Databases oriented to Neo4j graphs

Extension to Databases

Professor: Pablo Ramos

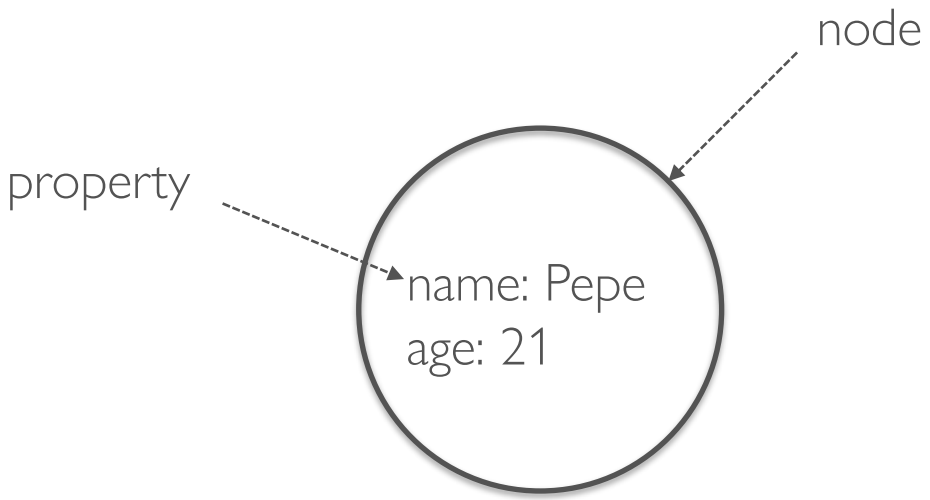
pablo.ramos@u-tad.com

INTRODUCTION

- Using graphs to store information:
  - Nodes: represents information entities
  - Edges: represent relationships between nodes.
  - Properties: Both nodes and edges have associated fields that provide additional information.
- Patterns
  - Graphs allow us to describe patterns in a way similar to how humans structure information.

