

A hand is shown holding several yellow puzzle pieces. Overlaid on the image is a network graph with dark grey nodes of varying sizes connected by thin lines. The background is a soft-focus image of a hand holding puzzle pieces, with a green-to-blue gradient overlay.

# Introduction to Graph Databases

v 1.0

# What is a graph?

- Nodes
- Relationships
- Properties
- Labels

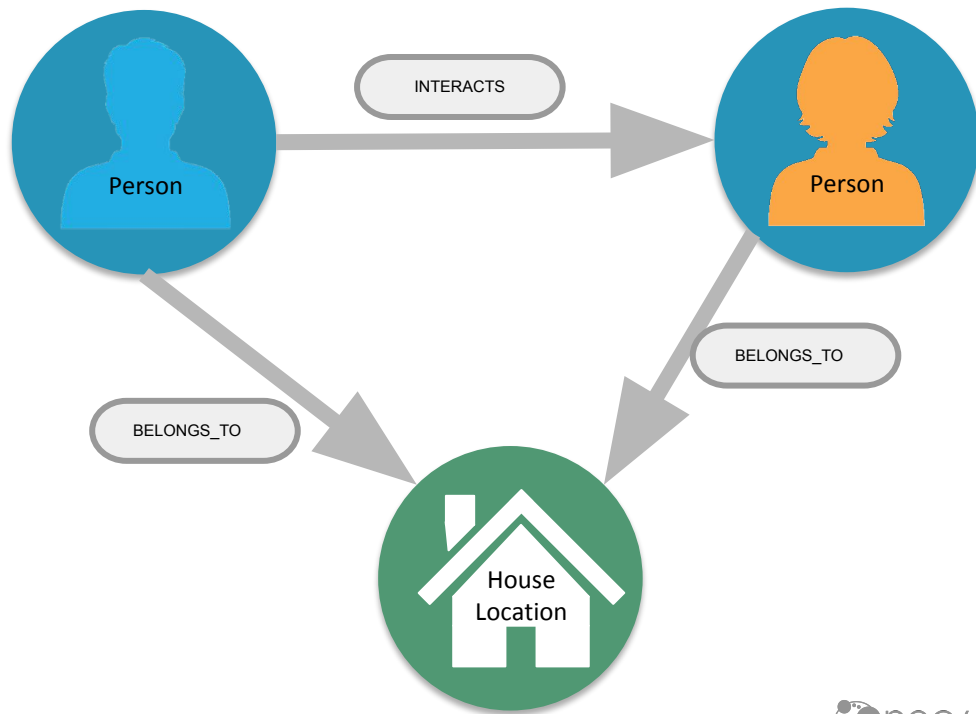
# Nodes

- Nouns in your model
- Represent the objects or entities in the graph
- Can be *labeled*:
  - Person
  - House
  - Location
  - Region



# Relationships

- Verbs in your model
- Represent the connection between nodes in the graph
- Has a type:
  - INTERACTS
  - BELONGS\_TO
- Directed relationship



# Properties

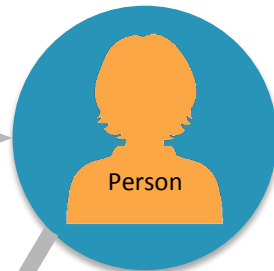
- Adjectives to describe nodes
- Adverbs to describe relationships
- Property:
  - Key/value pair
  - Can be optional or required
  - Values can be unique for nodes
  - Values have no type

name: 'Cersei Lannister'  
birth\_year: 266



INTERACTS

name: 'Dorcas'



BELONGS\_TO

since: 237

BELONGS\_TO

name: Lannister'



# Modeling relational to graph

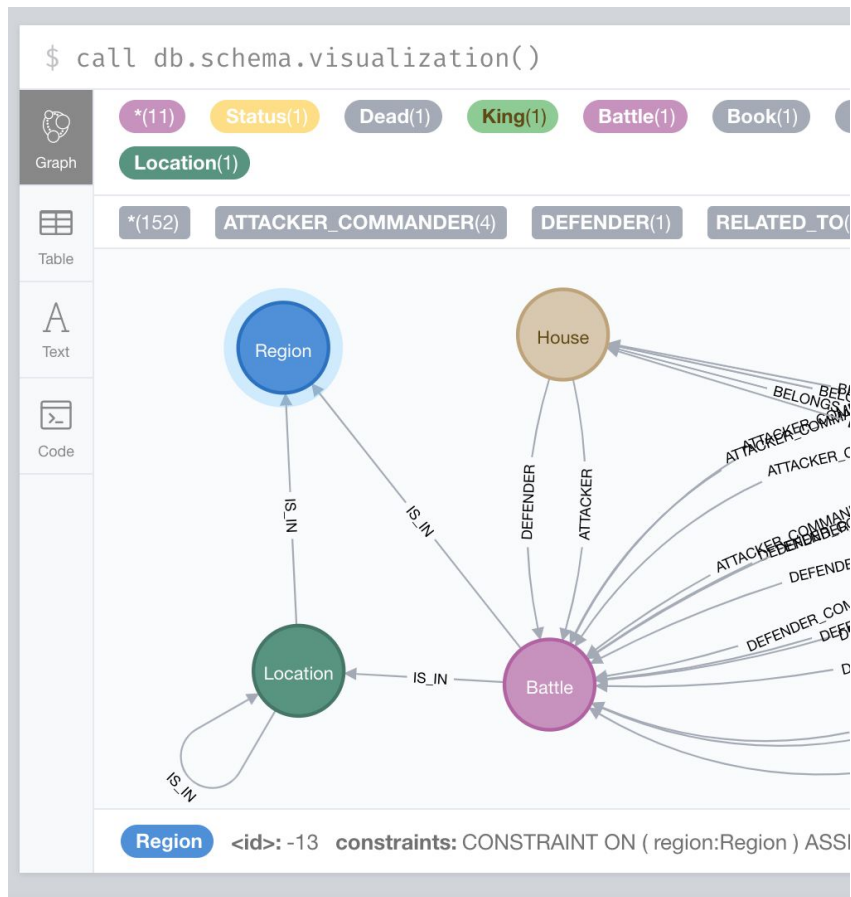
In some ways they're similar:

Relational	Graph
Rows	Nodes
Joins	Relationships
Table names	Labels
Columns	Properties

In some ways they're not:

Relational	Graph
Each column must have a field value.	Nodes with the same label aren't required to have the same set of properties.
Joins are calculated at query time.	Relationships are stored on disk when they are created.
A row can belong to one table.	A node can have many labels.

