

**CC7220-1**

**LA WEB DE DATOS**

**PRIMAVERA 2024**

**LECTURE 4: SPARQL [1.0]**

Aidan Hogan

aidhog@gmail.com

TODAY'S TOPIC

# SEMANTIC WEB: QUERY


DATA:

Ireland



(Ireland,partOf,Europe)  
(Ireland,isA,Country)  
(Ireland,capital,Dublin)

Dublin

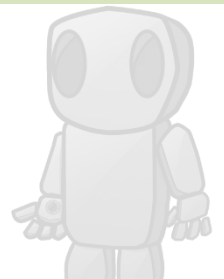


(Ireland,capital,Dublin)  
(Dublin,population,1000000)

LOGIC:  $“(b, \text{capital}, a) \rightarrow (a, \text{partOf}, b)”$   
 $“(a, \text{partOf}, b), (b, \text{partOf}, c) \rightarrow (a, \text{partOf}, c)”$

QUERY:  $“(x, \text{partOf}, y)”$

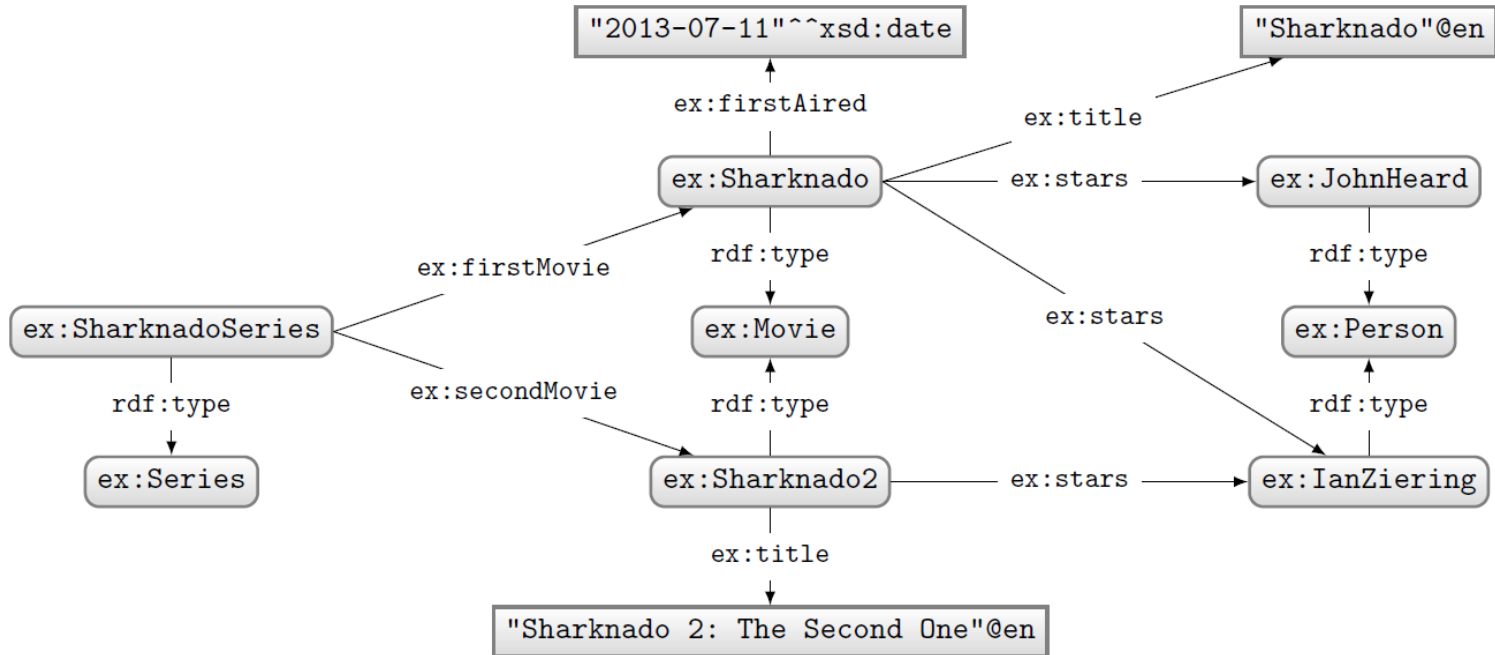
OUTPUT:  $\{(x \mapsto \text{Ireland}, y \mapsto \text{Europe}),$   
 $(x \mapsto \text{Dublin}, y \mapsto \text{Ireland}),$   
 $(x \mapsto \text{Dublin}, y \mapsto \text{Europe})\}$