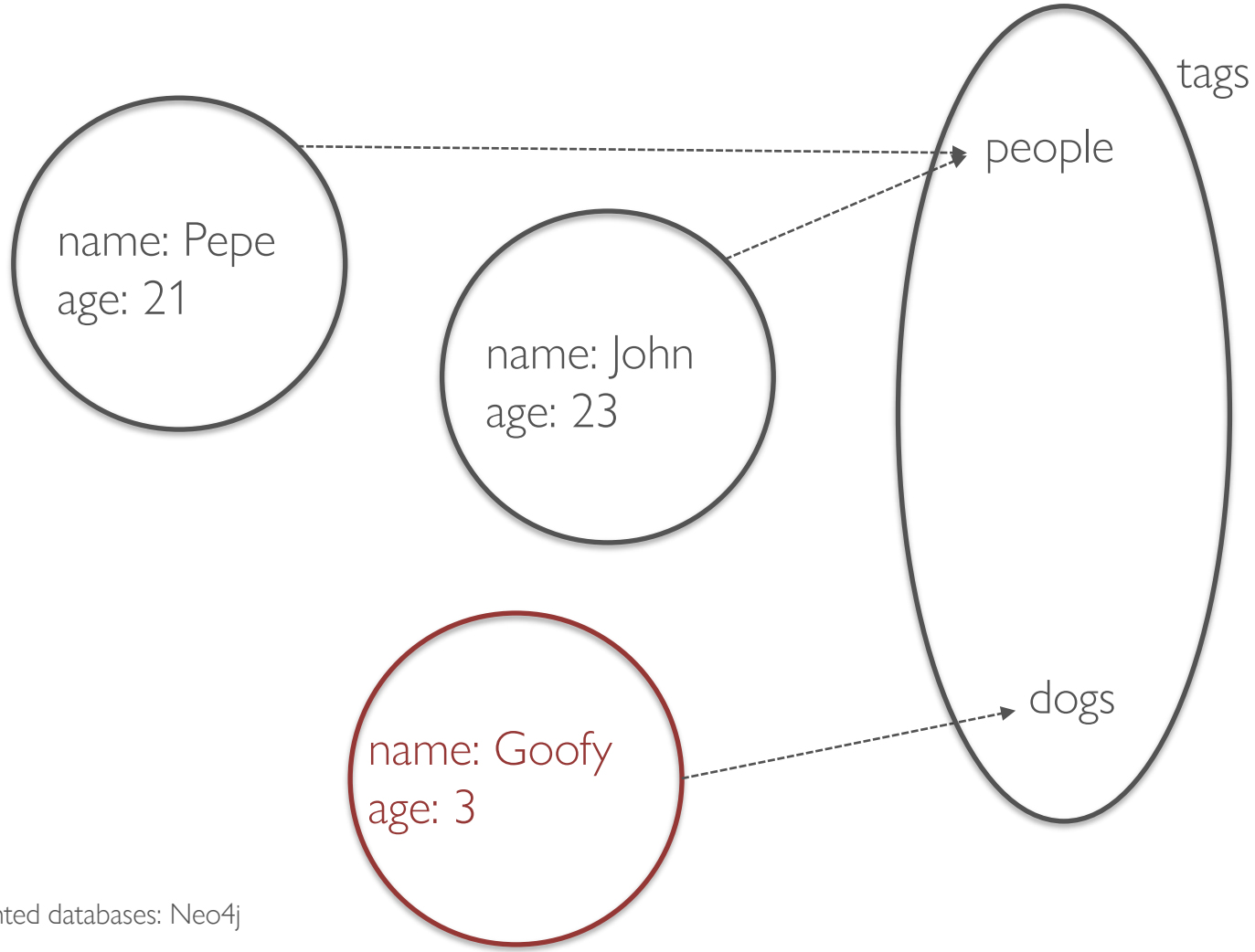
ELEMENTS: Node and properties

- A node represents an information entity.
- Nodes have a set of properties that contain information about the node.



ELEMENTS : Tags

- Nodes are classified by labels
  - Each node can have zero or more labels.

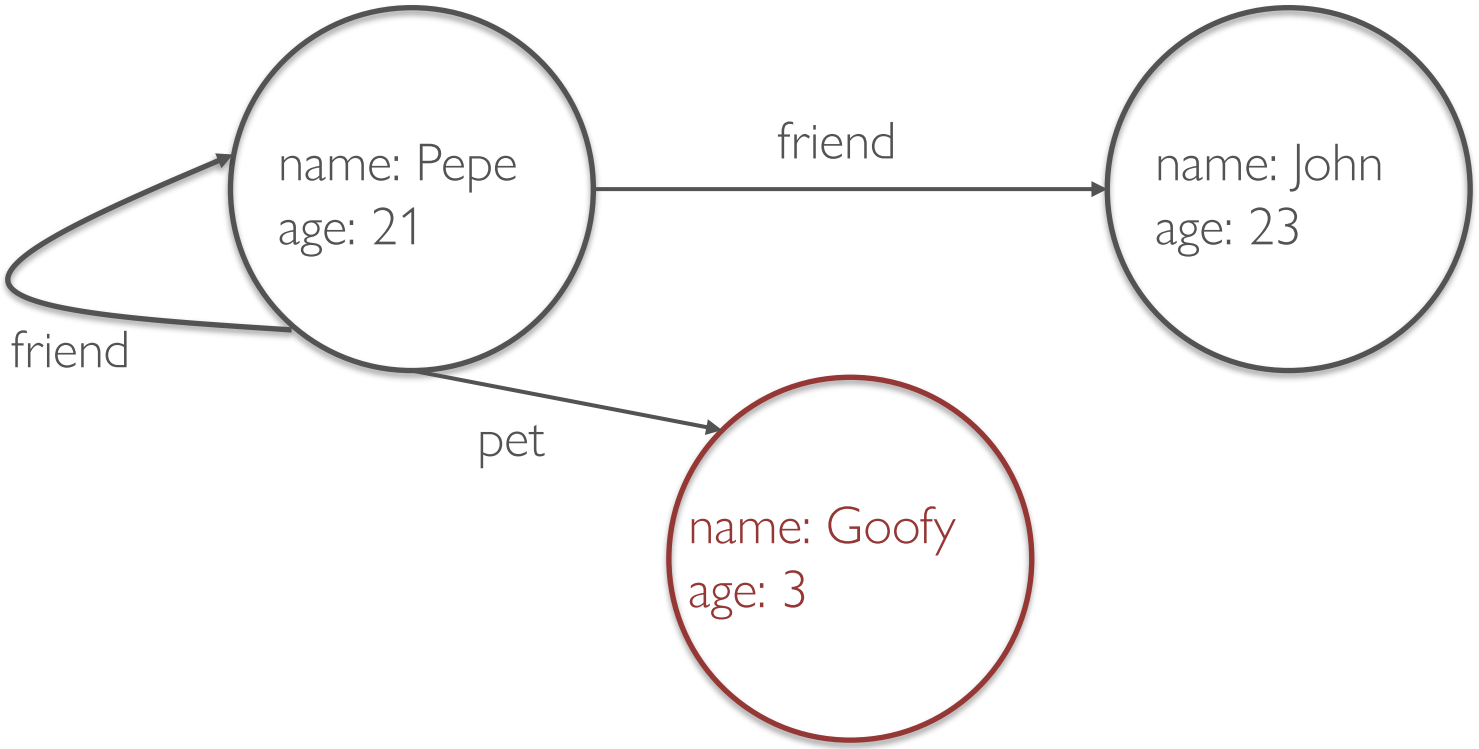


ELEMENTS : Tags

- Nodes are classified by labels
  - Each node can have zero or more labels.
  - Tags can be added or removed at runtime.
  - Nodes are grouped into sets based on their labels.
    - People = {Pepe, Juan}
    - Dogs = { Goofy }
  - Queries can be made by searching by tag.

