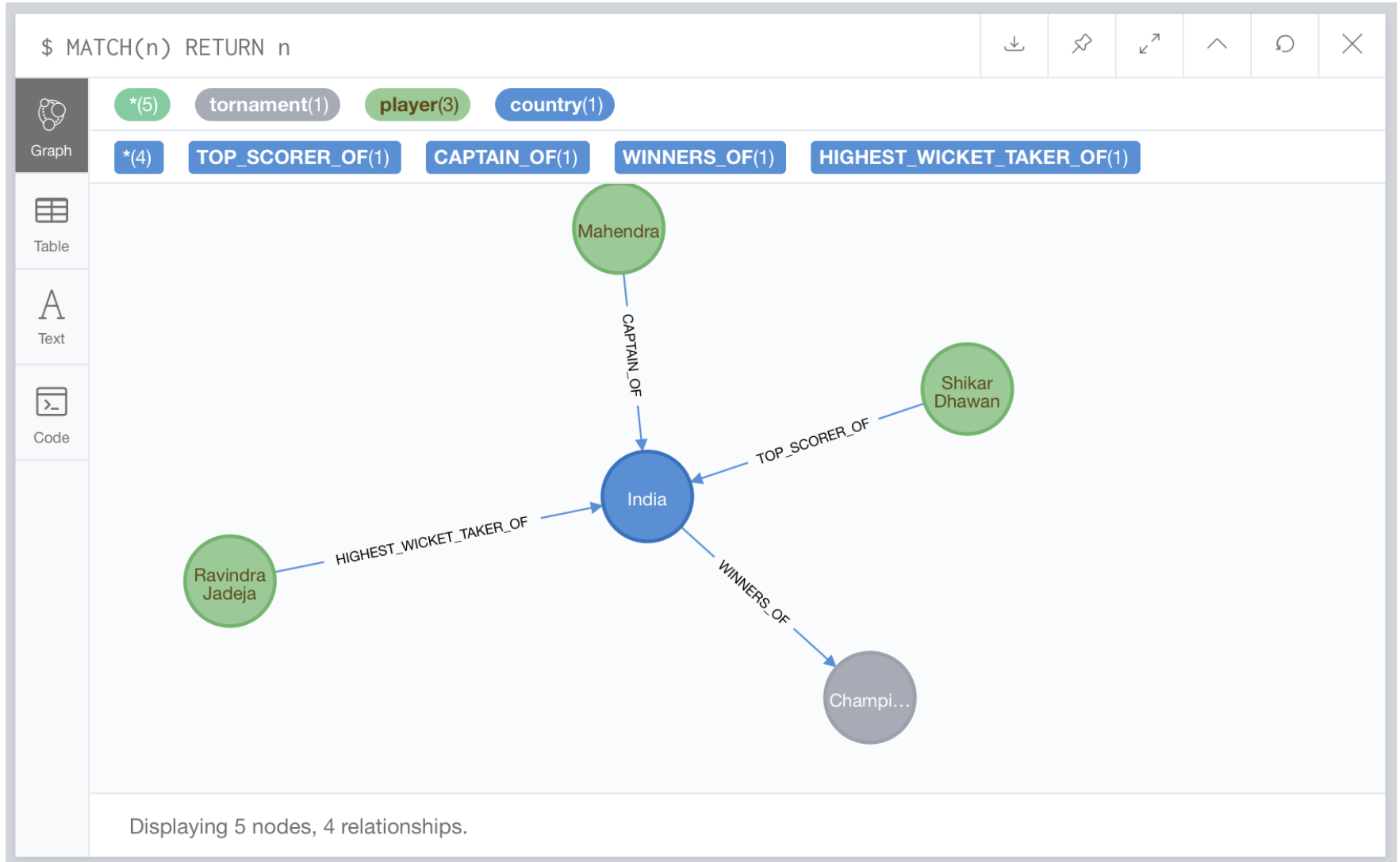
CREATE (Ind)-[r1:WINNERS_OF {NRR:0.938 ,pts:6}]->(CT2019)
CREATE (Dhoni)-[r2:CAPTAIN_OF]->(Ind)