# Neo4j GOT model: CALL db.schema.visualization()

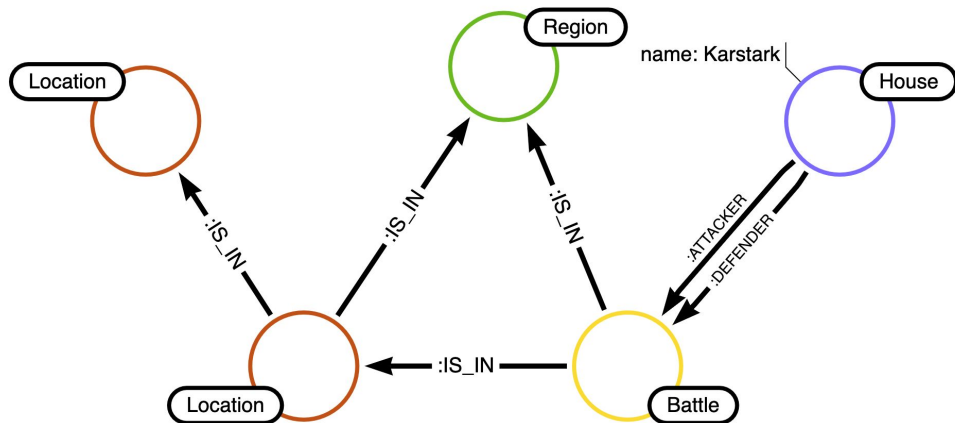


# Introduction to Cypher

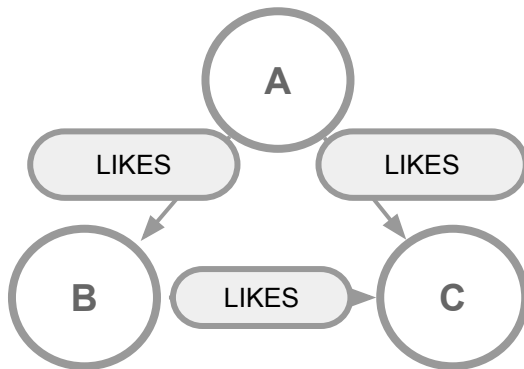


# What is Cypher?

- Declarative query language
- Focuses on **what**, not how to retrieve
- Uses keywords such as **MATCH, WHERE, CREATE**
- Runs in the database server for the graph
- ASCII art to represent nodes and relationships



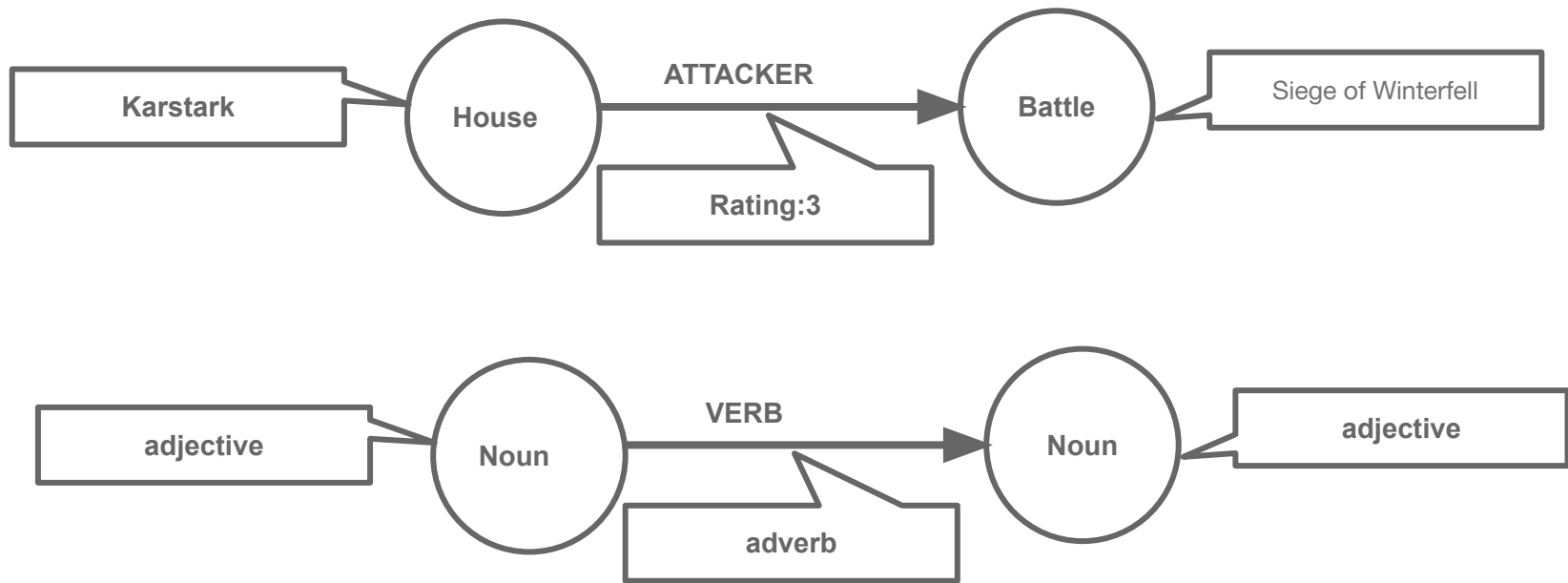
# Cypher is ASCII Art



```
(A) - [:LIKES] -> (B) , (A) - [:LIKES] -> (C) , (B) - [:LIKES] -> (C)
```

```
(A) - [:LIKES] -> (B) - [:LIKES] -> (C) <- [:LIKES] - (A)
```

# Cypher is readable



# Labels

(:Location)

(p:Region)

(:Battle)

(l:House)

## Node Labels

\*(2,640)