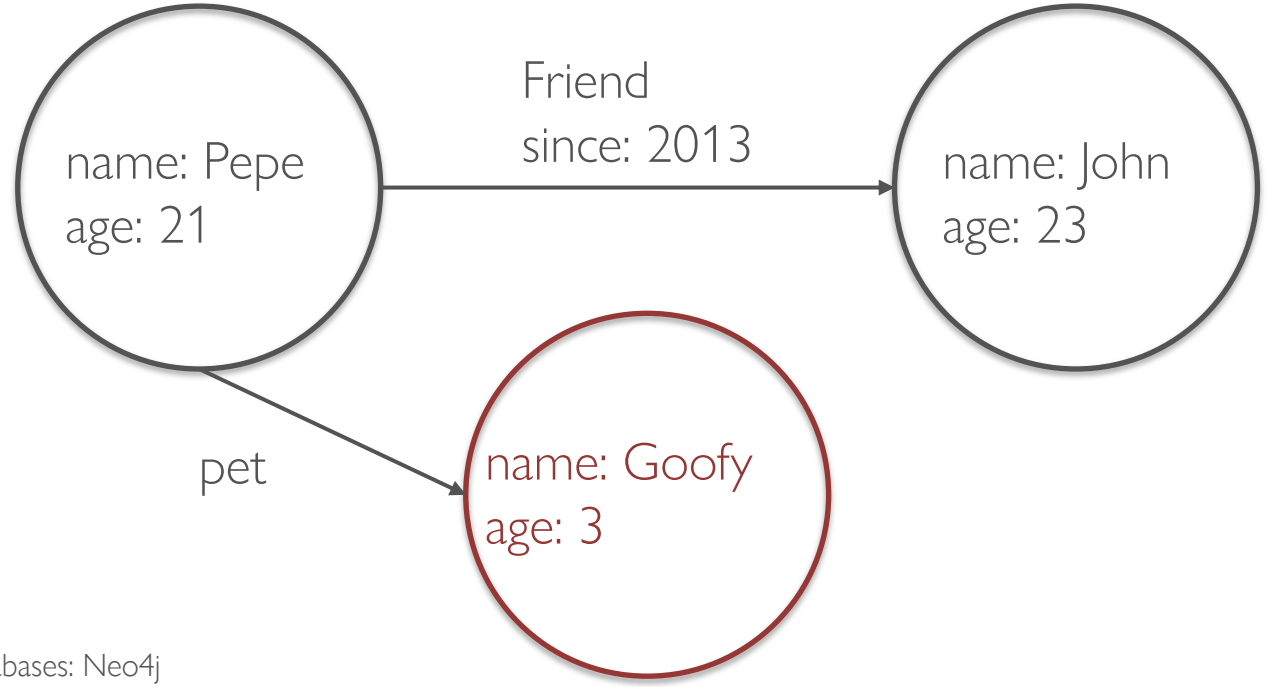
ELEMENTS : Relationships

- Relationships are edges between nodes that connect nodes and express some kind of relationship between the entities they link.
  - Relationships have direction.
  - Relationships are of a type.



ELEMENTS : Relationships

- Relationships are edges between nodes that express some kind of relationship between the entities they link.
  - Relationships have direction.
  - Relationships are of a type.
  - Relationships have properties that provide more information about the type of relationship they have.

