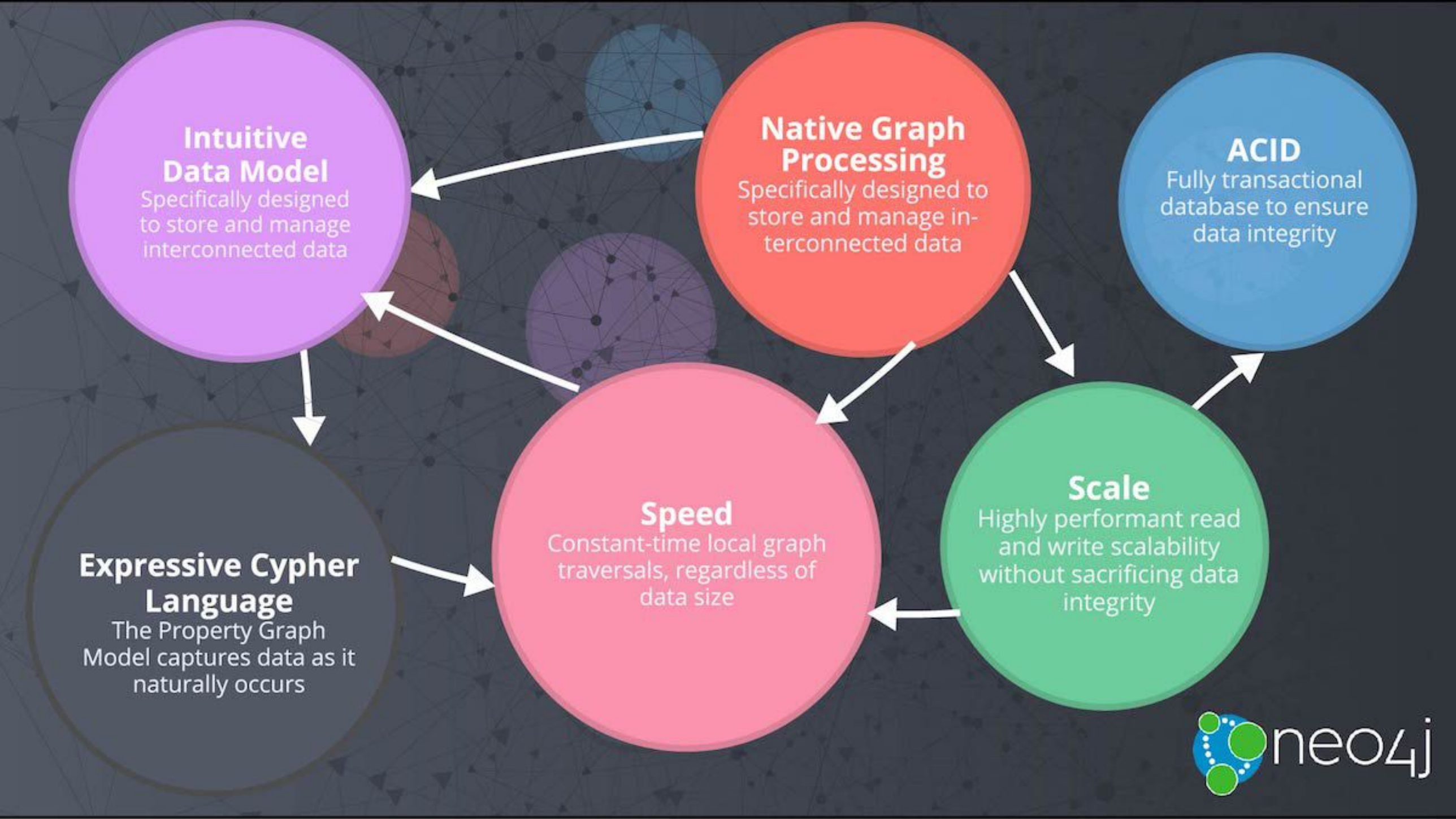