Battle

Book

Culture

Dead

House

King

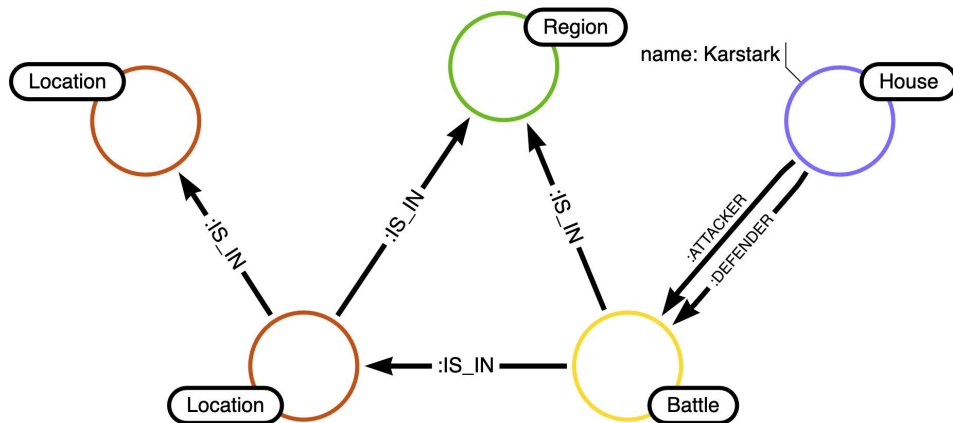
Knight

Location

Person

Region

Status



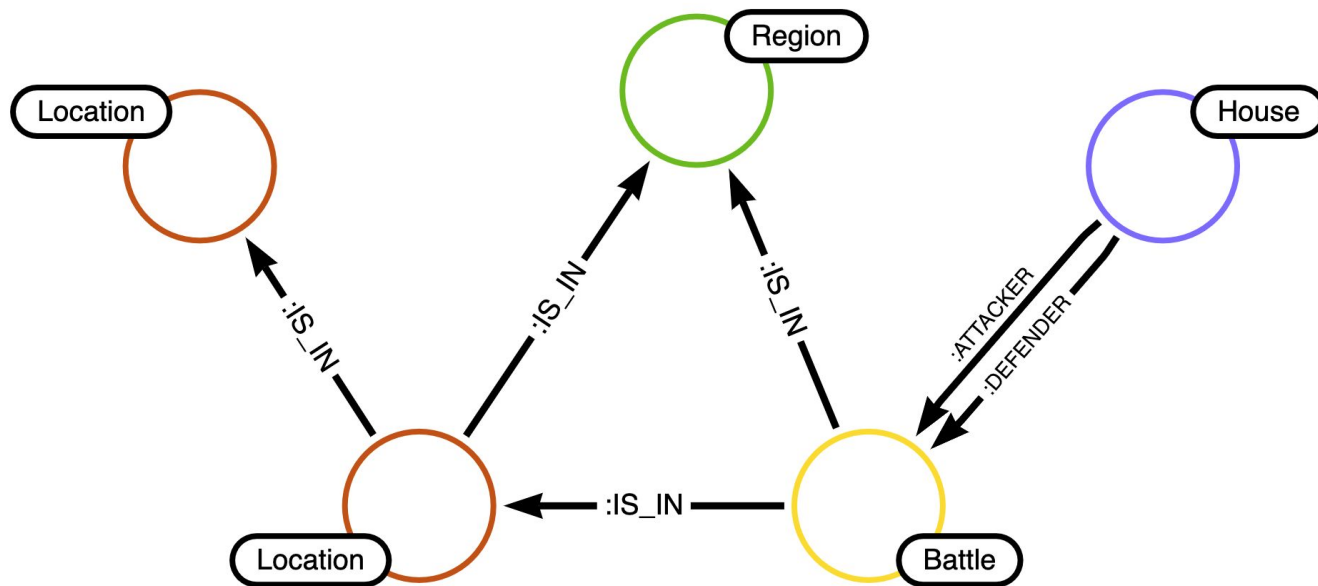
# Using MATCH to retrieve nodes

```
MATCH (n) // returns all nodes in the graph  
RETURN n
```

```
MATCH (p:House) // returns all House nodes in the graph  
RETURN p
```



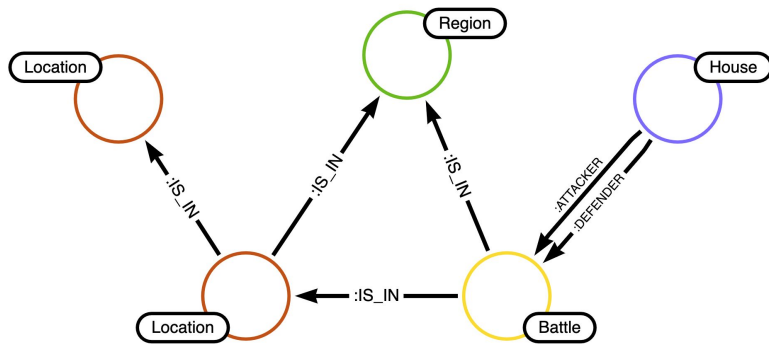
# Relationships



# ASCII art for nodes and relationships

```
()          // a node
()--()       // 2 nodes have some type of relationship
()-->()      // the first node has a relationship to the second node
()<--()      // the second node has a relationship to the first node
```

# Querying using relationships



```
MATCH (h:House) -[:ATTACKER]->(b:Battle)
RETURN h.name, b.name
```

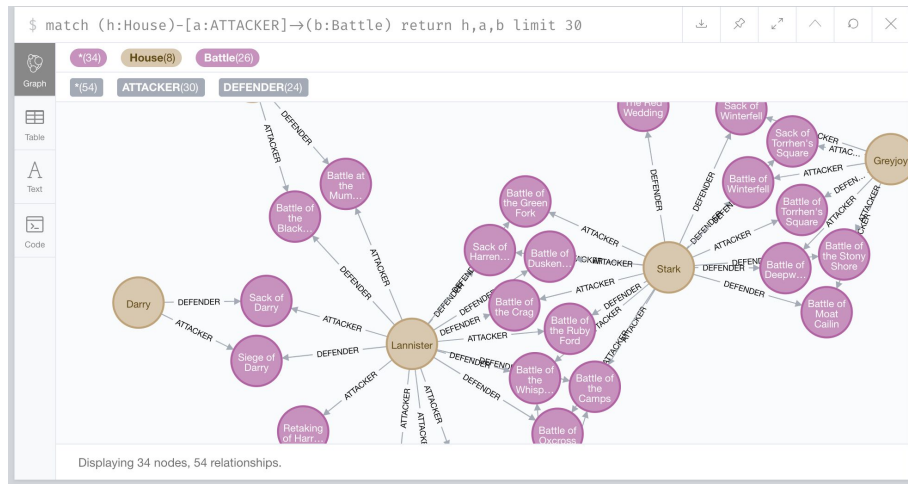
```
MATCH (h:House) --(b:Battle) // any relationship
RETURN h.name, b.name
```



## Using a relationship in a query

Find the Houses attacking in a battle returning the nodes and relationships found:

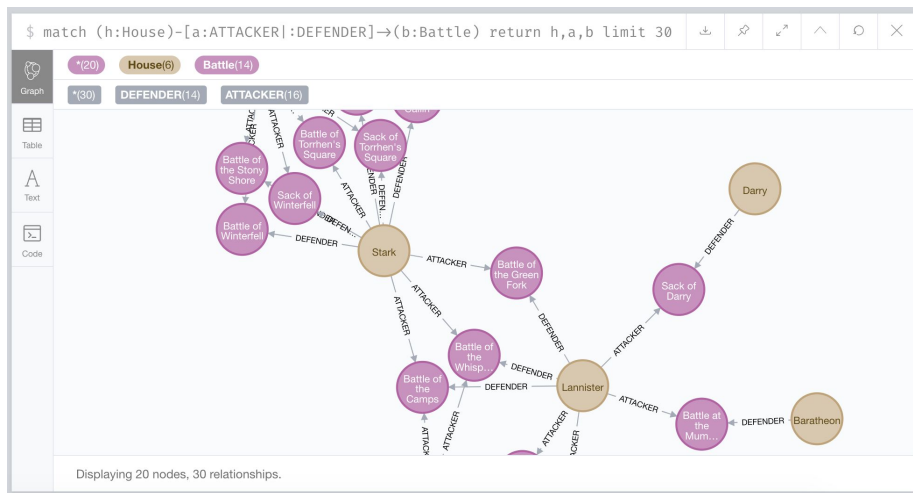
```
match (h:House)-[a:ATTACKER]->(b:Battle)
return h,a,b limit 30
```



# Querying by multiple relationships

Find the Houses attacking or defending in a battle returning the nodes and relationships found:

```
match (h:House)-[a:ATTACKER|:DEFENDER]->(b:Battle)  
return h,a,b limit 30
```



# APOC

v 1.0



# APOC

- Add-on Neo4j library
- Custom implementations of certain functionality, that can't be (easily) expressed in Cypher itself
- ~450 procedures and functions
- **CALL**ed from Cypher
  - CALL apoc.meta.stats();
  - CALL apoc.cypher.run(some cypher statement);

# GDS Cypher

v 1.0



# Important Cypher Keywords - GDS Training

- WHERE
- COUNT
- ORDER BY
- DISTINCT
- WITH
- COLLECT
- UNWIND
- SIZE
- CALL

# Filtering queries using WHERE

Previously you retrieved nodes as follows:

```
MATCH (h:House {name:'Darry'})-[r:DEFENDER]->(b:Battle)
RETURN h,b
```

A more flexible syntax for the same query is:

```
MATCH (h:House)-[r:DEFENDER]->(b:Battle)
WHERE h.name = 'Darry'
RETURN h,b
```

Testing more than equality:

```
MATCH (h:House)-[r:DEFENDER]->(b:Battle)
WHERE h.name = 'Darry' or b.name='Sack of
Winterfell'
RETURN h,b
```