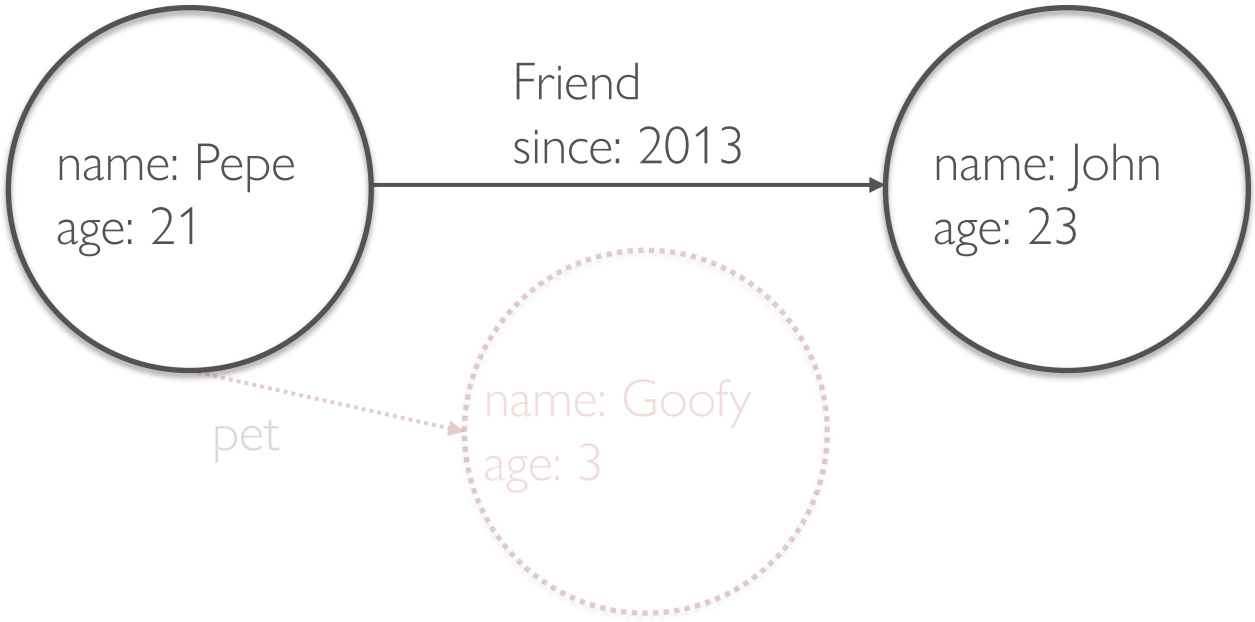


ELEMENTS : Properties

- The properties of nodes and relationships are key-value pairs.
  - Keys are text strings
  - Values can be:
    - Numbers
    - Text strings
    - Booleans
    - Collections of numbers, text strings or booleans.

QUERIES: *Traversal* (Route)

- *Traversal* is the way queries are performed in Neo4j.
- A traversal is performed through the graph following the specified rules.
  - Find friends of Pepe



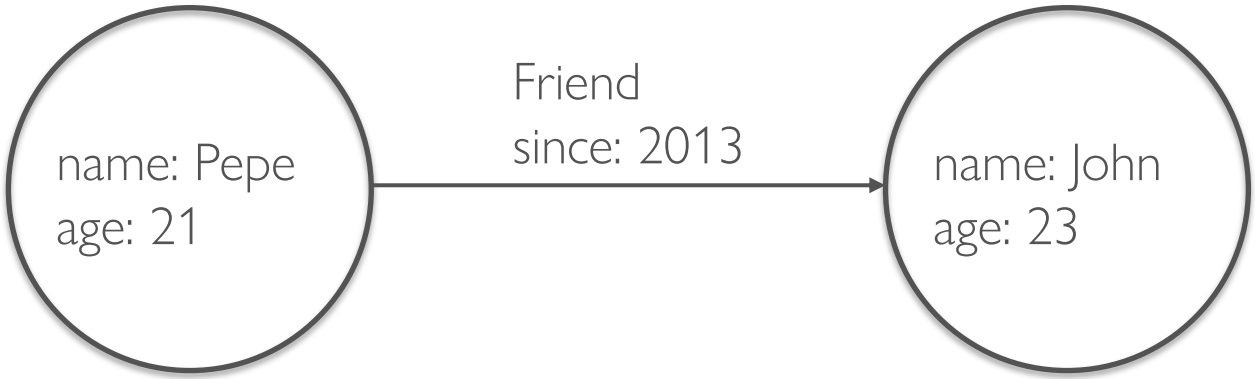
- Cypher : Query language for Neo4j

QUERIES: *Traversal* (Route)

- ***Traversal*** is the way queries are performed in Neo4j.
- A traversal is performed through the graph following the specified rules.
- **Cypher** : Query language for Neo4j
  - Cheat Sheet :
    - https://neo4j.com/ docs / cypher-cheat-sheet /5/ auradb-enterprise
- When making a tour:
  - A node can be traversed **multiple times.**
  - A relationship can only be traveled **once**

CONSULTATIONS :

- Path , is the result of a query in Neo4j.
- Those relationships that satisfy the constraints are returned along with the nodes they interconnect.
  - Pepe's friends



CYPHER: Nodes

- Nodes are represented by ()

  ()

  (identifier)

  (:label)

  (identifier: tag)

  (identifier: label { property: value })

  (identifier: tag WHERE ...)