CREATE (Dhawan:player {name: "Shikar Dhawan", YOB: 1995, POB: "Delhi"})
CREATE (Jadeja:player {name: "Ravindra Jadeja", YOB: 1988, POB: "NavagamGhed"})

CREATE (Dhawan)-[:TOP_SCORER_OF {Runs:363}]->(Ind)
CREATE (Jadeja)-[:HIGHEST_WICKET_TAKER_OF {Wickets:12}]->(Ind);

$ MATCH (n) RETURN n;
```

# Match Clause (2)



# Match Clause (3)

```
$ MATCH (n:player) RETURN n;
```

The image shows the Cypher Studio interface. At the top, the query editor contains the query: `$ MATCH (n:player) RETURN n`. Below the query editor, on the left, is a sidebar with icons for Graph, Table, Text, and Code. The 'Graph' icon is selected. The main area displays a graph visualization with three green circular nodes. The nodes are labeled 'Ravindra Jadeja', 'Mahendra', and 'Shikar Dhawan'. Above the graph, there are two green pill-shaped labels: `*(3)` and `player(3)`. The top right of the interface has a toolbar with icons for download, share, expand, zoom in, zoom out, and close.

```
$ MATCH (Ind:country {name: "India", result: "Winners"})<-[:TOP_SCORER_OF]-(n)
RETURN n;
```

```
$ MATCH (Ind:country {name: "India", result: "Winners"})<-[:TOP_SCORER_OF]-(n)
RETURN n.name;
```

The image shows the Cypher Studio interface. At the top, the query editor contains the query: `$ MATCH (Ind:country {name: "India", result: "Winners"})<-[: TOP_SCOR...`. Below the query editor, on the left, is a sidebar with icons for Table, Text, and Code. The 'Table' icon is selected. The main area displays a table with one column, `n.name`, and one row containing the value `"Shikar Dhawan"`. The top right of the interface has a toolbar with icons for download, share, expand, zoom in, zoom out, and close.

# Related Nodes

- The symbol – (-[ ]-) means related to, without regard to type or direction of the relationship.

```
$ MATCH (Ind:country {name:"India"}) -- (n)  
RETURN n.name;
```

- To consider the direction of a relationship we can use --> or <--.

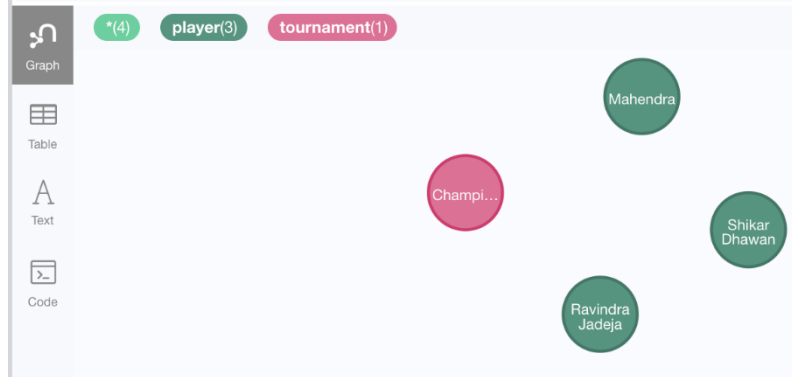
```
$ MATCH (Ind:country {name:"India"}) --> (n)  
RETURN n;
```

```
$ MATCH (Ind:country {name:"India"}) <-- (n)  
RETURN n;
```

```
neo4j$ MATCH (Ind:country {name:"India"}) <-- (n) RETURN n;
```



```
neo4j$ MATCH (Ind:country {name:"India"}) -- (n) RETURN n;
```



```
neo4j$ MATCH (Ind:country {name:"India"}) --> (n) RETURN n;
```



# Optional Match Clause

- The **optional match** clause is used to search for the pattern described in it, while using nulls for missing parts of the pattern.
  - OPTIONAL MATCH is similar to the match clause, the only difference being it returns null as a result of the missing parts of the pattern.