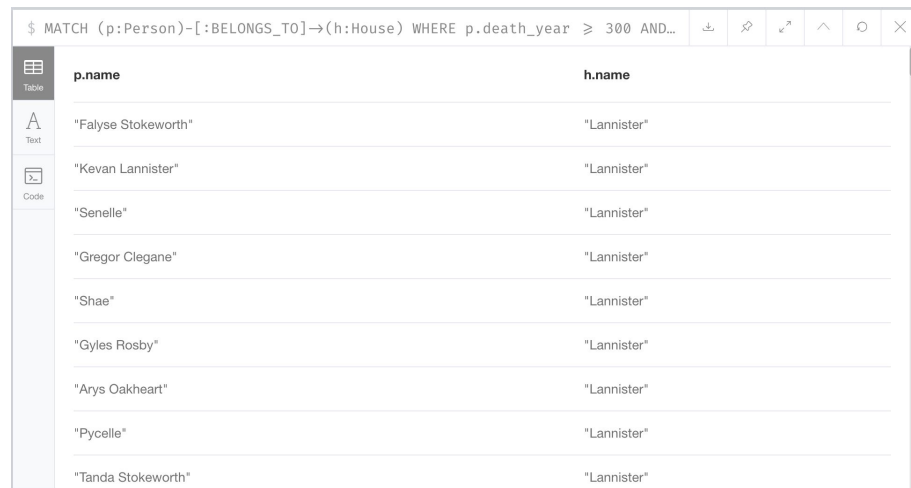
# Specifying ranges in WHERE clauses

This query to find all people who belonged to a house and died between 300 and 1200.

```
MATCH (p:Person)-[:BELONGS_TO]->(h:House)
WHERE p.death_year >= 300 AND p.death_year <= 1200
RETURN p.name, h.name
```

Is the same as:

```
MATCH (p:Person)-[:BELONGS_TO]->(h:House)
WHERE 300 <= p.death_year <= 1200
RETURN p.name, h.name
```



The screenshot shows a Neo4j Cypher query interface. The query entered is: `$ MATCH (p:Person)-[:BELONGS_TO]->(h:House) WHERE p.death_year >= 300 AND...`. The results are displayed in a table with two columns: **p.name** and **h.name**. The table contains 10 rows of data, all of which are members of the Lannister house.

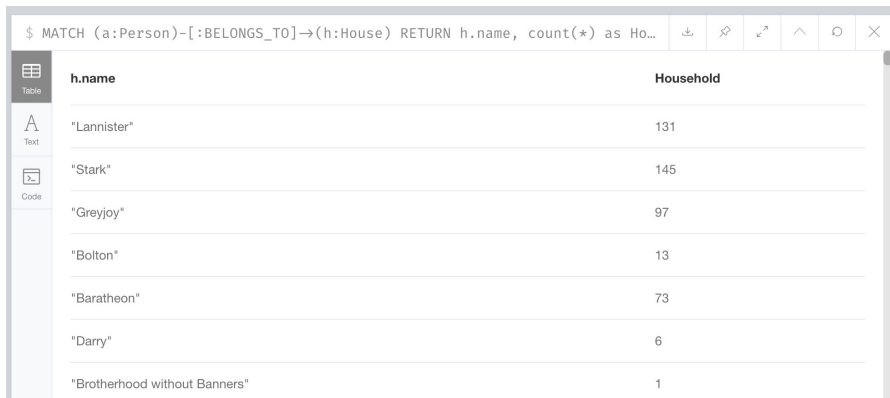
p.name	h.name
"Falyse Stokeworth"	"Lannister"
"Kevan Lannister"	"Lannister"
"Senelle"	"Lannister"
"Gregor Clegane"	"Lannister"
"Shae"	"Lannister"
"Gyles Rosby"	"Lannister"
"Arys Oakheart"	"Lannister"
"Pycelle"	"Lannister"
"Tanda Stokeworth"	"Lannister"



# Aggregation in Cypher

- Different from SQL - no need to specify a grouping key.
- As soon as you use an aggregation function, all non-aggregated result columns automatically become grouping keys.
- Implicit grouping based upon fields in the RETURN clause.

```
// implicitly groups by h.name  
MATCH (a:Person)-[:BELONGS_TO]->(h:House)  
RETURN h.name, count(*) as Household
```



h.name	Household
"Lannister"	131
"Stark"	145
"Greyjoy"	97
"Bolton"	13
"Baratheon"	73
"Darry"	6
"Brotherhood without Banners"	1

# Counting results

Find all of the attacking and defending commanders who were in a battle, return the count of the number paths found between attacker and defender and collect the battle names as a list:

MATCH

```
(attacker:Person) -[:ATTACKER_COMMANDER] -> (b:Battle) <- [d:DEFENDER_COMMANDER] -  
(defender:Person)
```

```
RETURN attacker.name, defender.name, count(b) AS commonBattles,  
       collect(b.name) AS battlenames  
order by commonBattles desc limit 5
```

attacker.name	defender.name	commonBattles	battlenames
"Stannis Baratheon"	"Randyll Tarly"	2	["Siege of Storm's End", "Battle of the Blackwater"]
"Davos Seaworth"	"Randyll Tarly"	2	["Siege of Storm's End", "Battle of the Blackwater"]
"Jaime Lannister"	"Tytos Blackwood"	2	["Battle of Riverrun", "Siege of Raventree"]
"Gregor Clegane"	"Beric Dondarrion"	1	["Battle at the Mummer's Ford"]
"Andros Brax"	"Tytos Blackwood"	1	["Battle of Riverrun"]

Started streaming 5 records after 9 ms and completed after 10 ms.

# Ordering results

You can return results in order based upon the property value:

```
MATCH
```

```
(attacker:Person)-[:ATTACKER_COMMANDER]->(b:Battle)<-[:DEFENDER_COMMANDER]-(defender:Person)
```

```
RETURN attacker.name, defender.name, count(b) AS commonBattles,  
       collect(b.name) AS battlenames
```

```
ORDER BY commonBattles desc limit 5
```



The screenshot shows a Neo4j query interface with the following Cypher query:

```
$ MATCH (attacker:Person)-[:ATTACKER_COMMANDER]->(b:Battle)<-[:DEFENDER_COMMANDER]-(defender:Person)  
RETURN attacker.name, defender.name, count(b) AS commonBattles,  
       collect(b.name) AS battlenames  
ORDER BY commonBattles desc limit 5
```

The results are displayed in a table with the following columns: attacker.name, defender.name, commonBattles, and battlenames. The table contains 5 rows of data, ordered by the number of common battles in descending order.

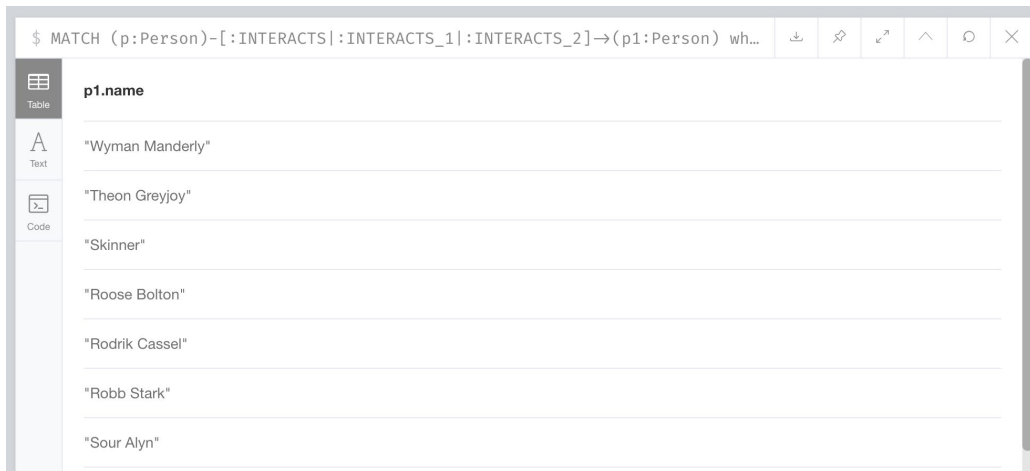
attacker.name	defender.name	commonBattles	battlenames
"Stannis Baratheon"	"Randyll Tarly"	2	["Siege of Storm's End", "Battle of the Blackwater"]
"Davos Seaworth"	"Randyll Tarly"	2	["Siege of Storm's End", "Battle of the Blackwater"]
"Jaime Lannister"	"Tytos Blackwood"	2	["Battle of Riverrun", "Siege of Raventree"]
"Gregor Clegane"	"Beric Dondarrion"	1	["Battle at the Mummer's Ford"]
"Andros Brax"	"Tytos Blackwood"	1	["Battle of Riverrun"]

Started streaming 5 records after 9 ms and completed after 10 ms.