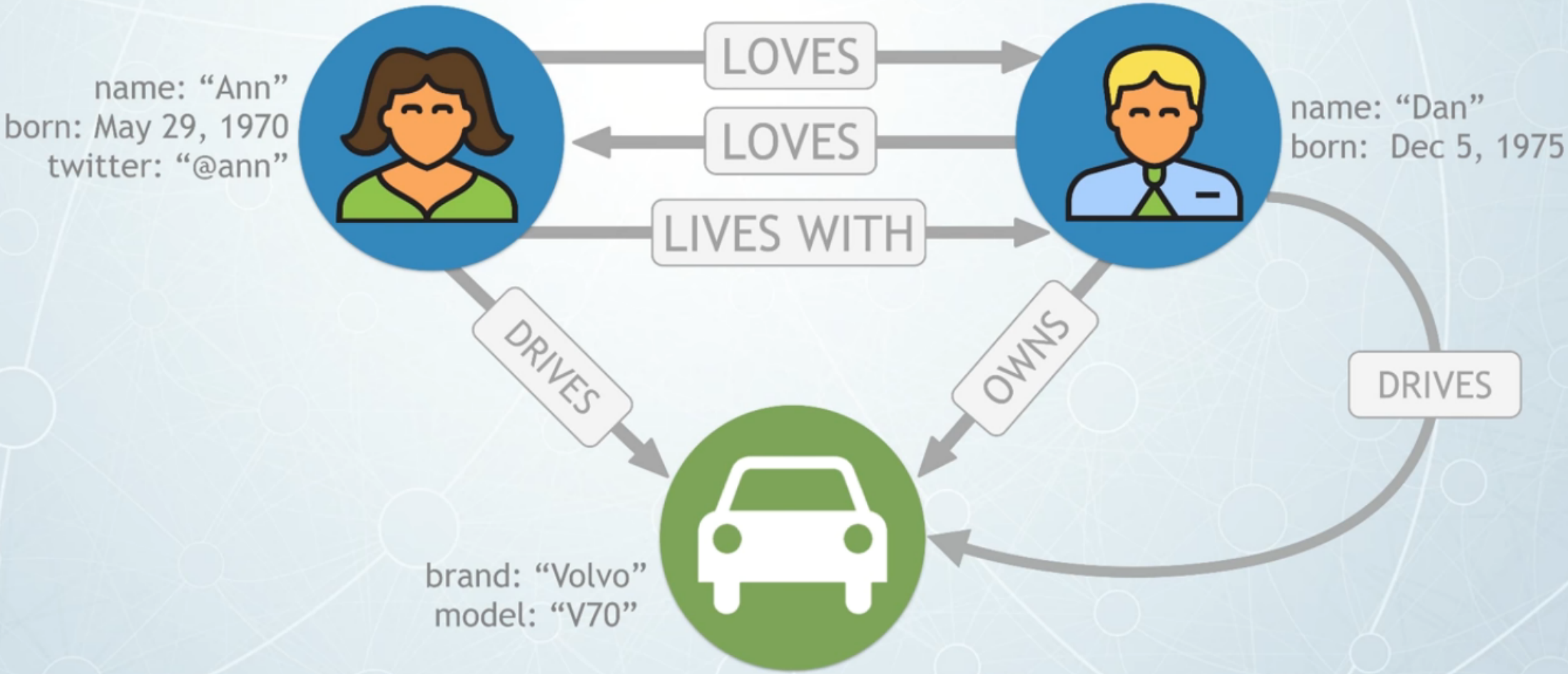
Credits