```
$ MATCH (a:tournament {name: "Champions Trophy 2019"})
```

```
OPTIONAL MATCH (a)-->(x)
```

```
RETURN x;
```

```
$ MATCH (a:tournament {name: "Champions Trophy 2019"}) OPTIONAL MATC...
```



Table



Text

x

null

```
$ MATCH (a:tournament {name: "Champions Trophy 2019"})
```

```
OPTIONAL MATCH (a)--(x)
```

```
RETURN x;
```

```
$ MATCH (a:tournament {name: "Champions Trophy 2019"}) OPTIONAL MATC...
```



Graph



Table



Text

\*(1)

country(1)

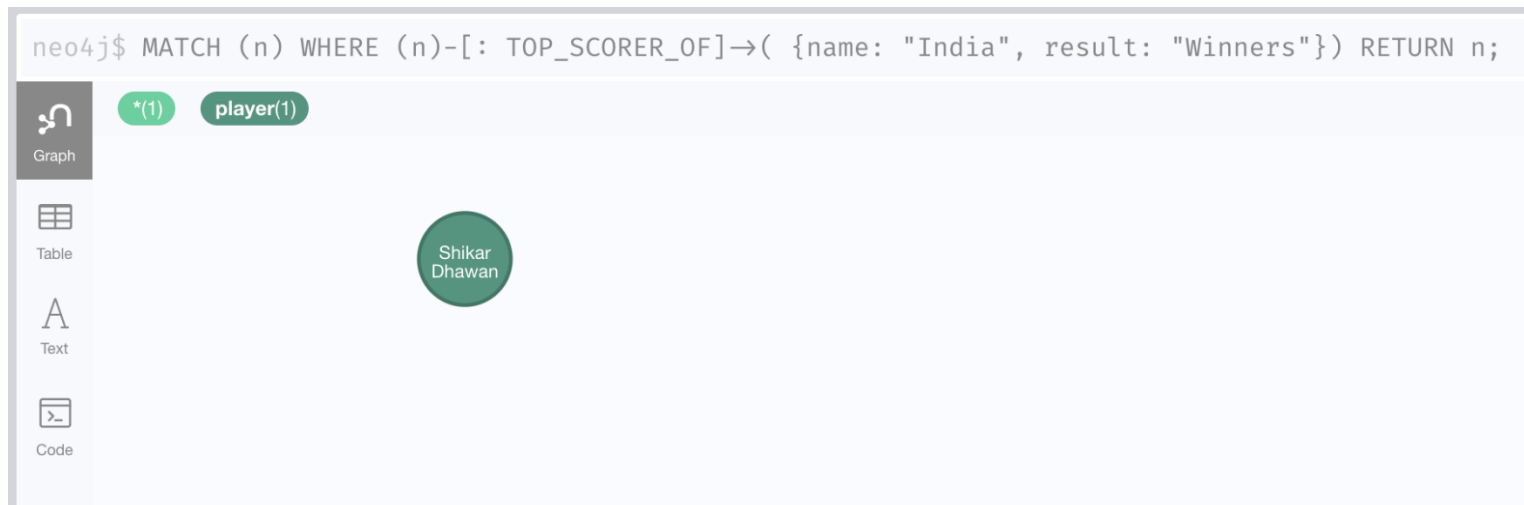
India



# Using Relationship with Where Clause

- To use **where** clause to filter the nodes using the relationships.

```
$ MATCH (n)  
  WHERE (n)-[: TOP_SCORER_OF]->( {name: "India", result: "Winners"})  
  RETURN n;
```

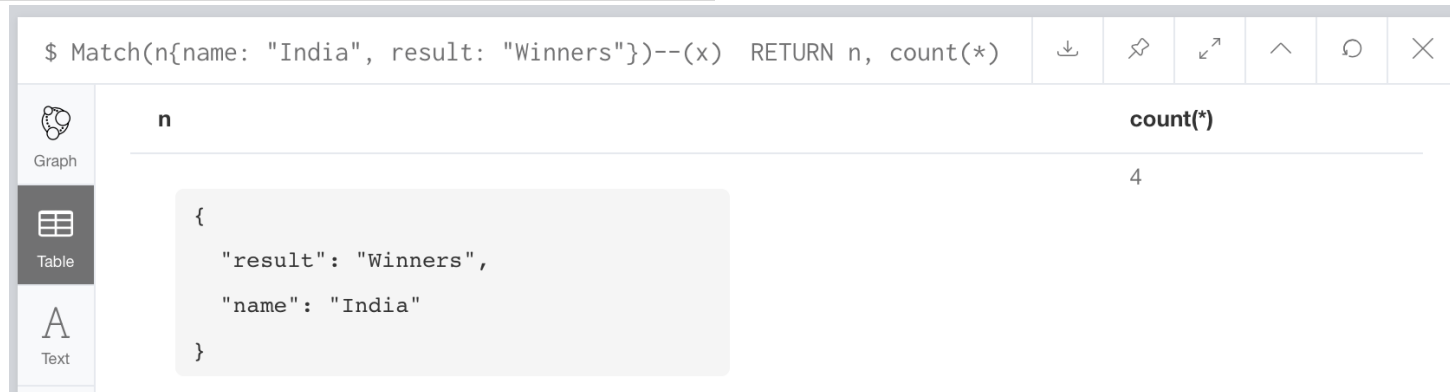


# Count Function

- The **count()** function is used to count the number of rows.