# Eliminating duplication

We can eliminate the duplication in this query by specifying the **DISTINCT** keyword as follows:

```
MATCH (p:Person)-[:INTERACTS|:INTERACTS_1|:INTERACTS_2]->(p1:Person)
WHERE p.name = 'Ramsay Snow'
RETURN distinct p1.name
```

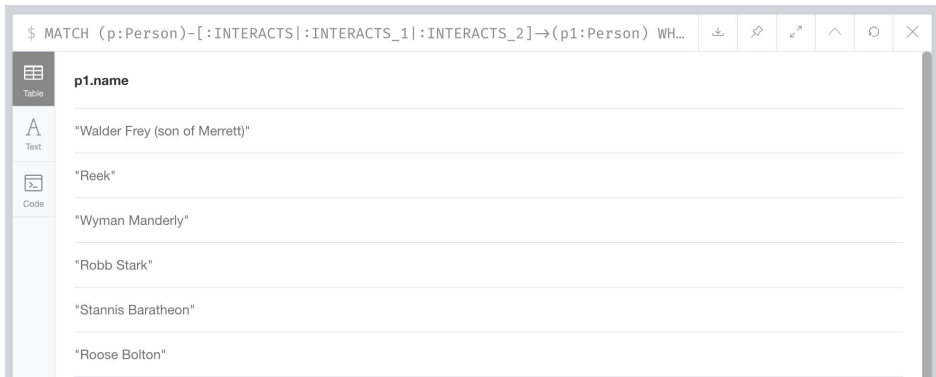


p1.name
"Wyman Manderly"
"Theon Greyjoy"
"Skinner"
"Roose Bolton"
"Rodrik Cassel"
"Robb Stark"
"Sour Alyn"

# Using WITH and DISTINCT to eliminate duplication

We can also eliminate the duplication in this query by specifying WITH DISTINCT as follows:

```
MATCH (p:Person)-[:INTERACTS|:INTERACTS_1|:INTERACTS_2]->(p1:Person)
WHERE p.name = 'Ramsay Snow'
WITH distinct p1
RETURN p1.name
```

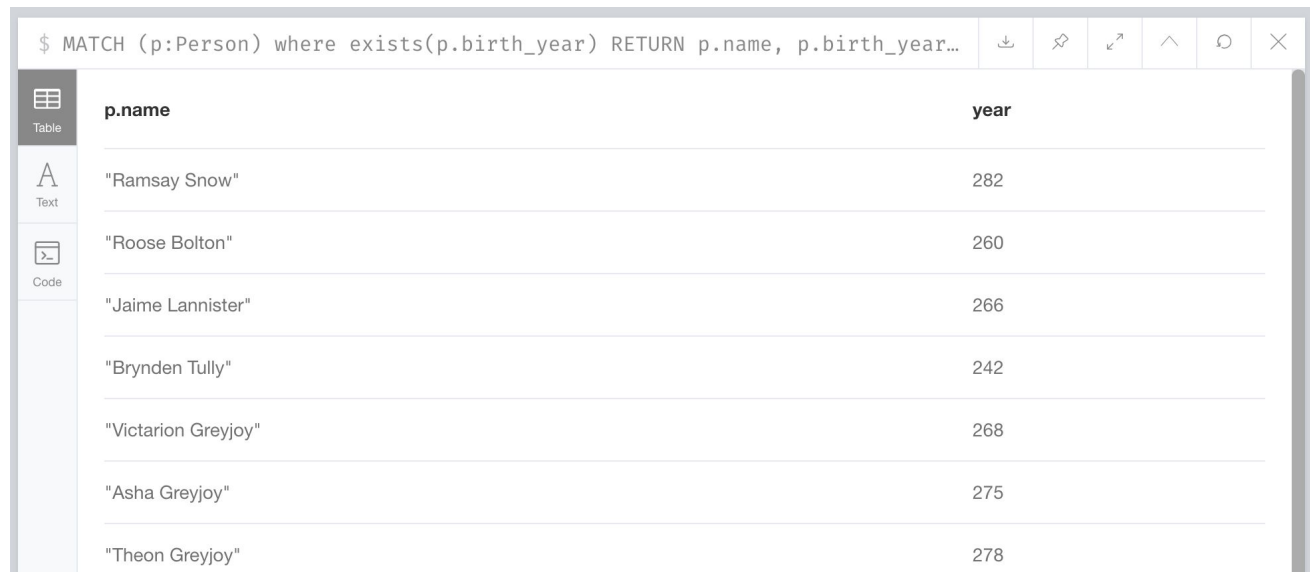


p1.name
"Walder Frey (son of Merrett)"
"*Reek*"
"Wyman Manderly"
"Robb Stark"
"Stannis Baratheon"
"Roose Bolton"

# Limiting results

What are the names of the ten youngest persons?

```
MATCH (p:Person) where exists(p.birth_year)
RETURN p.name, p.birth_year as year ORDER BY p.birth_year DESC LIMIT
10
```

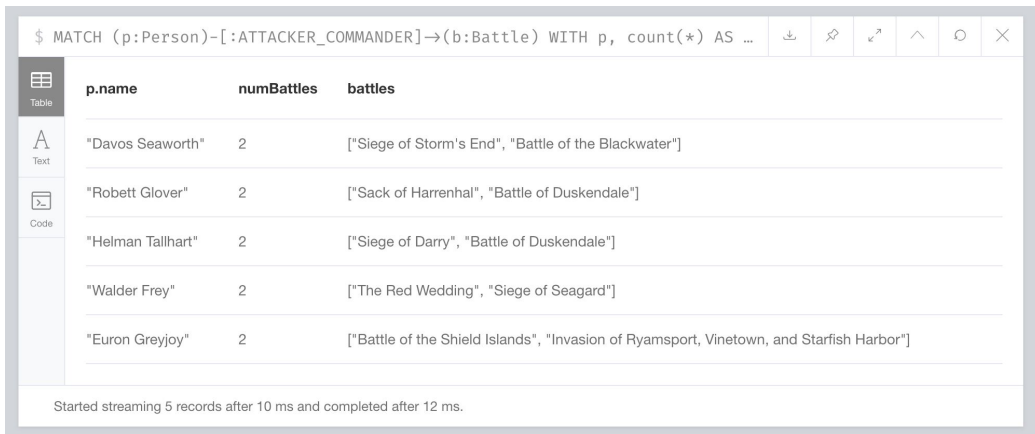


	p.name	year
	"Ramsay Snow"	282
	"Roose Bolton"	260
	"Jaime Lannister"	266
	"Brynden Tully"	242
	"Victarion Greyjoy"	268
	"Asha Greyjoy"	275
	"Theon Greyjoy"	278

# Controlling number of results using WITH

Retrieve the persons who were the ATTACKER\_COMMANDER in two battles, returning the list of battles:

```
MATCH (p:Person)-[:ATTACKER_COMMANDER]->(b:Battle)
WITH p, count(*) AS numBattles, collect(b.name) as battles
WHERE numBattles = 2
RETURN p.name, numBattles, battles
```



\$ MATCH (p:Person)-[:ATTACKER\_COMMANDER]->(b:Battle) WITH p, count(\*) AS ...