- Most of the slides/images are taken from an online course, Intro to Graph Databases Series, taught by Ryan Boyd (Neo4j Head of Developer Relations).
- Few are taken from the tutorials found on Neo4j's website.


Intuitive Data Model
Specifically designed to store and manage interconnected data
Native Graph Processing
Specifically designed to store and manage in-terconnected data
ACID
Fully transactional database to ensure data integrity
Expressive Cypher Language
The Property Graph Model captures data as it naturally occurs
Speed
Constant-time local graph traversals, regardless of data size
Scale
Highly performant read and write scalability without sacrificing data integrity
neo4j

Detailed Property Graph



Basic Create and Query

Creating the Data

NODE

NODE

```
CREATE (:Person { name:“Ann”} ) - [:LOVES]-> (:Person { name:“Dan”} )
```

LABEL PROPERTY

LABEL PROPERTY

Patterns use AsciiArt

AsciiArt for Nodes

Nodes are surrounded by parenthesis

`()` or `(p)`

Labels, or tags, start with : and group nodes by roles or types

```
(p:Person:Mammal)
```

Nodes can have properties

```
(p:Person {name: 'Veronica'})
```

AsciiArt for Relationships

Relationships are wrapped with hyphens or square brackets

`-->` or `-[h:HIRED]->`

Direction of relationship is specified with < >

`(p1)-[:HIRED]->(p2)` or `(p1)<-[:HIRED]-(p2)`

Relationships have properties too

```
-[:HIRED {type: 'fulltime'})->
```

Patterns

Patterns are drawn by connecting nodes and relationships with hyphens, optionally specifying a direction with > and < signs.

```
()-[]-()
()-[]->()
()<-[]-()
```

What were those?

Relationships are wrapped with hyphens or square brackets

```
--> or -[h:HIRED]->
```

Direction of relationship is specified with < >

```
(p1)-[:HIRED]->(p2)   or   (p1)<-[:HIRED]-(p2)
```

Relationships have properties too

```
-[:HIRED {type: 'fulltime'})->
```

Variables

Used to refer to Relationships

```
-[h:HIRED]->
```

Used to refer to Nodes

```
(p1)-[:HIRED]->(p2)
```

or

```
(p1)<-[:HIRED]-(p2)
```

How do I find Ann's Car?

```
MATCH
  (:Person {name: 'Ann'})-[:DRIVES]->(c:Car)
RETURN
  c
```

Is there another way?

```
MATCH
  (a:Person)-[:DRIVES]->(c:Car)
WHERE
  a.name='Ann'
RETURN
  c
```

Whom does Ann Love?


```
MATCH
  (:Person {name:“Ann”})-[:LOVES]->(op:Person)
RETURN
  op
```

Whom does Ann Love?


```
MATCH
  (:Person {name:"Ann"})-[:LOVES]->(op:Person)
RETURN
  op
```

It's Dan, of course!

Who drives a car owned by a lover?

```
MATCH
  (p1:Person)-[:DRIVES]->(c:Car)-[:OWNED_BY]->(p2:Person)<-[:LOVES]-(p1)
RETURN
  p1
```

Components of a Cypher Query

```
MATCH path = (:Person)-[:ACTED_IN]->(:Movie)
RETURN path
```