```
$ MATCH (n {name: "India", result: "Winners"})--(x)
RETURN n, count(*);
```

show as table



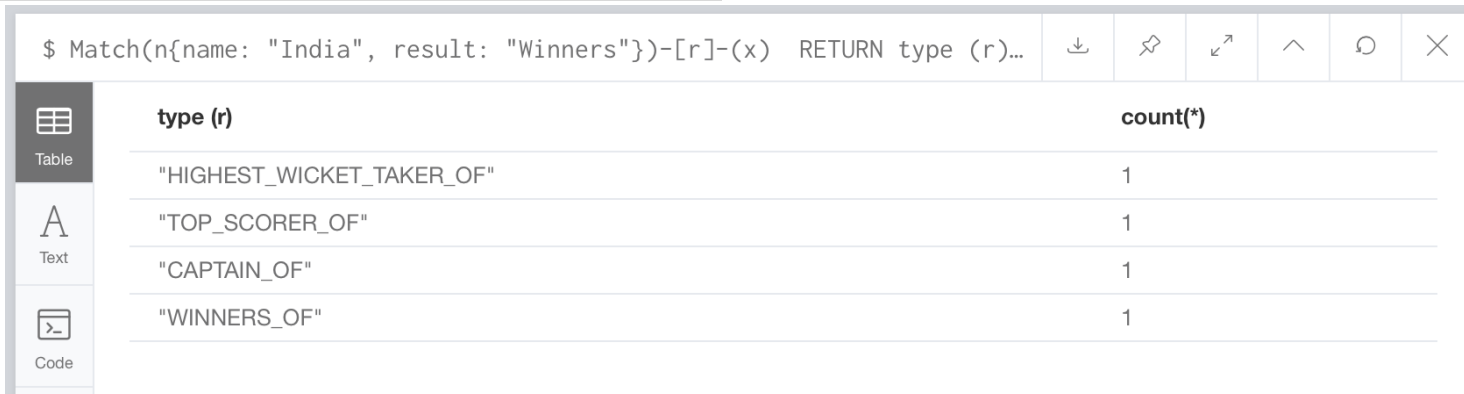
The interface shows a query editor with the following query:

```
$ Match(n{name: "India", result: "Winners"})--(x) RETURN n, count(*)
```

The results are displayed in table view with two columns: **n** and **count(\*)**. The **n** column contains a JSON object representing the match, and the **count(\*)** column shows the value 4.

n	count(*)
{ "result": "Winners", "name": "India" }	4

```
$ MATCH (n{name: "India", result: "Winners"})-[r]-(x)
RETURN type(r), count(*);
```



The interface shows a query editor with the following query:

```
$ Match(n{name: "India", result: "Winners"})-[r]-(x) RETURN type (r)...
```

The results are displayed in table view with two columns: **type (r)** and **count(\*)**. The **type (r)** column lists the relationship types, and the **count(\*)** column shows the count for each type.

type (r)	count(*)
"HIGHEST_WICKET_TAKER_OF"	1
"TOP_SCORER_OF"	1
"CAPTAIN_OF"	1
"WINNERS_OF"	1

# Order By Clause