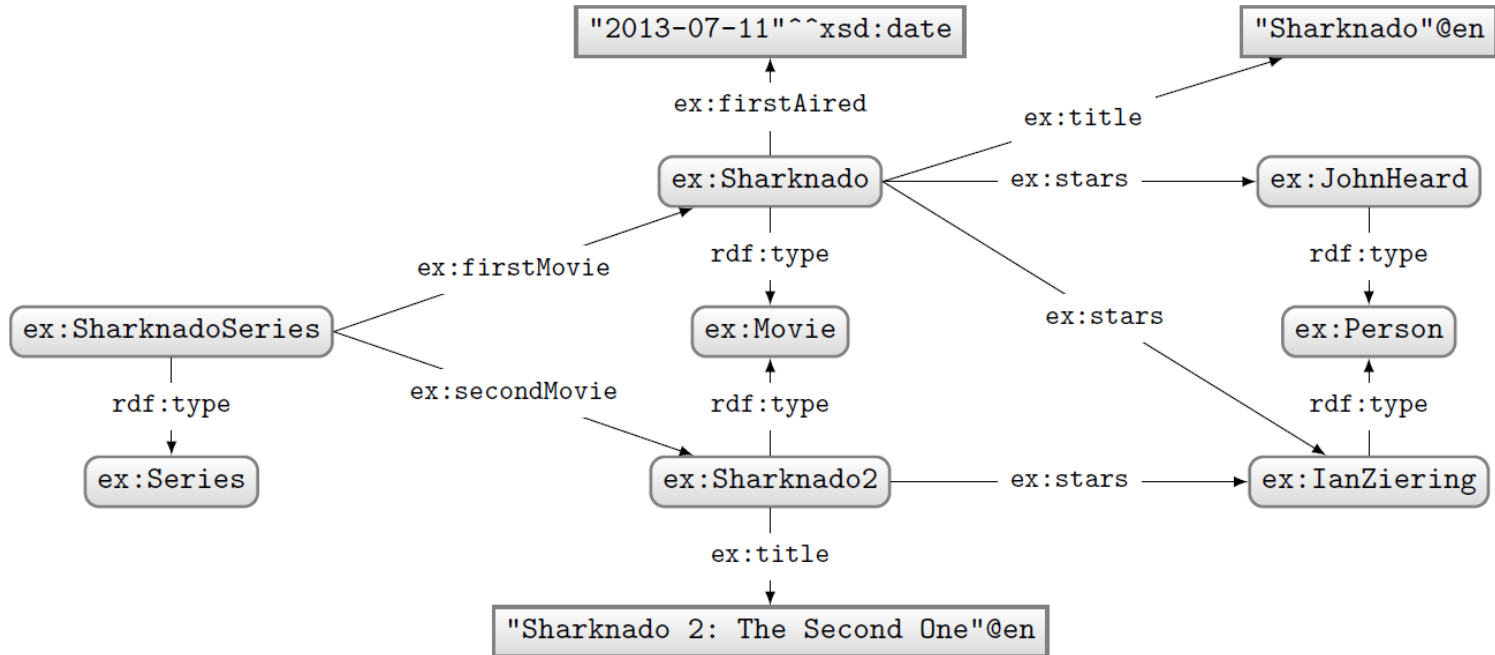
First SPARQL (1.0)  
Then SPARQL 1.1

# SPARQL: QUERY LANGUAGE FOR RDF



How to ask: “Who stars in ‘Sharknado’?”

# SPARQL: QUERY LANGUAGE FOR RDF



Query:

```
PREFIX ex: <http://ex.org/voc#>
SELECT *
WHERE {
  ex:Sharknado ex:stars ?star .
}
```

Solutions:

?star

ex:JohnHeard  
ex:IanZiering

# SPARQL: PREFIX DECLARATIONS

# SPARQL: PREFIX DECLARATIONS

- Shortcuts for IRIs (exactly like in Turtle)

```
PREFIX ex: <http://ex.org/voc#>
SELECT *
WHERE {
  ex:Sharknado ex:stars ?star .
}
```