`MATCH` and `RETURN` are Cypher keywords

`path` is a variable

`:Movie` is a node label

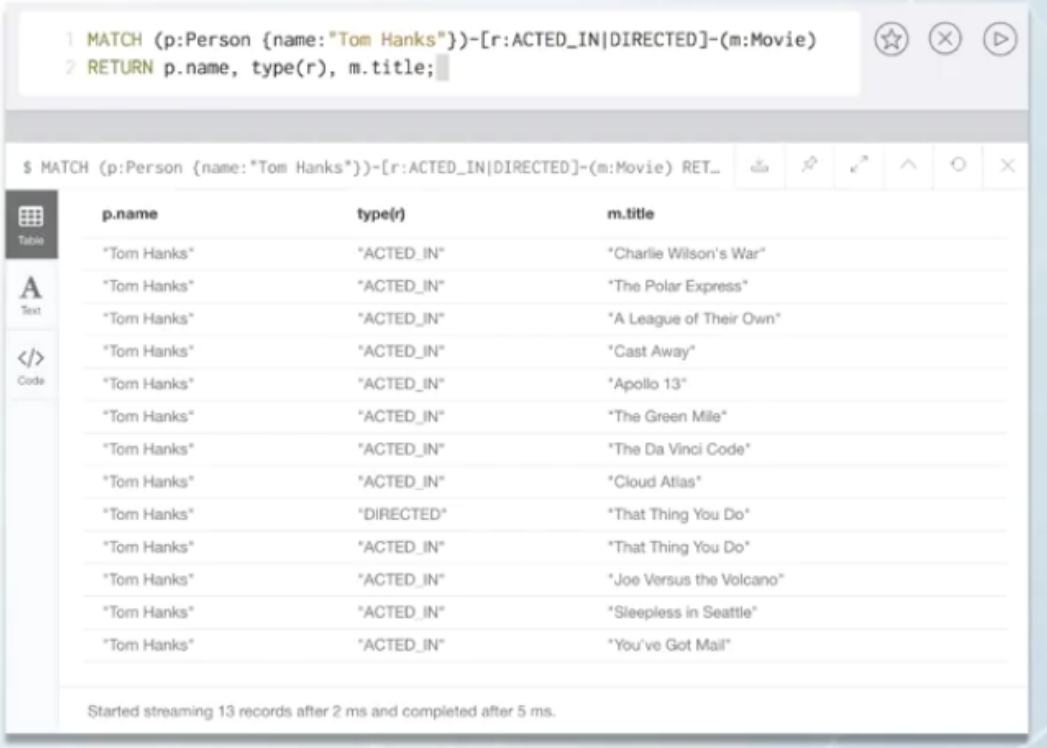
`:ACTED_IN` is a relationship

More on Syntax

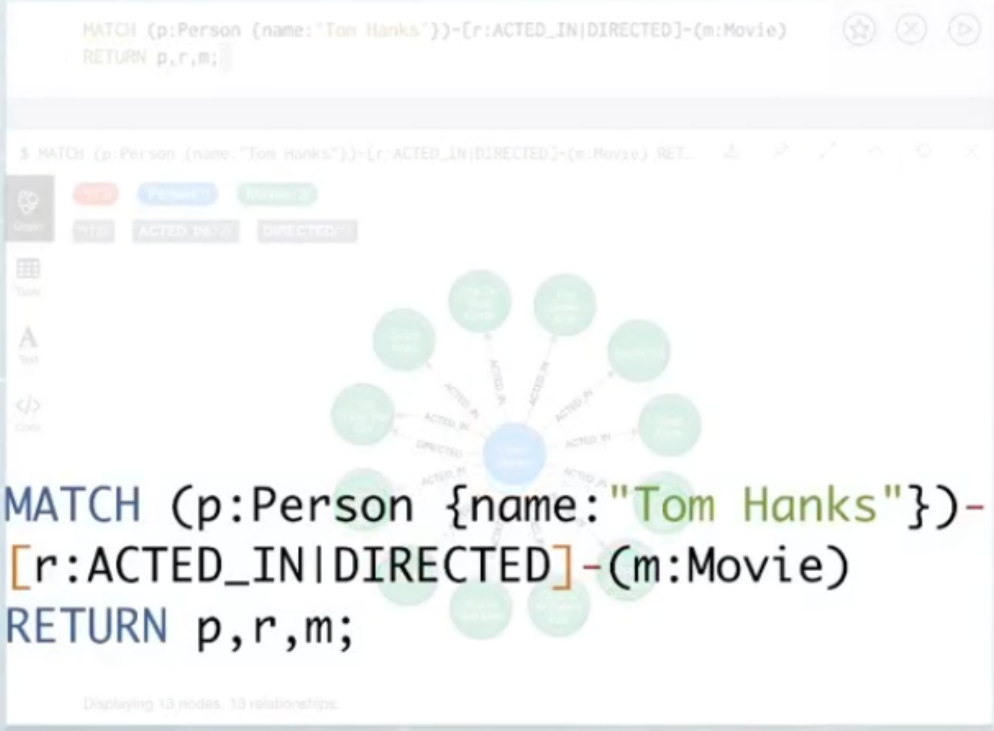
Graph vs Tabular Results

```
1 MATCH (p:Person {name:"Tom Hanks"})-[r:ACTED_IN|DIRECTED]-(m:Movie)
2 RETURN p,r,m;
```