```
$ MATCH (n:player)  
RETURN n.name, n.YOB  
ORDER BY n.YOB;
```

```
$ MATCH (n:player)  
RETURN n.name, n.YOB  
ORDER BY n.YOB, n.name;
```

```
$ MATCH (n:player)  
RETURN n.name, n.YOB  
ORDER BY n.YOB DESC;
```

```
$ MATCH (n:player) RETURN n.name, n.YOB ORDER BY n.YOB DESC;
```



Table



Text

n.name

n.YOB

"Shikar Dhawan"

1995

"Ravindra Jadeja"

1988

"Mahendra Dhoni"

1981

# Limit , Skip and With

```
$ MATCH (n:player)
  RETURN n.name, n.YOB
  ORDER BY n.YOB DESC
  LIMIT 2;
```

```
$ MATCH (n:player)
  RETURN n.name, n.YOB
  ORDER BY n.YOB DESC
  LIMIT toInteger(2 * rand()) + 1;
```

```
$ MATCH (n:player)
  RETURN n.name, n.YOB
  ORDER BY n.YOB DESC
  SKIP 1 LIMIT 1;
```

```
$ MATCH (n:player)
  WITH n
  ORDER BY n.name DESC LIMIT 2
  RETURN collect(n.name)
```

- The **with** clause allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.
- The function **collect()** returns a single aggregated list containing the values returned by an expression.

# String and Aggregation Functions

Function & Description	
<b>toUpper</b> It is used to change all letters into upper case letters.	<code>\$ MATCH (n:player) RETURN <b>toUpper(n.name)</b>, n.YOB, n.POB;</code>
<b>toLower</b> It is used to change all letters into lower case letters.	
<b>substring</b> It is used to get substring of a given String.	<code>\$ MATCH (n:player) RETURN <b>substring(n.name, 0, 5)</b>, n.YOB, n.POB;</code>
<b>replace</b> It is used to replace a substring with a given substring of a String.	
<b>count</b> It returns the number of rows returned by MATCH command.	
<b>max</b> It returns the maximum value from a set of rows returned by MATCH command.	
<b>min</b> It returns the minimum value from a set of rows returned by MATCH command.	<code>\$ MATCH (n:player) RETURN <b>min(n.YOB)</b>;</code>
<b>sum</b> It returns the summation value of all rows returned by MATCH command.	
<b>avg</b> It returns the average value of all rows returned by MATCH command.	

# Unwind

- With **unwind**, you can transform any list back into individual rows.
  - ▣ These lists can be parameters that were passed in, previously collected result or other list expressions.
- One common usage of unwind is to create distinct lists.

```
$ UNWIND [1, 2, 3, null] AS x  
RETURN x, 'val' AS y;
```

```
$ WITH [1, 1, 2, 2] AS coll UNWIND coll AS x  
WITH DISTINCT x  
RETURN collect(x) AS setOfVals;
```

```
neo4j$ UNWIND [1, 2, 3, null] AS x RETURN x, 'val' AS y;
```

	x	y
1	1	"val"
2	2	"val"
3	3	"val"
4	null	"val"

```
neo4j$ WITH [1, 1, 2, 2] AS coll UNWIND coll AS x WITH DISTINCT x RETURN collect(x) AS setOfVals;
```

	setOfVals
1	[1, 2]

# Union

- ❑ **union** combines the results of two or more queries into a single result set that includes all the rows that belong to all queries in the union.
  - ❑ The number and the names of the columns must be identical in all queries combined by using union.

```
$ MATCH (n:country)
RETURN n.name AS name
UNION ALL
MATCH (n:player)
RETURN n.POB AS name;
```

By not including **ALL** in the union, duplicates are removed from the combined result set

```
neo4j$ MATCH (n:country) RETURN n.name AS name UNION ALL MATCH (n:player) RETURN n.POB AS name;
```

	name
1	"India"
2	"Ranchi"
3	"Delhi"
4	"NavagamGhed"

# Index

- Neo4j supports Indexes on node or relationship properties to improve the performance of the application.
  - ▣ We can create indexes on properties for all nodes, which have the same label name.