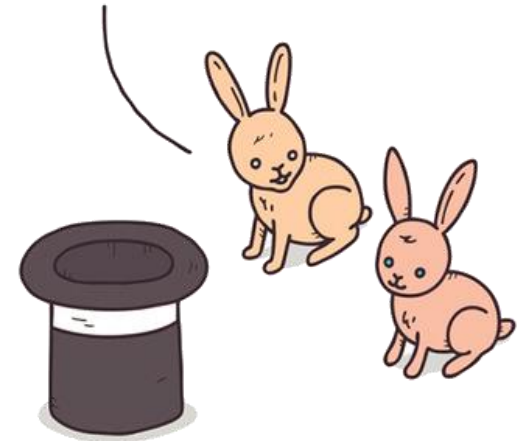


SPARQL: WHERE CLAUSE

# SPARQL: WHERE CLAUSE

- Specifies what to match in the data

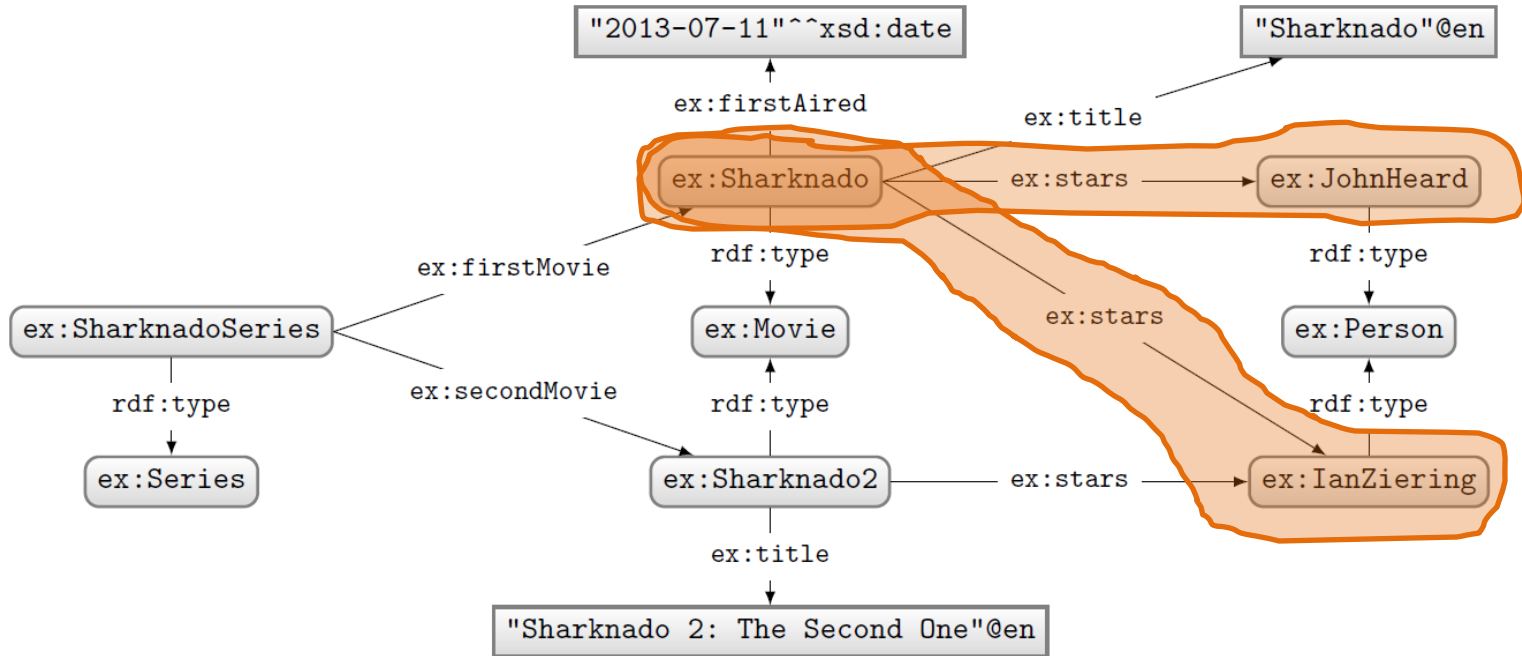
THIS IS WHERE  
THE MAGIC HAPPENS



```
PREFIX ex: <http://ex.org/voc#>
SELECT *
WHERE {
  ex:Sharknado ex:stars ?star .
}
```

“Triple pattern”  
(a triple with variables)

# SPARQL: WHERE CLAUSE



Query:

```
PREFIX ex: <http://ex.org/voc#>
SELECT *
WHERE {
  ex:Sharknado ex:stars ?star .
}
```

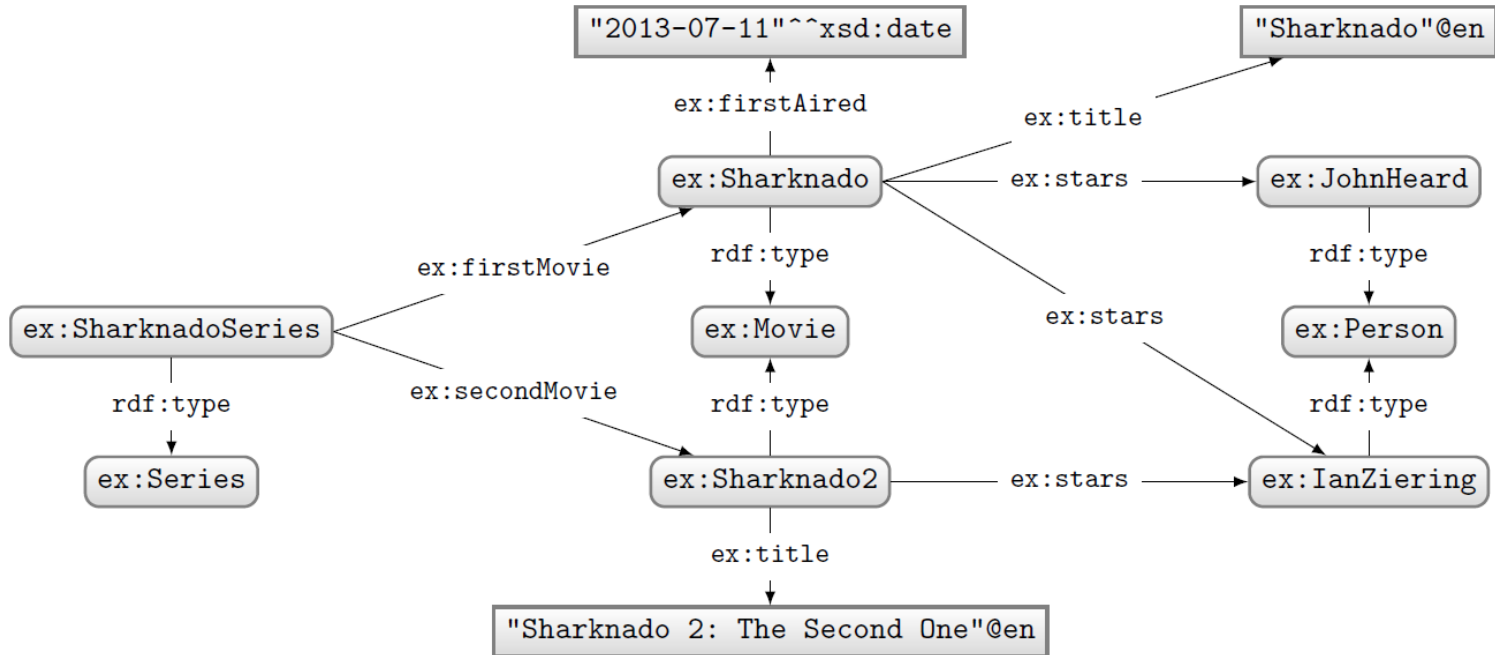
Solutions:

?star

ex:JohnHeard

ex:IanZiering

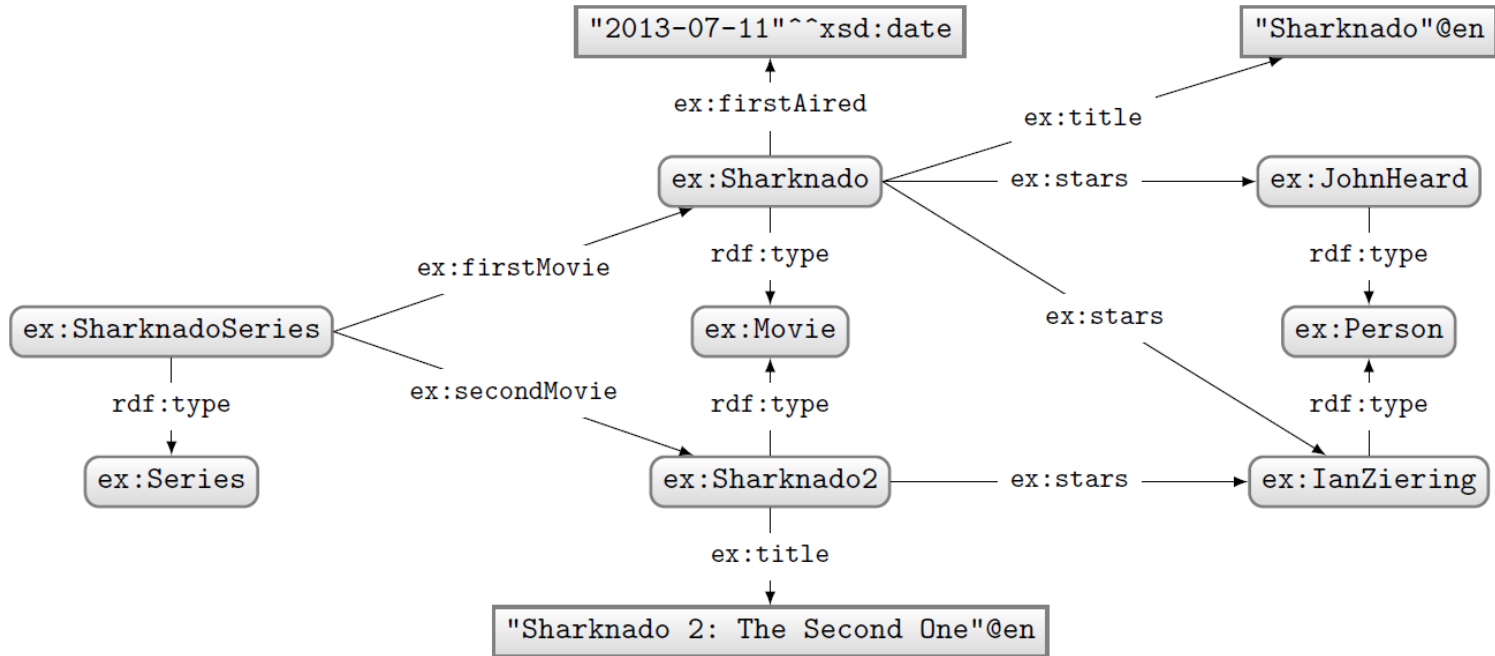
# SPARQL: WHERE CLAUSE



How to ask: “What movies did the stars of ‘Sharknado’ also star in?”