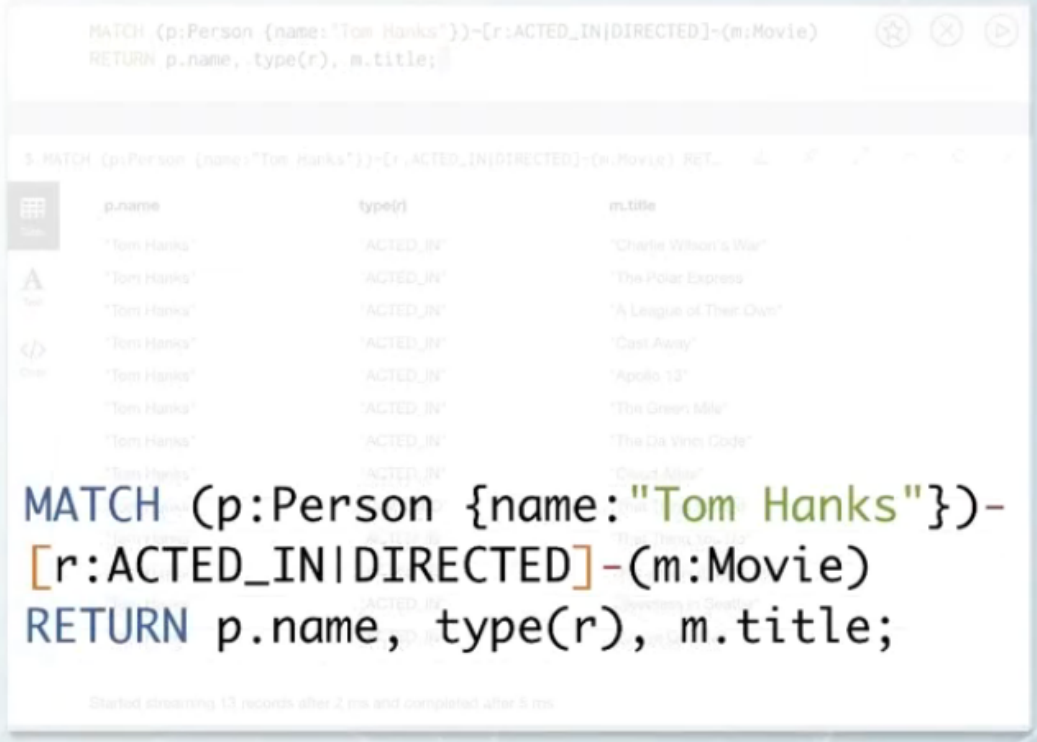

```
1 MATCH (p:Person {name:"Tom Hanks"})-[r:ACTED_IN|DIRECTED]-(m:Movie)
2 RETURN p.name, type(r), m.title;
```



Graph vs Tabular Results



```
MATCH (p:Person {name:"Tom Hanks"})-
[r:ACTED_IN|DIRECTED]-(m:Movie)
RETURN p,r,m;
```



```
MATCH (p:Person {name:"Tom Hanks"})-
[r:ACTED_IN|DIRECTED]-(m:Movie)
RETURN p.name, type(r), m.title;
```

Tabular Results

Properties are accessed using: `{variable}.{property_key}`

Case Sensitivity

Case Sensitive

Node labels

Relationship types

Property keys

Case Insensitive

Cypher keywords

Case Sensitivity

Case Sensitive	Case Insensitive
`:Person`	`MaTcH`
`:ACTED_IN`	`return`
`name`	

Tip for SETing properties

Using JSON Syntax for SET

```
MATCH
  (a:Person)-[:DRIVES]->(c:Car)
WHERE
  a.name='Ann'
SET
  c.brand='Volvo',
  c.model='V70'
RETURN
  c
```