- Example:

  (a:person {name: “Pepe”, age:21})

  (a:person

  WHERE a.name = “Pepe” AND a.age = 21})

https://neo4j.com/docs/cypher-manual/current/patterns/reference/

CYPHER: Relationships

- Relationships are represented by -->, --, <--

  -->

  -[identifier]->

  -[:tag]->

  -[identifier:label]->

  -[identifier: label { property: value }]->

  -[identifier: tag WHERE ...]->

- Example:

  -[r:friend {during:5}]->

  -[r:friend WHERE r.during =5]->

CYPHER : Pattern

- Nodes and combinations of nodes and relationships create structures called patterns.

```
( identifier: label { property: value })
-[ identifier: label { property: value }]->
( identifier: label { property: value })
```

- Example:

```
( p:person {name: “Pepe”, age:21})
-[ r:friend {since:2013}]->
( j:person {name: “Juan”, age:23})
```

CYPHER: Pattern

- A pattern can have an identifier

  identifier = (:label)-->(:label)

- Example:

  best_friend = (:person {name: Pepe,
  age:21})
  -[:friend {since:2013}]->
  (:person {name: John,
  age:23})

CYPHER: Pattern

- Patterns are also used to identify specific structures in the database.
  - Any node

```
(n) // We use n as the node identifier // so we can operate with it.
```

  - Nodes with some kind of relationship:

```
(a)-->(b)
(a)--(b)-[ r:REL_TYPE ]->()
(a)-[r:TYPE_1|TYPE_2]->() // type 1 or 2
```

CYPHER: Pattern

- Patterns are also used to identify specific structures in the database.

  - Paths of defined length:

    (a)-[*2]->(b) // length 2

    (a)-[*2..5]->(b) // length 2 to 5

    (a)-[*]->(c) // any length

Variant	Description
*	1 or more iterations .
*n	Exactly n iterations .
*m..n	Between myn iterations .
*m..	more or less iterations .
*..n	Between 1 and n iterations .

CYPHER: Pattern

- Patterns are also used to identify specific structures in the database.
  - Pattern repeats:

    (a)-[r]->(b){2} // 2 times the pattern

    (a)-[r]->{2}(b) // 2 times the ratio

Variant	Canonical	Description
{m,n}	{m,n}	Between myn iterations .
+	{1,}	1 or more iterations .
*	{0,}	0 or more iterations .
{n}	{n,n}	Exactly n iterations .
{m,}	{m,}	m or more iterations .
{,n}	{0,n}	Between 0 and n iterations .
{,}	{0,}	0 or more iterations .

CYPHER: Queries

- query
  - Identifier:
    - Node
    - Relationship
    - property

  RETURN identifier[.property]

  - All elements

  RETURN *

  - Unique results

  RETURN DISTINCT identifier[.property]

  - Evaluate expressions

  RETURN expression // eg . over 30

CYPHER: Writing Queries

- CREATE allows you to create new patterns.

  CREATE pattern

  [RETURN identifier]

- Examples:

  CREATE (p:person {name:"Pepe",age:21})

  RETURN p

  CREATE (:person {name: "Pepe", age:21})

  -[:friend {since:2013}]->

  (:person {name: "Juan", age:23})

CYPHER: Reading Queries

- MATCH allows you to perform queries that match the specified pattern.

  MATCH pattern

- Examples:

```
MATCH (n) RETURN n
//All nodes
MATCH ( p:Person ) RETURN p
//All nodes of type person
MATCH ( p:Person {age: 21}) RETURN p.age
// Age of person type nodes of 21 years
MATCH (p)-[ r:friend ]->(s) RETURN r.from
// Year since with are friends any node
MATCH (p)-[*]-() //Any length
```

CYPHER: Reading Queries

- OPTIPONAL MATCH allows you to include additional patterns in your queries. If the pattern is not matched , null is returned instead of nothing.

```
MATCH pattern
OPTIONAL MATCH pattern
```

- Examples:

```
MATCH ( n:person )
OPTIONAL MATCH (n)-[r*1]->()
RETURN r
```

CYPHER: Reading Queries

- WHERE allows you to include restrictions in the query for the specified pattern.

  ```
  WHERE restrictions
  ```

- Example:

  ```
  MATCH (p) //Filter by tag
  WHERE p:Person

  MATCH (p) //Filter by property
  WHERE EXISTS( p.edad ) AND // Has property
  ( p.age < 30 AND p.age > 20) AND
      p.name =~ "Pe.*" // Regular expression
  ```

  – It is possible to use regular expressions to filter properties

CYPHER: Reading Queries

- WHERE allows you to include restrictions in the query for the specified pattern.