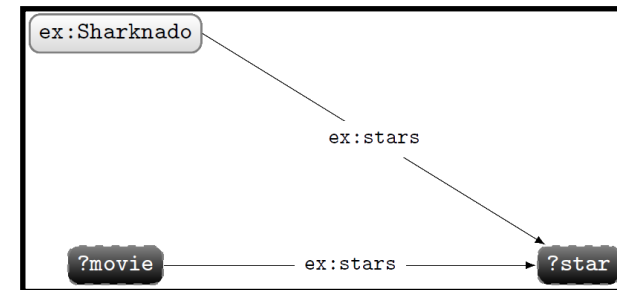


# SPARQL: BASIC GRAPH PATTERNS



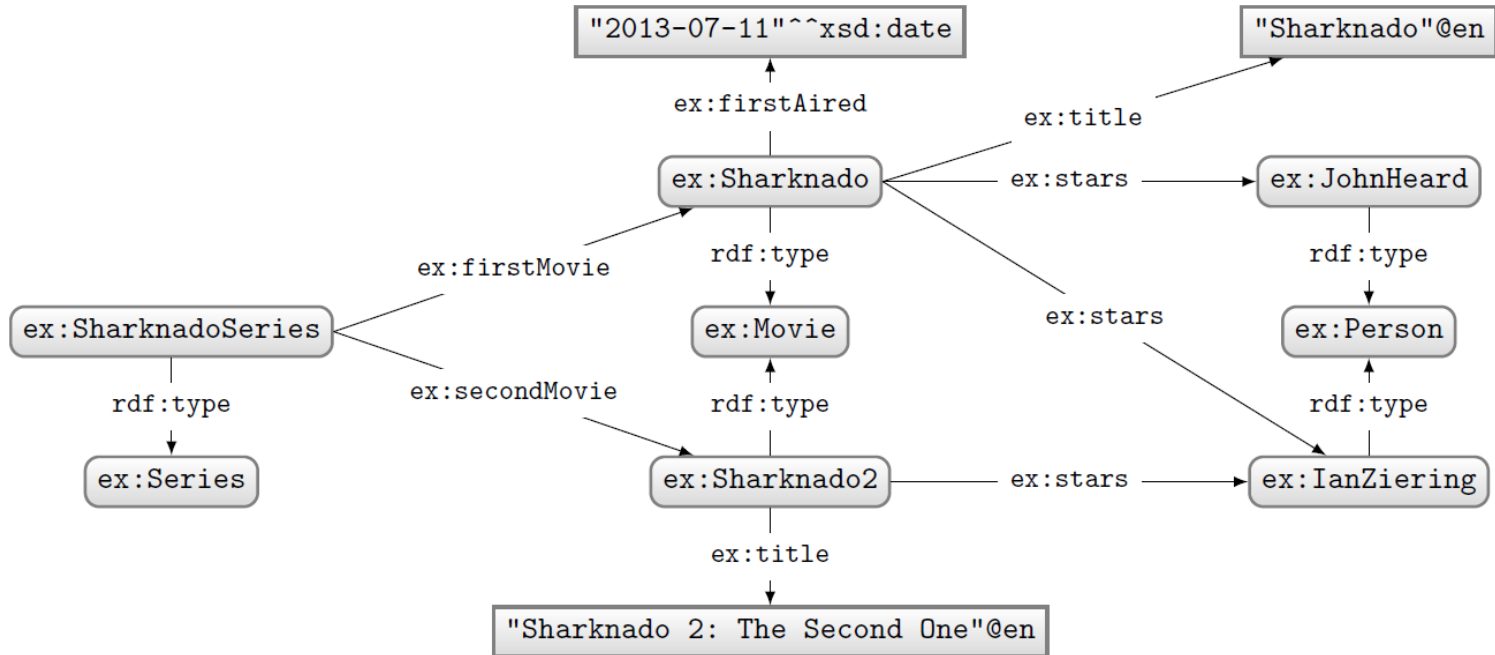
Query:

```
PREFIX ex: <http://ex.org/voc#>
SELECT *
WHERE {
  ex:Sharknado ex:stars ?star .
  ?movie ex:stars ?star .
}
```



**"Basic Graph Pattern"**  
(a set of triple patterns)

# SPARQL: JOIN VARIABLES



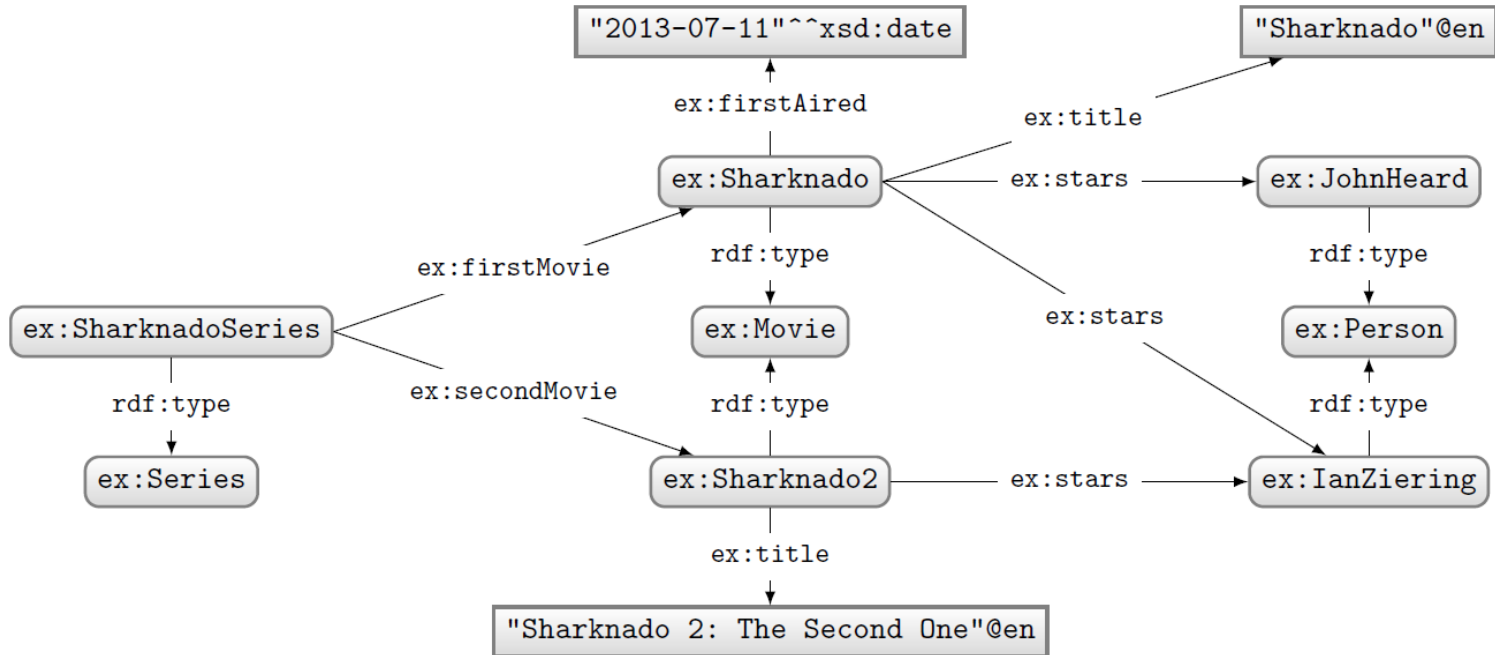
Query:

```
PREFIX ex: <http://ex.org/voc#>
SELECT *
WHERE {
  ex:Sharknado ex:stars ?star
  ?movie ex:stars ?star
}
```