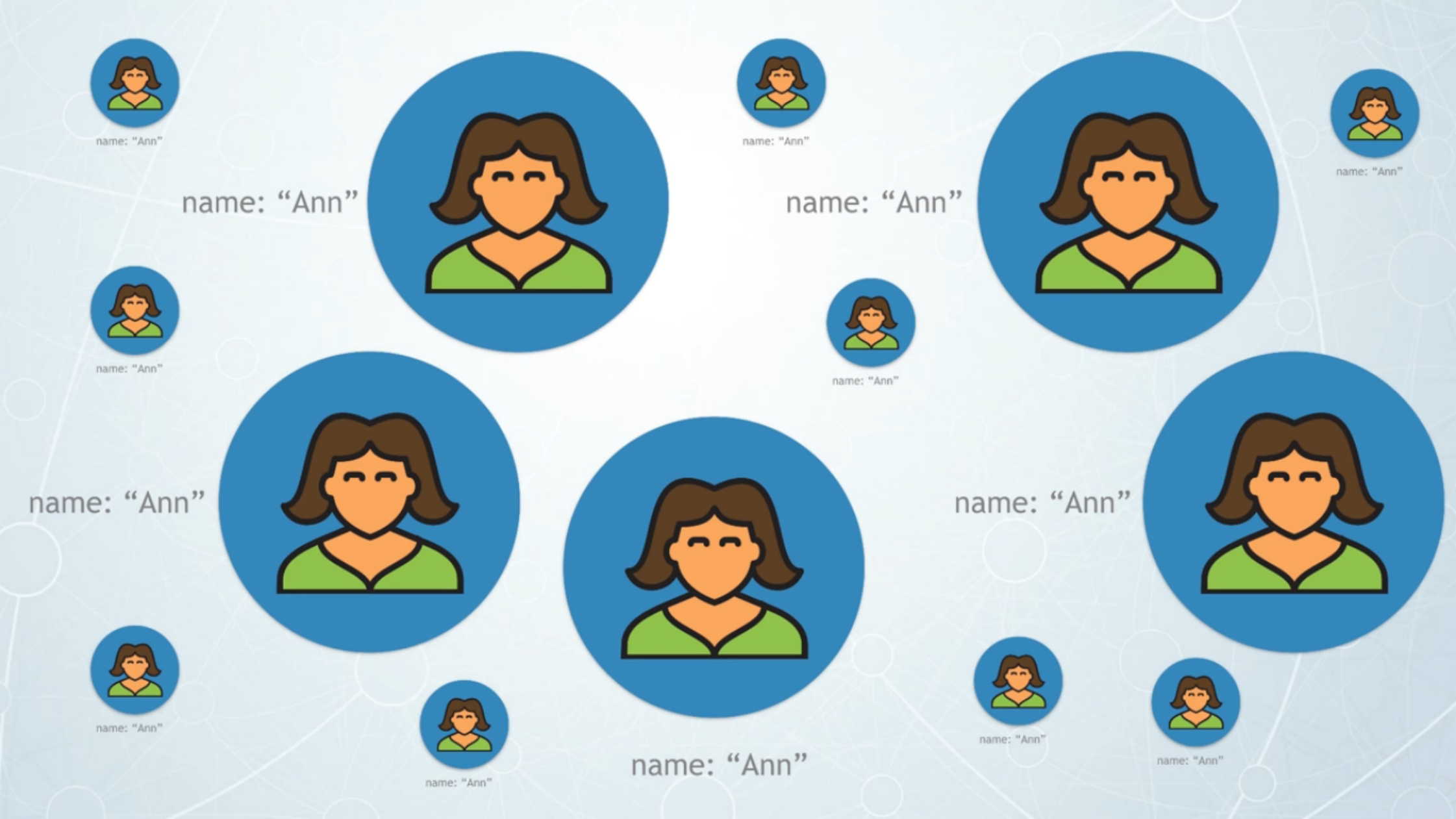
Using JSON Syntax for SET

Sometimes it's easiest to pass in an object to set properties

```
MATCH
  (a:Person)-[:DRIVES]->(c:Car)
WHERE
  a.name='Ann'
SET
  c.brand='Volvo',
  c.model='V70'
RETURN
  c
```

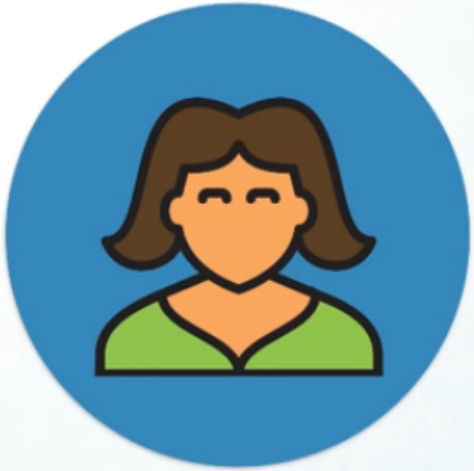
```
MATCH
  (a:Person)-[:DRIVES]->(c:Car)
WHERE
  a.name='Ann'
SET c += {brand:'Volvo', model:'V70'}
RETURN
  c
```

Ensuring Uniqueness


name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”
name: “Ann”

There can be only One!

name: “Ann”