  WHERE restrictions

- Example:

```
MATCH (p) //Filter by pattern
WHERE p.name IN [Pepe, Pedro] AND //Collection
(p)-->({ name:John }) //Pattern

MATCH ()–[r]->() //Filter by pattern
WHERE type (r)=~"A.*" //Type relationships
//starting with A
```

CYPHER: Writing Queries

- MATCH CREATE allows you to create new patterns from existing patterns.

```
MATCH pattern
[WHERE restrictions]
CREATE pattern
[RETURN identifier]
```

- Examples

```
MATCH ( a:person ), ( b:person )
WHERE a.name = "Pepe" AND b.name = "Juan"
CREATE (a)-[:friend {since:2013} ]->(b)
```

CYPHER: Writing Queries

- MERGE ensures that a pattern exists, if not, it creates one.

```
MERGE pattern
RETURN identifier
```

- Examples

```
CREATE (:person {name: "Pepe"})

// Create a new node
MERGE ( a:person {name: "Pepe", age: "21"})
RETURN to
// Returns the two people named Pepe
MERGE ( b:person {name: "Pepe"})
RETURN b
```

CYPHER: Writing Queries

- MERGE ensures that a pattern exists, if not, it creates one.

```
MERGE pattern
[ON CREATE SET properties]
[ON MATCH SET properties]
RETURN identifier
```

- ON CREATE allows you to assign properties if the pattern is created
- ON MATCH allows to assign properties if the pattern exists

- Example

```
MERGE ( a:person {name: "Pepe", age: "21"})
ON CREATE SET a.created = timestamp (),
    a.new = True
RETURN to
```

CYPHER: Writing Queries

- MATCH MERGE starts from existing patterns, ensures that a pattern exists that includes it/them, if not, it creates it/them.

  MATCH pattern

  MERGE pattern

  RETURN identifier

- Examples

  MATCH (a:person), (b:person)

  WHERE a.name = "Pepe" AND b.name = "Juan"

  MERGE (a)-[:friend]-(b) //We can not specify the direction of the relationship, so it searches in any type of relationship, if it does not exist it creates any one.

CYPHER: Reading Queries

- ORDER BY specifies the order in which the results should be displayed.

```
ORDER BY identifier.property
```

- Example:

```
MATCH (p)
RETURN p
ORDER BY p.age
```

```
MATCH (p)
RETURN p
ORDER BY p.age , p.name DESC
```

CYPHER: Reading Queries

- LIMIT limits the number of items to display

```
LIMIT number_elements
```

- Example:

```
MATCH (p)
RETURN p
LIMIT 2 // Display first 2 elements
```

CYPHER: Reading Queries

- SKIP specifies from which row to start displaying the result

```
SKIP number_items
```

- Example:

```
MATCH (p)
RETURN p
SKIP 2 // Skip first 2 elements

MATCH (p)
RETURN p
SKIP 1
LIMIT 2 // Display second element
```

CYPHER: Read Queries, *Aggregation*

- COUNT allows you to count the number of elements in the result of a query.
  - Number of rows

    count (*)
  - Number of non-null elements with the specified identifier

    count (identifier)

- Example:

```
MATCH (p)-[r]->(s)
RETURN count (*)
```

CYPHER: Read Queries, *Aggregation*

- SUM adds all values with the specified identifier. NULL values are ignored.

sum(identifier)

- AVG returns the average of all values with the specified identifier. NULL values are ignored.

  avg (identifier)

- MAX returns the maximum of all values with the specified identifier.

  max (identifier)

- MIN returns the minimum of all values with the specified identifier.

  min(identifier)

CYPHER: Read Queries, *Aggregation*

- COLLECT allows you to create a collection with all the elements resulting from a query. NULL values are ignored.

```
collect (identifier)
```

- Example:

```
MATCH ( p:Person )
RETURN collect ( p.age )
```

- DISTINCT removes duplicate values for a given identifier from the result of a query

```
DISTINCT identifier
```

- Example:

```
MATCH ( p:Person )
RETURN collect (DISTINCT p.age )
```

CYPHER: Read Queries, *Aggregation*

- *Grouping* keys *keys*)
  - Allows adding by grouping key
    - Aggregation for each K element

```
K, AGGREGATION_OPERATOR(ARG)
```

- Example: relationship types and number of relationships for each type

```
MATCH (p)-[r]->(s)
RETURN type (r), count (*)
```

CYPHER: Reading Queries

- WITH allows you to specify which elements and how they will be passed to the next part of the query (pipeline)

```
WITH identifiers [AS aliases]
WITH function AS alias
```

- Example:

```
MATCH ( p:person )
WITH collect (p) as persons
RETURN people
```

CYPHER: Reading Queries

- UNWIND expands a collection to a set of rows

```
UNWIND collection AS iterator
```

- Example:

```
UNWIND [1,2,3,4] AS number
RETURN number

UNWIND [1,1,2,3,4,4] AS number
WITH DISTINCT number
RETURN collect (number) AS set
```

CYPHER: Reading Queries

- UNION combines results from multiple queries
  - Combine all results

  UNION ALL
  - Combine the results by removing duplicates

  UNION

- Example:

```
MATCH ( n:person )
RETURN n.name AS name
UNION
MATCH ( n:animal )
RETURN n.name AS name
```