“Join Variable”

(a variable appearing in more than one triple pattern)

# SPARQL: DISJUNCTION

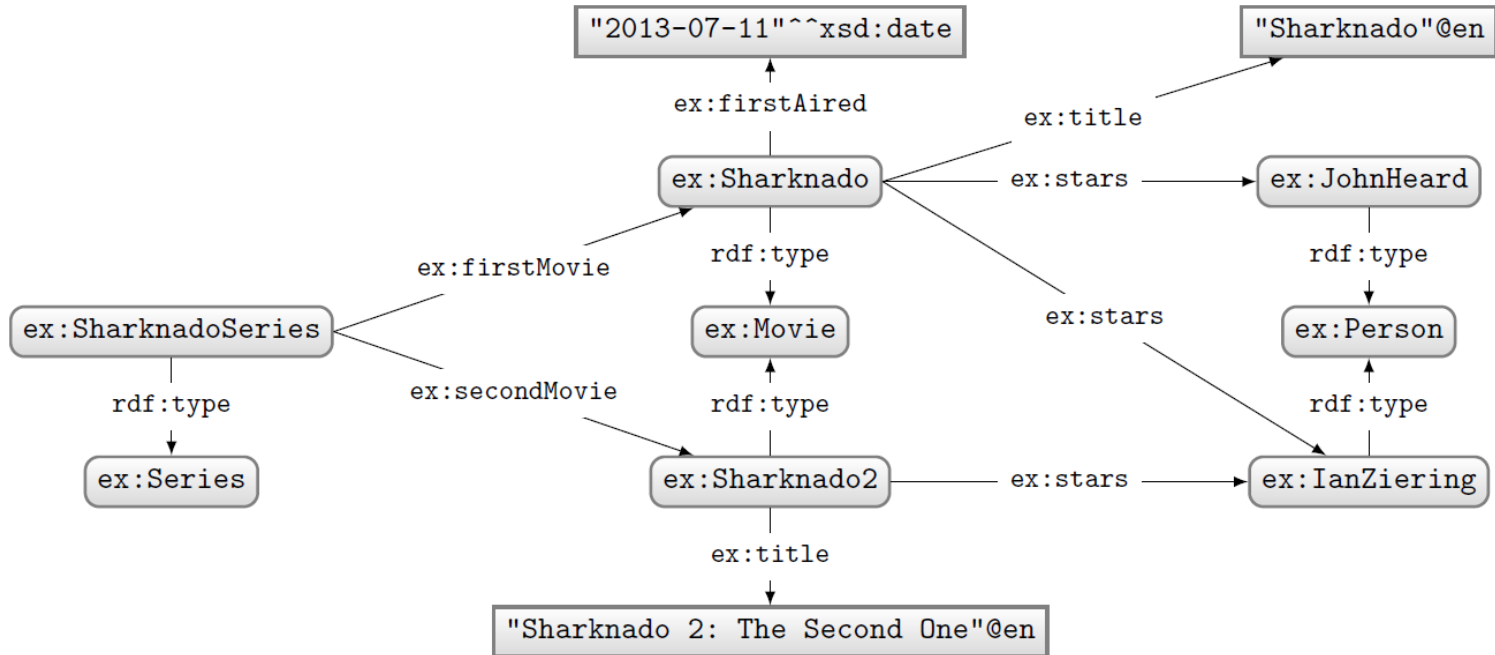


How to ask: “What are the titles of the (first two) movies in the Sharknado series?”