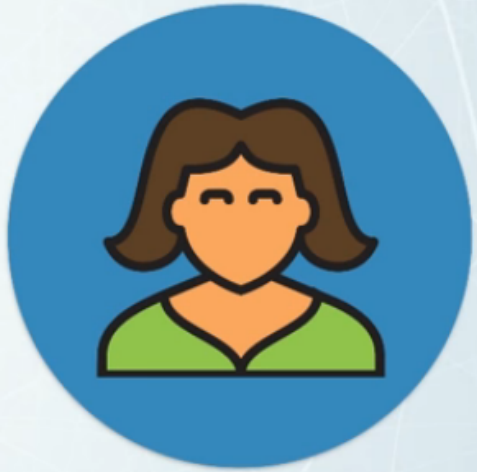


```
CREATE CONSTRAINT ON (p:Person)
ASSERT p.name IS UNIQUE
```

I want another Ann!!

name: “Ann”

name: “Ann”



```
CREATE (:Person {name:"Ann"})
```

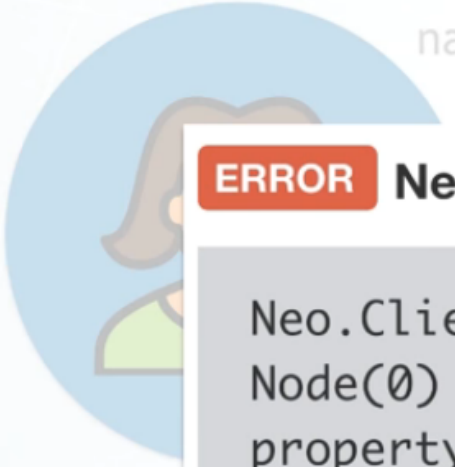
ERROR **Neo.ClientError.Schema.ConstraintValidationFailed**

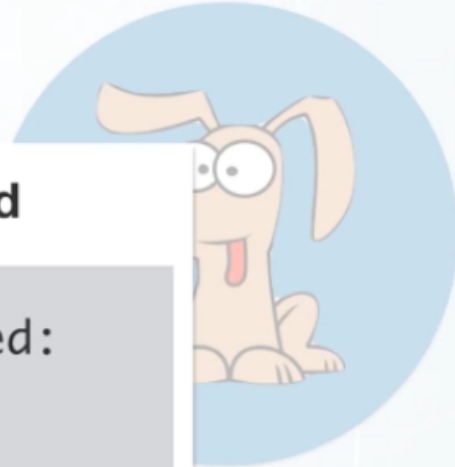
```
Neo.ClientError.Schema.ConstraintValidationFailed:
Node(0) already exists with label `Person` and
property `name` = 'Ann'
```

Uniqueness on Create

I want Ann to Have a Pet Dog

name: “Ann”





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

```
Neo.ClientError.Schema.ConstraintValidationFailed:
Node(0) already exists with label `Person` and
property `name` = 'Ann'
```

```
CREATE (a:Person {name:"Ann"})
CREATE (a)-[:HAS_PET]->(:Dog {name:"Sam"})
```

I want Ann to Have a Pet Dog


```
MERGE  (a:Person {name:"Ann"})
CREATE (a)-[:HAS_PET]->(:Dog {name:"Sam"})
```

I want Ann to Have a Pet Dog


```
MERGE (a:Person {name:"Ann"})
ON CREATE SET
  a.twitter = "@ann"
MERGE (a)-[:HAS_PET]->(:Dog {name:"Sam"})
```

Let's Aggregate

Aggregates

Aggregate queries in Cypher are a little bit different than in SQL as we don't need to specify a grouping key.

~~`GROUP BY p.name`~~

Aggregates

We implicitly group by any non-aggregate fields in the `RETURN` statement

```
// implicitly groups by p.name
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
RETURN p.name, count(*) AS numberOfMovies
```

Aggregates

```
// implicitly groups by p.name
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
RETURN p.name, count(*) AS numberOfMovies
```

p.name	numberOfMovies
"River Phoenix"	1
"Halle Berry"	1
"Ben Miles"	3
"Ethan Hawke"	1

Other Aggregating Functions

Aggregating Functions

`count(*)`
The number of matching rows.

`count(variable)`
The number of non-`null` values.