CYPHER: Read Queries, Functions

- PREDICATES:
  - ALL: Checks whether all elements of a collection satisfy a predicate.
    all (identifier IN collection WHERE predicate)
  - ANY: Checks whether any of the elements in a collection satisfy a predicate.
    any (identifier IN collection WHERE predicate)
  - NONE: Checks whether none of the elements in a collection satisfy a predicate.
    none (identifier IN collection WHERE predicate)
  - SINGLE: Checks whether an element of a collection satisfies a predicate.
    single(identifier IN collection WHERE predicate)
  - EXISTS: Checks if the pattern or identifier exists.
    exists (pattern or property)

- Example: All relationships in which they are adults

```
MATCH relation = ()-[ r:friend ]-()
WHERE all (n IN nodes (relation) WHERE n.age > 18)
RETURN relationship
```

CYPHER: Read Queries, Functions

- SCALAR FUNCTIONS (return a single value):

  FUNCTION_NAME(identifier)

  - **length** : Length of a *path* .
  - **size** : Length of a list or string .
  - **type** : The relationship type of the specified identifier.
  - **head, last** : Returns the first or last element of a collection.
  - **timestamp** : Returns the timestamp .
  - **toInt , toFloat , toString** : Converts a variable to integer, real or text.
  - **startNode , endNode** : Returns the start or end node of a relationship.
  - **coalesce** : Returns the first non-NULL value in a list.
  - **elementId** – ID of a node or relationship.
  - **properties** : Returns the properties of a node or relationship.

CYPHER: Read Queries, Functions

- COLLECTIONS (return collections):

  FUNCTION_NAME(identifier)

  - **nodes** : Returns all nodes in a path.
  - **relationships** : Returns all relationships for a path.
  - **labels** : List of all labels for a node.
  - **keys** : List of all the keys of the properties of a node or relation.
  - **reduce** : Returns the accumulated result of applying an expression to all elements in a collection.
  - **tail** : Returns all elements of a collection except the first.
  - **range** : Returns all elements in a collection within a specified range and with a specified step.

CYPHER: Read Queries, Functions

- COMPLETE DOCUMENTATION
  - https://neo4j.com/docs/cypher-manual/current/functions/

- MATH:
  - https://neo4j.com/docs/cypher-manual/current/functions/mathematical-numeric/

- CHAINS
  - https://neo4j.com/docs/cypher-manual/current/functions/string/

CYPHER: Update queries

- SET allows you to update node labels and properties of nodes and relationships.

  MATCH pattern

  SET label | property

  [RETURN identifier]

- Examples

```
MATCH ( a:person {name: "Pepe"})
SET a : Extraterrestrial:Mars //New tag

MATCH ( a:person {name: "Pepe"})
SET a.name = John, a.age = 29 //New value

MATCH ( a:person {name: "Pepe"})
SET a.name = NULL // Delete property
```

CYPHER: Update queries

- SET can be used to copy all properties from one node to another. Properties of the receiving node will be cleared first.

```
MATCH patterns
SET identifier = properties (identifier)
[RETURN identifier]
```

- Examples

```
MATCH ( a:person {name: "Pepe"}),
( b:person {name: "John"})
SET a = properties (b)
```

CYPHER: Update queries

- SET can also be used maps

- Examples

```
MATCH ( a:person {name: "Pepe"})
SET a += {height: 1.80, weight: 75}
//Add properties

map = {height: 1.80, weight: 75}

MATCH ( a:person {name: "Pepe"})
SET a = $ map //Replace properties

MATCH ( a:person {name: "Pepe"})
SET a = {} //Clear all properties
```

CYPHER: Update queries

- FOREACH allows you to update data in data collections, paths, and aggregate results.

```
MATCH pattern
FOREACH ( iterator | operator )
```

- Examples

```
MATCH c =(:person)-[*]-(:person)
FOREACH (n IN nodes (c)|SET n:adult )
```

CYPHER: Delete Queries

- DELETE allows you to delete nodes and relationships.

```
MATCH pattern
DELETE identifier
```

- Examples

```
MATCH ( a:person {name: "Pepe"})
DELETE a
```

CYPHER: Delete Queries

- REMOVE allows you to remove tags and properties.

```
MATCH pattern
REMOVE id:label | id.property
```

- Examples

```
MATCH ( a:person {name: "Pepe"})
REMOVE a:person , a.name
```

SCHEMA: Indexes

- CREATE INDEX creates an index for a property of a given tag or type.
  - The type is not necessary if it is a classic index (Range)
    - Types: RANGE, TEXT, POINT

```
CREATE [TYPE] INDEX name [IF NOT EXISTS]
FOR label|type
ON property
```

- Examples:

```
CREATE INDEX person_name IF NOT EXISTS
FOR ( p:person ) ON p.name

SHOW INDEXES
```

SCHEMA: Indexes

- DROP INDEX removes the index of a property for a given tag.