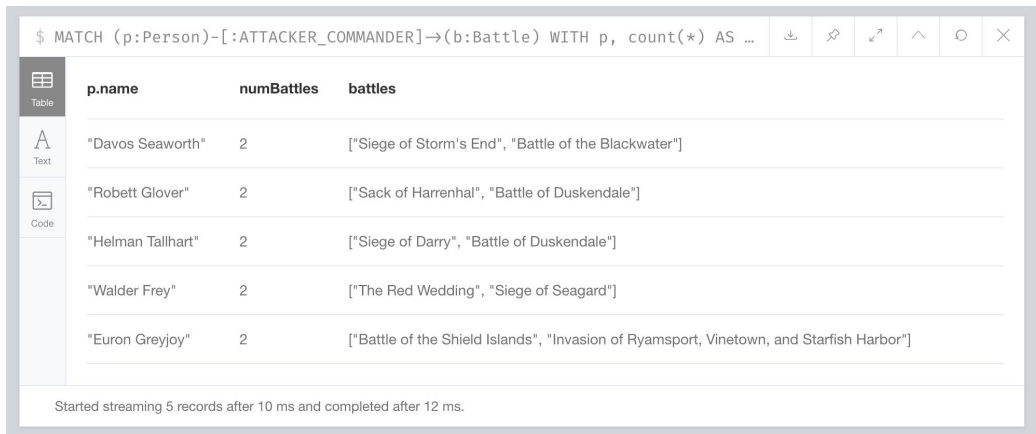
p.name	numBattles	battles
"Davos Seaworth"	2	["Siege of Storm's End", "Battle of the Blackwater"]
"Robett Glover"	2	["Sack of Harrenhal", "Battle of Duskendale"]
"Helman Tallhart"	2	["Siege of Darry", "Battle of Duskendale"]
"Walder Frey"	2	["The Red Wedding", "Siege of Seagard"]
"Euron Greyjoy"	2	["Battle of the Shield Islands", "Invasion of Ryamport, Vinetown, and Starfish Harbor"]

Started streaming 5 records after 10 ms and completed after 12 ms.

# Additional processing using WITH

Use the WITH clause to perform intermediate processing or data flow operations.

```
MATCH (p:Person)-[:ATTACKER_COMMANDER]->(b:Battle)
WITH p, count(*) AS numBattles, collect(b.name) as battles
WHERE numBattles = 2
RETURN p.name, numBattles, battles
```



The image shows a screenshot of the Neo4j Cypher query interface. At the top, the query is entered: `$ MATCH (p:Person)-[:ATTACKER_COMMANDER]->(b:Battle) WITH p, count(*) AS numBattles, collect(b.name) as battles`. Below the query bar, there are icons for Table, Text, and Code. The Table view is selected, displaying a table with three columns: **p.name**, **numBattles**, and **battles**. The table contains five rows of data. At the bottom of the interface, a status message reads: "Started streaming 5 records after 10 ms and completed after 12 ms."

p.name	numBattles	battles
"Davos Seaworth"	2	["Siege of Storm's End", "Battle of the Blackwater"]
"Robett Glover"	2	["Sack of Harrenhal", "Battle of Duskendale"]
"Helman Tallhart"	2	["Siege of Darry", "Battle of Duskendale"]
"Walder Frey"	2	["The Red Wedding", "Siege of Seagard"]
"Euron Greyjoy"	2	["Battle of the Shield Islands", "Invasion of Ryamport, Vinetown, and Starfish Harbor"]

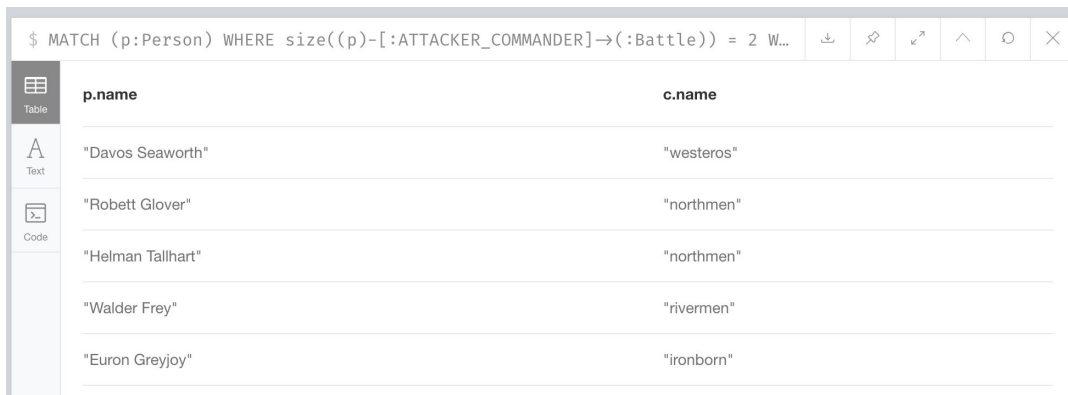
Started streaming 5 records after 10 ms and completed after 12 ms.



# Additional processing using WITH

Find all person who were an ATTACKER\_COMMANDER in 2 battles, and find (optionally) the cultures they belong to and return the person and the culture:

```
MATCH (p:Person)
WHERE size((p)-[:ATTACKER_COMMANDER]->(:Battle)) = 2
WITH p
OPTIONAL MATCH (p)-[:MEMBER_OF_CULTURE]->(c:Culture)
RETURN p.name, c.name;
```



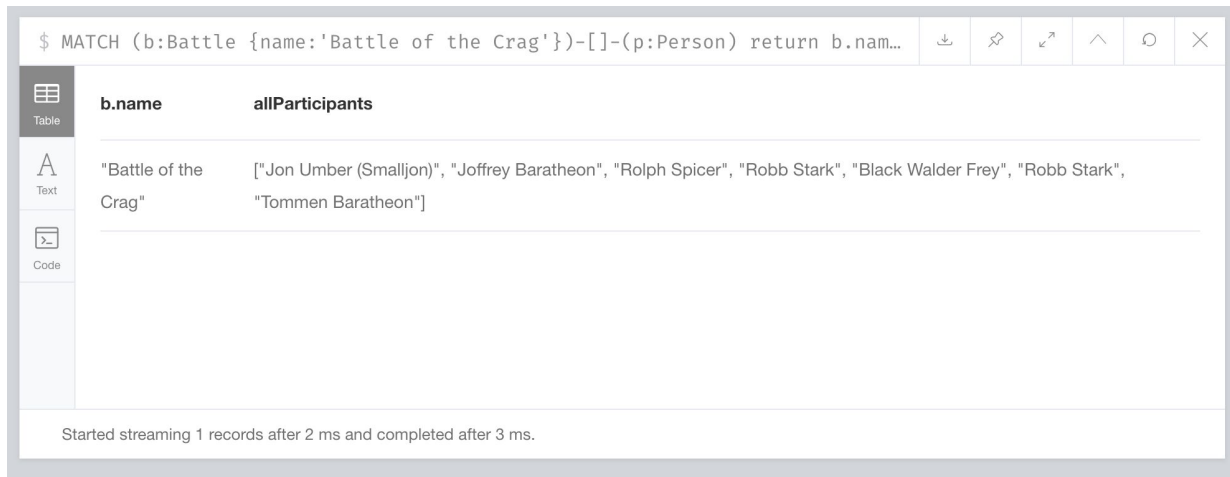
The image shows a screenshot of the Neo4j interface. At the top, a Cypher query is entered in the editor: `$ MATCH (p:Person) WHERE size((p)-[:ATTACKER_COMMANDER]->(:Battle)) = 2 W...`. Below the editor, a sidebar on the left contains icons for 'Table', 'Text', and 'Code', with 'Table' selected. The main area displays a table with two columns: 'p.name' and 'c.name'. The table contains five rows of data, representing characters from Game of Thrones and the cultures they belong to.