```
$ CREATE (Dhawan:player {name: "Shikar Dhawan", YOB: 1995, POB: "Delhi"});
```

```
$ CREATE INDEX ON :player(Dhawan);
```

```
$ CREATE INDEX ON :player(name);
```

```
$ CREATE INDEX ON :player(name, YOB);
```

```
$ DROP INDEX ON :player(Dhawan);
```

```
// Show all indexes
```

```
$ CALL db.indexes
```

Procedures are called using the **CALL** clause



# Create Unique Constraint

- **create** command always creates a new node or relationship which means even though you use the same values, it inserts a new row.
- As per our application requirements for some nodes or relationships, we have to avoid this duplication.
  - ▣ We should use some database constraints to create a rule on one or more properties of a node or relationship.

# Create Unique Constraint (2)

```
$ CREATE (Dhawan:player {id:001, name: "Shikar Dhawan", YOB: 1995, POB: "Delhi"})  
CREATE (Jonathan:player {id:002, name: "Jonathan Trott", YOB: 1981, POB: "CapeTown"})  
CREATE (Sangakkara:player {id:003, name: "Kumar Sangakkara", YOB: 1977, POB: "Matale"})  
CREATE (Rohit:player {id:004, name: "Rohit Sharma", YOB: 1987, POB: "Nagpur"})  
CREATE (Virat:player {id:005, name: "Virat Kohli", YOB: 1988, POB: "Delhi"});
```

```
$ CREATE CONSTRAINT ON (n:player) ASSERT n.id IS UNIQUE;
```

```
$ CREATE (Jadeja:player {id:002, name: "Ravindra Jadeja", YOB: 1988, POB: "NavagamGhed"});
```

```
$ CREATE (Jadeja:player {id:002, name: "Ravindra Jadeja", YOB: 1988, POB: "NavagamGhed"})
```



Error

## ERROR

Neo.ClientError.Schema.ConstraintValidationFailed

```
Neo.ClientError.Schema.ConstraintValidationFailed: Node(20) already exists with label `player` and property  
`id` = 2
```

```
$ DROP CONSTRAINT ON (n:player) ASSERT n.id IS UNIQUE;
```

# Example

CSV: 64224 rows x 22 columns

```
$ USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:///HMETHYL.csv" AS row FIELDTERMINATOR ";"
MERGE (:DISEASE {label_disease:row.label_disease, disease:row.disease,
specific_disease:row.specific_disease, genetic_variant:row.genetic_variant, stage:row.stage,
tissue:row.tissue})
MERGE (:SAMPLE {gsm:row.gsm, gender:row.gender, age:row.age, ethnicity:row.ethnicity,
treatment_sample:row.treatment_sample, marker:row.marker})
MERGE (:EXPERIMENT {gse_1:row.gse_1, gse_2:row.gse_2, gse_description:row.gse_description,
gpl:row.gpl, type:row.type, organism:row.organism})
MERGE (:CELL_LINE {immortalization:row.immortalization, cell_type:row.cell_type,
cell_line:row.cell_line, label_cell_line:row.label_cell_line});

$ CREATE CONSTRAINT ON (s:SAMPLE) ASSERT s.gsm IS UNIQUE;
$ CREATE INDEX ON :DISEASE(label_disease, disease, specific_disease, genetic_variant, stage,
tissue);

$ CREATE INDEX ON :DISEASE(label_disease);

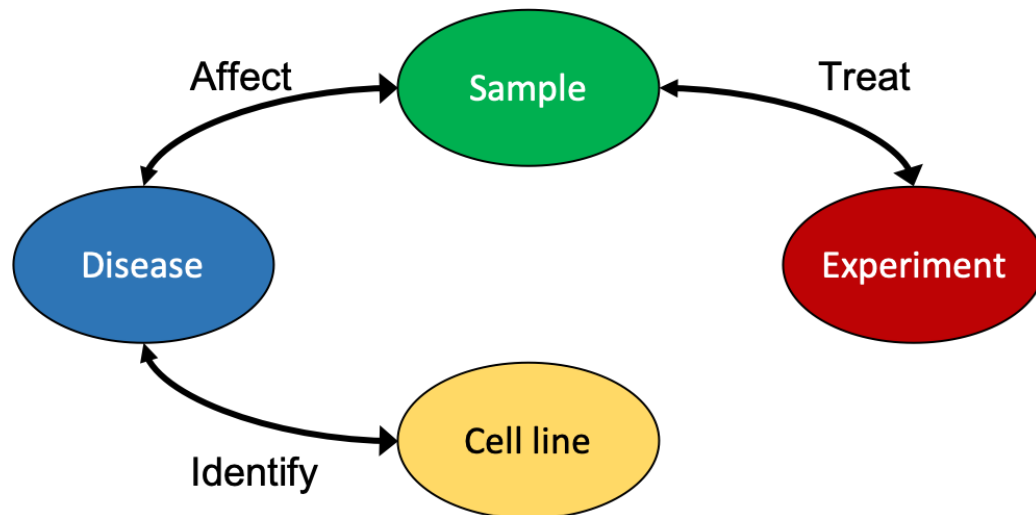
$ CREATE INDEX ON :DISEASE(label_disease, disease);