`count(DISTINCT variable)`
All aggregating functions also take the `DISTINCT` operator, which removes duplicates from the values.

`collect(n.property)`
List from the values, ignores `null`.

`sum(n.property)`
Sum numerical values. Similar functions are `avg()`, `min()`, `max()`.

`percentileDisc(n.property, $percentile)`
Discrete percentile. Continuous percentile is `percentileCont()`. The `percentile` argument is from `0.0` to `1.0`.

`stDev(n.property)`
Standard deviation for a sample of a population. For an entire population use `stDevP()`.

neo4j.com/docs/cypher-refcard/

Neo4j is Extensible: Write your Own!

6.2.3.2. Writing a user-defined aggregation function

User-defined aggregation functions are annotated with `@UserAggregationFunction`. The annotated function must return an instance of an aggregator class. An aggregator class contains one method annotated with `@UserAggregationUpdate` and one method annotated with `@UserAggregationResult`. The method annotated with `@UserAggregationUpdate` will be called multiple times and allows the class to aggregate data. When the aggregation is done the method annotated with `@UserAggregationResult` is called once and the result of the aggregation will be returned. Valid output types are `long`, `Long`, `double`, `Double`, `boolean`, `Boolean`, `String`, `Node`, `Relationship`, `Path`, `Map<String, Object>`, or `List<T>`, where `T` can be any of the supported types.

For more details, see the API documentation for user-defined aggregation functions.

The correct way to signal an error from within an aggregation function is to throw a `RuntimeException`.

```
package example;

import org.neo4j.procedure.Description;
import org.neo4j.procedure.Name;
import org.neo4j.procedure.UserAggregationFunction;
import org.neo4j.procedure.UserAggregationResult;
import org.neo4j.procedure.UserAggregationUpdate;

public class LongestString
{
    @UserAggregationFunction
    @Description( "org.neo4j.function.example.longestString(string) - aggregates the longest string found" )
    public LongStringAggregator longestString()
    {
        return new LongStringAggregator();
    }

    public static class LongStringAggregator
    {
        private int longest;
        private String longestString;

        @UserAggregationUpdate
        public void findLongest(
                @Name( "string" ) String string )
        {
            if ( string != null && string.length() > longest)
            {
                longest = string.length();
                longestString = string;
            }
        }

        @UserAggregationResult
        public String result()
        {
            return longestString;
        }
    }
}
```

User-defined aggregation functions

neo4j.com/docs/

Neo4j is Extensible: Use Open Source

Aggregation Functions

`apoc.agg.nth(value,offset)`	returns non-null value of nth row (or -1 for last) offset is 0 based
`apoc.agg.first(value)`	returns first non-null value
`apoc.agg.last(value)`	returns last non-null value
`apoc.agg.slice(value, start, length)`	returns subset of non-null values, start is 0 based and length can be -1
`apoc.agg.product(number)`	returns given product for non-null values
`apoc.agg.median(number)`	returns median for non-null numeric values
`apoc.agg.percentiles(value, [percentiles = 0.5,0.75,0.9,0.95,0.99])`	returns given percentiles for integer values
`apoc.agg.statistics(value, [percentiles = 0.5,0.75,0.9,0.95,0.99])`	returns numeric statistics (percentiles, min,minNonZero,max,total,mean,stdev) for values

Aggregation Functions in APOC library

neo4j-contrib.github.io/neo4j-apoc-procedures/

The WHERE clause

More on WHERE

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE m.released >= 2000
RETURN m.released, a.name;
```

Comparison Operators

`=, <>, <, >, <=, >=, IS NULL, IS NOT NULL`

Note on null

`null` is not `null`

null represents missing or undefined values.

You do not store a null value in a property. It just doesn't exist on that particular node.

Another Comparison Operator

=~

Regular Expressions

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE p.name =~ "K.+" OR m.released > 2000
RETURN p,r,m
```

Boolean Logic

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE p.name =~ "K.+" OR m.released > 2000
RETURN p,r,m
```

Boolean Operators

AND, OR, XOR, NOT

Grouping Boolean Operators

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE m.released > 2000 OR
  (m.released = 1997 AND m.title='As Good as It Gets')
RETURN p.name, m.title, m.released
```