p.name	c.name
"Davos Seaworth"	"westeros"
"Robett Glover"	"northmen"
"Helman Tallhart"	"northmen"
"Walder Frey"	"rivermen"
"Euron Greyjoy"	"ironborn"

# Collecting results

Find the participants in the 'Battle of the Crag' and return them as a single list.

```
MATCH (b:Battle {name:'Battle of the Crag'})-[]-(p:Person)
return b.name, collect(p.name) as allParticipants
```

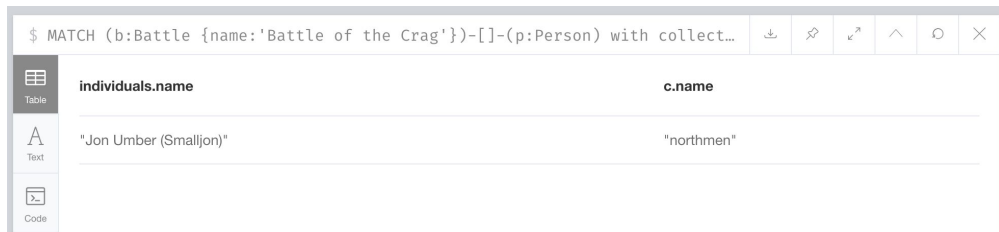


The image shows a screenshot of the Neo4j Cypher query interface. At the top, the query is entered: `$ MATCH (b:Battle {name:'Battle of the Crag'})-[]-(p:Person) return b.name, collect(p.name) as allParticipants`. Below the query bar, there are three tabs: 'Table', 'Text', and 'Code'. The 'Table' tab is selected, displaying a table with two columns: 'b.name' and 'allParticipants'. The table contains one row of data. At the bottom of the interface, a status message reads: 'Started streaming 1 records after 2 ms and completed after 3 ms.'

b.name	allParticipants
"Battle of the Crag"	["Jon Umber (Smalljon)", "Joffrey Baratheon", "Rolph Spicer", "Robb Stark", "Black Walder Frey", "Robb Stark", "Tommen Baratheon"]

# UNWIND

- Transform a collection into **rows**.
- Very useful for working with collections of properties, nodes, paths and sorting, etc.
- Allows collecting a set of nodes to avoid requerying.
- Especially useful after aggregation where you want to partition the data and perform further processing of the selected aggregated data.



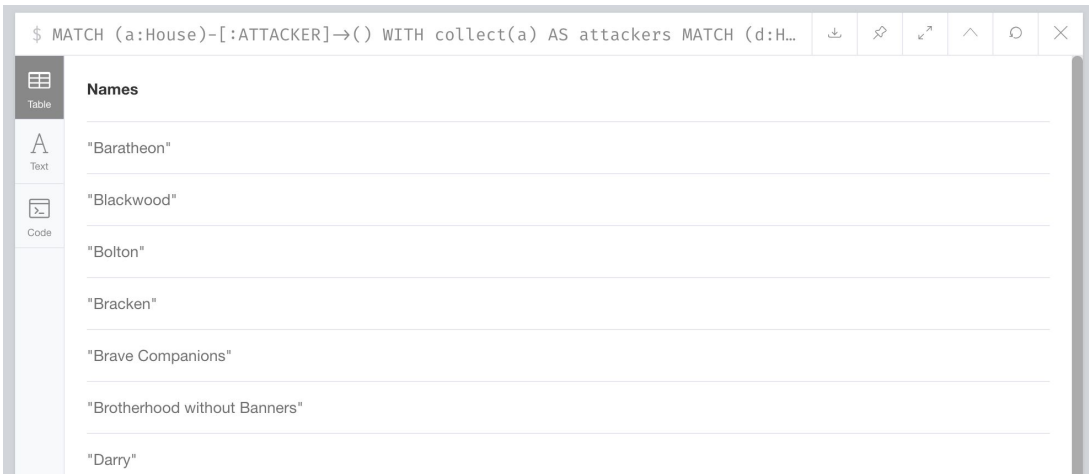
The screenshot shows the Neo4j Cypher Query Editor interface. The query entered is: `$ MATCH (b:Battle {name:'Battle of the Crag'})-[]-(p:Person) with collect...`. The results are displayed in a table view with two columns: `individuals.name` and `c.name`. The first row of data shows `"Jon Umber (Smalljon)"` and `"northmen"`. On the left sidebar, the 'Table' view is selected, with 'Text' and 'Code' options also visible.

individuals.name	c.name
"Jon Umber (Smalljon)"	"northmen"

```
MATCH (b:Battle {name:'Battle of the
Crag'})-[]-(p:Person)
with collect(p) as allParticipants
unwind allParticipants as individuals
match
(individuals)-[:MEMBER_OF_CULTURE]->(c:
Culture)
return individuals.name, c.name
```

# Another UNWIND example

```
MATCH (a:House)-[:ATTACKER]->()
WITH collect(a) AS attackers
MATCH (d:House)-[:DEFENDER]->()
WITH attackers, collect(d) AS defenders
UNWIND (attackers + defenders) AS houses
WITH DISTINCT(houses)
RETURN houses.name as Names ORDER BY Names LIMIT 10
```



The screenshot shows the Neo4j interface with a Cypher query entered in the editor. The query is: `$ MATCH (a:House)-[:ATTACKER]->() WITH collect(a) AS attackers MATCH (d:H...` . The results are displayed in a table view with the title "Names". The table contains 10 rows of house names, ordered alphabetically. The interface includes a sidebar with icons for Table, Text, and Code views, and a top toolbar with various action icons.

Names
"Baratheon"
"Blackwood"
"Bolton"
"Bracken"
"Brave Companions"
"Brotherhood without Banners"
"Darry"

# How much longer will this guy talk?



# Call and Yield - Neo4j Procedures

- Procedures are called using the [CALL](#) clause.
- Procedures have a signature and **YIELD** a result.
- **call dbms.listProcedures()** shows all available procedures

## EXAMPLE:

```
CALL gds.graph.create.estimate('Person', 'INTERACTS') YIELD  
nodeCount, relationshipCount, requiredMemory
```

# Call and Yield - Neo4j Procedures

- Procedures are called using the [CALL](#) clause.
- Procedures have a signature and **YIELD** a result.
  - Signatures for procedures are in the documentation.