$ CREATE INDEX ON :EXPERIMENT (gse_1, gse_2, gse_description, gpl, type, organism);

$ CREATE INDEX ON :CELL_LINE (immortalization, cell_type, cell_line, label_cell_line);
...
```

# Example (2)

```
$ USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:///HMETHYL.csv" AS row FIELDTERMINATOR ";";
MATCH (d:DISEASE {label_disease:row.label_disease, disease:row.disease,
specific_disease:row.specific_disease, genetic_variant:row.genetic_variant, stage:row.stage,
tissue:row.tissue})
MATCH (s:SAMPLE {gsm:row.gsm})
MATCH (e:EXPERIMENT {gse_1:row.gse_1, gse_2:row.gse_2,
gse_description:row.gse_description, gpl:row.gpl, type:row.type,organism:row.organism})
MATCH (c:CELL_LINE {immortalization:row.immortalization, cell_type:row.cell_type,
cell_line:row.cell_line, label_cell_line:row.label_cell_line})
MERGE (d)-[:Affect]->(s) MERGE (d)<-[:Affect]-(s)
MERGE (s)-[:Treat]->(e) MERGE (s)<-[:Treat]-(e)
MERGE (d)-[:Identify]->(c); MERGE (d)<-[:Identify]-(c);
```



Entity	Nodes	Total
Disease	13853	81966
Sample	64224	
Experiment	723	
Cell line	3266	
Relationships	145114	145114

# Neo4j for Java Developers

- The standalone Neo4j Server can be installed on any machine and then accessed via its **binary "Bolt" protocol** through our official driver or any bolt-enabled drivers.



- Download driver from:  
<https://search.maven.org/artifact/org.neo4j.driver/neo4j-java-driver>

# Neo4j and Java

```
import org.neo4j.driver.v1.AuthTokens;
import org.neo4j.driver.v1.Driver;
import org.neo4j.driver.v1.GraphDatabase;
import org.neo4j.driver.v1.Session;
import org.neo4j.driver.v1.StatementResult;
import org.neo4j.driver.v1.Transaction;
import org.neo4j.driver.v1.TransactionWork;

import static org.neo4j.driver.v1.Values.parameters;

public class HelloWorldExample implements AutoCloseable
{
    private final Driver driver;

    public HelloWorldExample( String uri, String user, String password ) {
        driver = GraphDatabase.driver( uri, AuthTokens.basic( user, password ) );
    }

    public void close() throws Exception {
        driver.close();
    }

    public void printGreeting( final String message ) {
        try ( Session session = driver.session() )
        {
            String greeting = session.writeTransaction( new TransactionWork<String>()
            {
                public String execute( Transaction tx ) {
                    StatementResult result = tx.run( "CREATE (a:Greeting) " +
                                                    "SET a.message = $message " +
                                                    "RETURN a.message + ', from node ' + id(a)",
                                                    parameters( "message", message ) );
                    return result.single().get( 0 ).asString();
                }
            } );
            System.out.println( greeting );
        }
    }

    public static void main( String... args ) throws Exception {
        try ( HelloWorldExample greeter =
            new HelloWorldExample( "bolt://localhost:7687", "neo4j", "password" ) ) {
            greeter.printGreeting( "hello, world" );
        }
    }
}
```