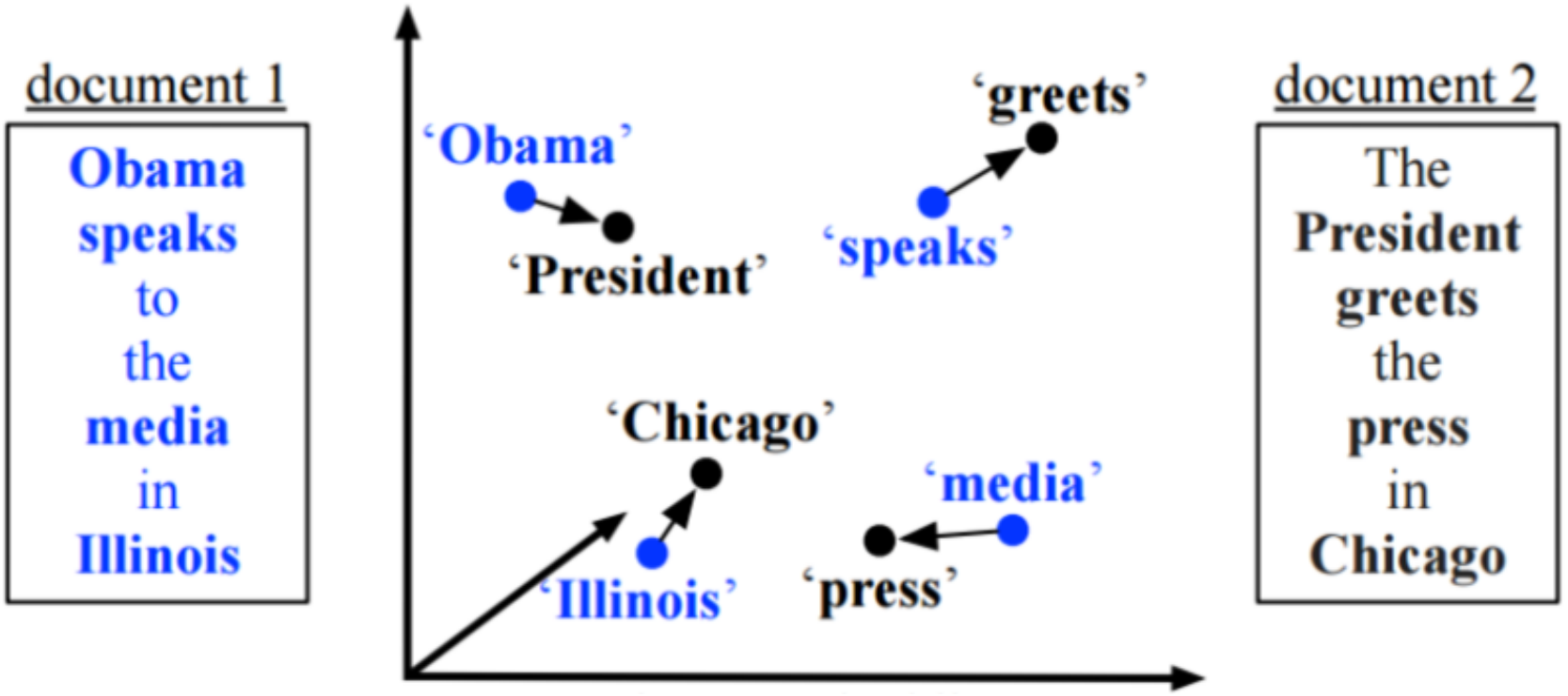
```
DROP INDEX index_name
```

- Examples

```
DROP INDEX person_name
```

SCHEMA: Indexes

- Ability to create vector indexes (*embeddings*)



SCHEMA: Restrictions

- CREATE CONSTRAINT creates a constraint of the specified type on the specified property. The constraint will be enforced at the tag level. If a unique type constraint is created, an index on that property will also be created.

```
CREATE CONSTRAINT name [IF NOT EXISTS]
FOR (label)
Requires restriction
```

- Examples

```
CREATE CONSTRAINT unique_dni IF NOT EXISTS
FOR ( p:person )
REQUIRE p.dni IS UNIQUE
```

SCHEMA: Restrictions

- DROP CONSTRAINT removes a constraint of the specified type from the specified property. If a unique constraint is dropped, the index that was created is also dropped.

```
DROP CONSTRAINT constraint_name
```

- Examples

```
DROP CONSTRAINT dni_unico
```