- GDS Example:

`CALL gds.wcc.stream('myGraph', { relationshipWeightProperty: 'weight', threshold: 1.0 , concurrency:4})`

`YIELD nodeId, componentId`

`RETURN gds.util.asNode(nodeId).name AS Name, componentId AS ComponentId`  
`ORDER BY ComponentId, Name`

 **Signature**

 **Yield**

# Data Modeling





# Developing the initial graph data model

1. Define high-level domain requirements
2. Create sample data for modeling purposes
3. Define the questions for the domain
4. Identify entities
5. Identify connections between entities
6. Test the model against the questions
7. Test scalability

# Domain requirements

- Description of the application
- Identify stakeholders, developers
- Identify users of the application (people, systems)
- Enumerate the use cases that are agreed upon by all stakeholders where users are part of the use case

# Developing the initial graph data model

1. Define domain requirements
2. Create sample data for modeling purposes
3. Define the questions for the domain
4. Identify entities
5. Identify connections between entities
6. Test the model against the questions
7. Test scalability



# Identify the entities from the questions

- Nouns
  - Entities are the generic nouns (example: City)
  - Name the entities which will be nodes in the graph:
    - Unambiguous meaning for the name
    - Agreement by stakeholders
  - Can entities be grouped or categorized?
    - Pets, dogs, cats
- Properties for the entities
  - Property value is a proper noun for the name of the entity (example: Boston)
  - Uniquely identify an entity
  - Used to describe the entity to answer questions (anchor for the query)

# Developing the initial graph data model

1. Define domain requirements
2. Create sample data for modeling purposes
3. Define the questions for the domain
4. Identify entities
5. Identify connections between entities
6. Test the model against the questions
7. Test scalability



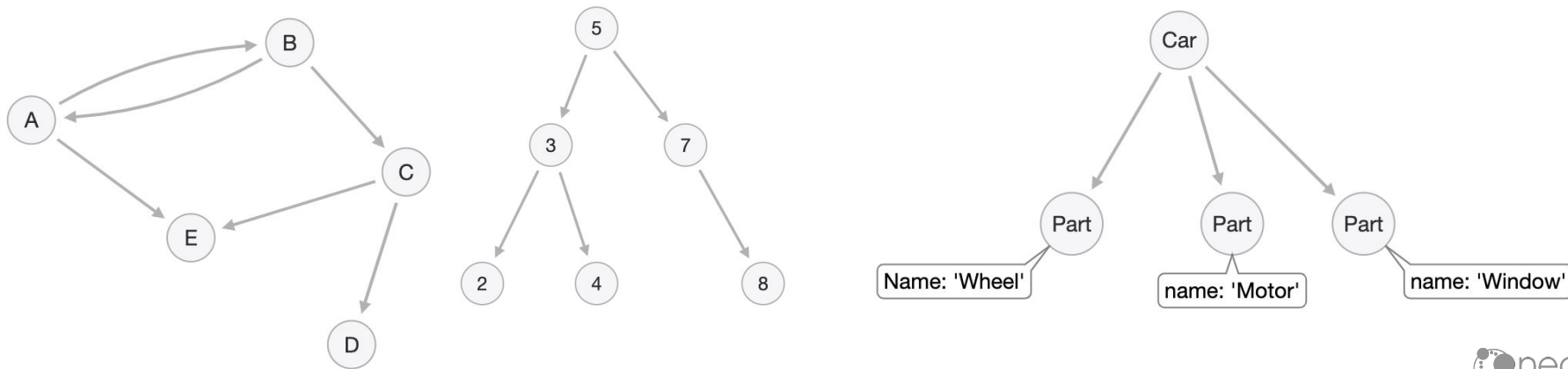
# Identify connections between entities

- The purpose of the model is to answer questions about the data.
- Connections are the verbs in the questions derived from use cases.
- Most questions are typically about the the connectedness of the data:
  - What ingredients are used in a recipe?
  - Who is married to this person?
- Exactly **one** node at the end of every relationship (connection).
- A relationship must have direction when it is created.



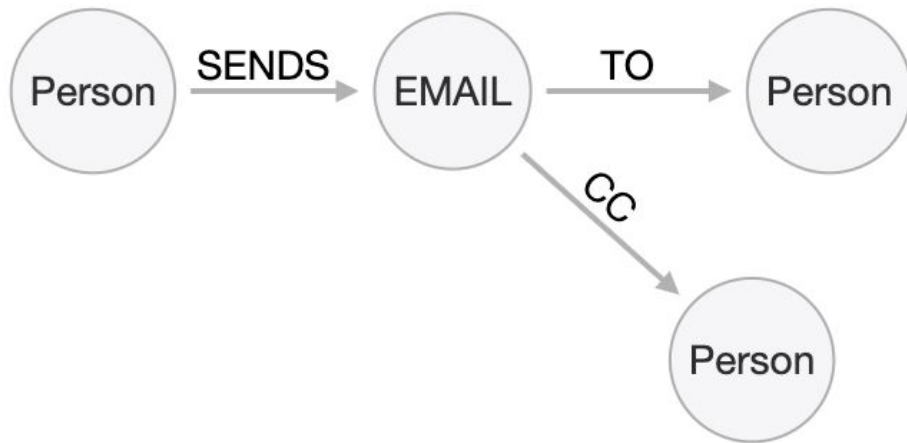
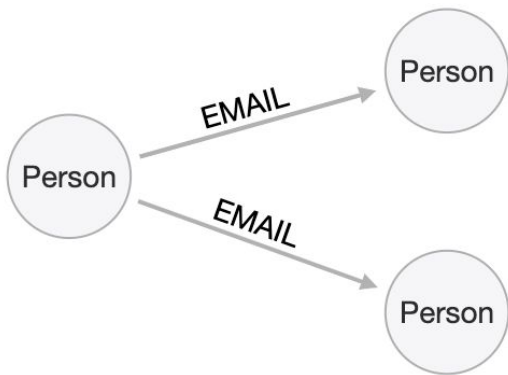
# Graph structure

- Relationships define the **structure** of the graph which is the **model**:
  - Between nodes of same type
  - Between nodes of different types
  - Determine navigational paths used at runtime (optimization)
  - Can also connect two different models in a graph
    - HR model connects to Payroll model



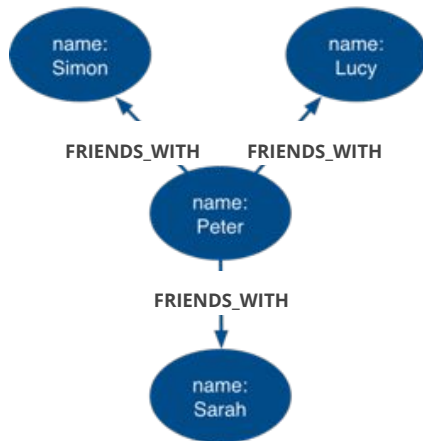
# Naming relationships

- Stakeholders must agree upon name that denotes the verbs.
- Avoid names that could be construed as nouns (for example **email**)
- Neo4j has a limit of 64K relationship types (names)

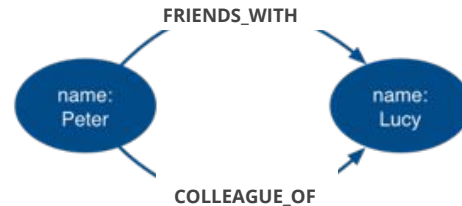




# How many relationships?



Nodes can have more than one relationship



Nodes can be connected by more than one relationship



Self relationships are allowed

# How important is direction?

Direction is required for creating the model and in particular for implementing the model (Cypher code) in the underlying Neo4j graph.



Whether direction is used for queries depends on the question:

- What are the names of the episodes of the Dr. Who series? (direction not used for the query)
- What episode follows The Ark in Space? (direction used for the query)

# Qualifying a relationship

Use properties to describe the weight or quality of the relationship.