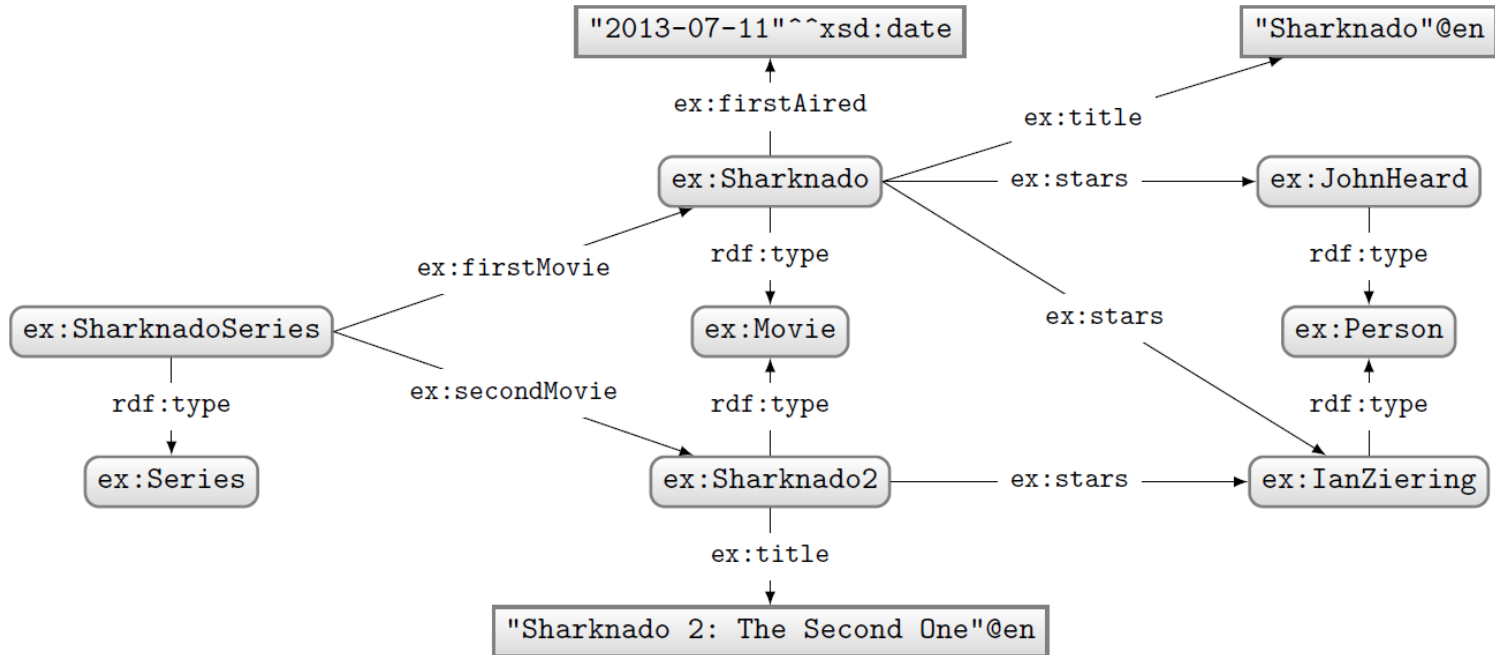


# SPARQL: LEFT-JOIN



How to ask: “Give me the titles of all movies and, if available, their first-aired date?”

# SPARQL: LEFT-JOIN (OPTIONAL)



Query:

```
PREFIX ex: <http://ex.org/voc#>
SELECT *
WHERE {
  ?movie a ex:Movie ; ex:title ?title .
  OPTIONAL { ?movie ex:firstAired ?date }
}
```

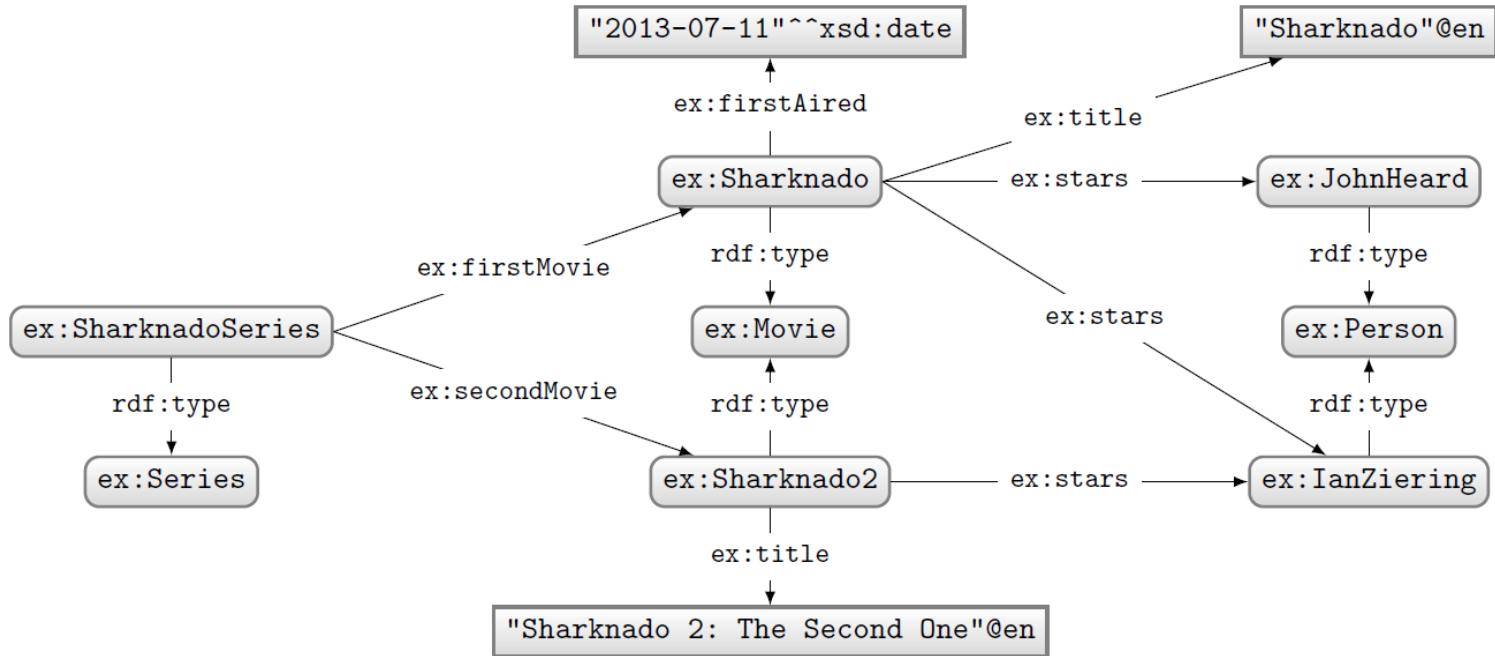
Solutions:

?movie	?title	?date
ex:Sharknado	"Sharknado"@en	"2013-07-11"^^xsd:date
ex:Sharknado2	"Sharknado 2: The Second One"@en	

“UNBOUND Variable”

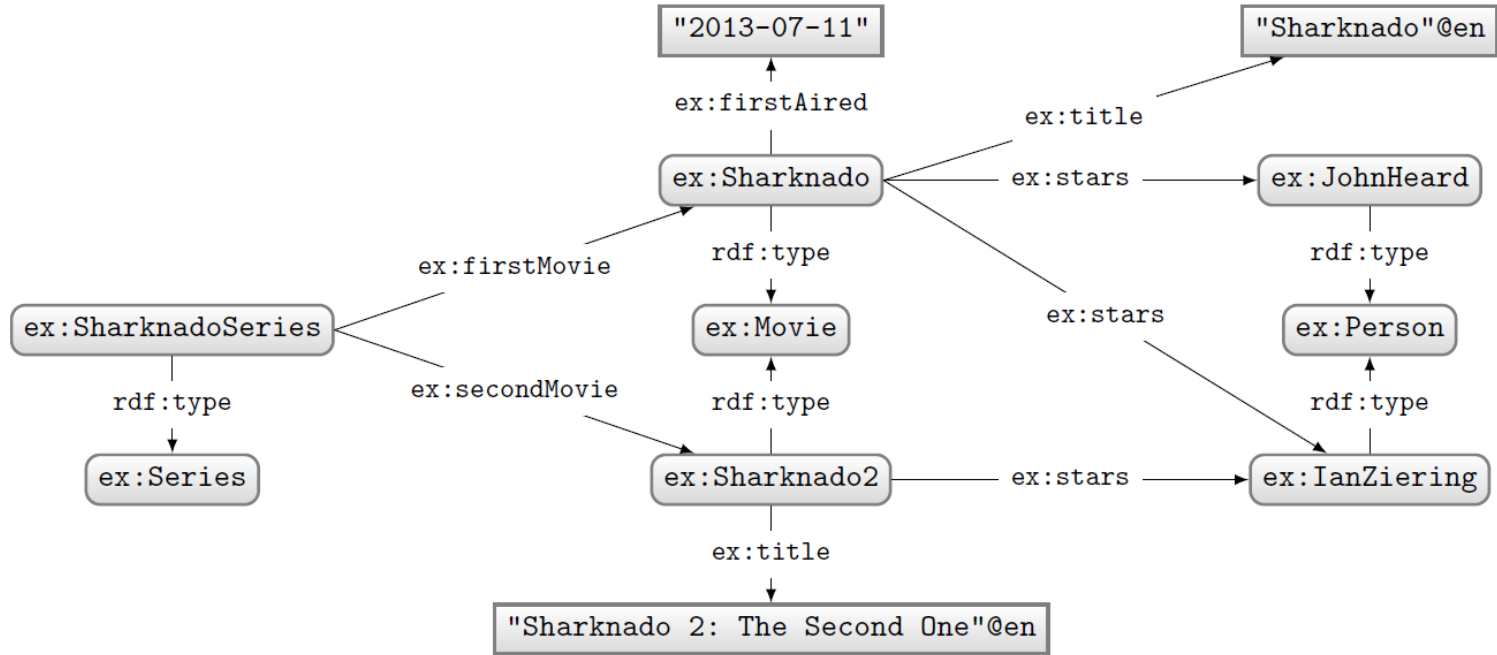
(a variable without a binding in a solution)

# SPARQL: FILTERING RESULTS



How to ask: "What movies were first aired in 2014?"

# SPARQL: FILTER



Query:

```
PREFIX ex: <http://ex.org/voc#>
SELECT *
WHERE {
  ?movie a ex:Movie ; ex:firstAired ?date .
  FILTER(?date > "2013-12-31"^^xsd:date
    && ?date <= "2014-12-31"^^xsd:date)
}
```

Solutions:

?movie	?date
--------	-------

"Empty Results"

# SPARQL: FILTER

## Query:

```
PREFIX ex: <http://ex.org/voc#>
SELECT *
WHERE {
  ?movie a ex:Movie ; ex:firstAired ?date .
  FILTER(?date > "2013-12-31"^^xsd:date
    && ?date <= "2014-12-31"^^xsd:date)
}
```

What happens in this case where ?date bound in data to a string?

**FILTERS** (and other functions we see later) expect certain types. If not given, a type error is given.

# SPARQL: BOOLEAN FILTER OPERATORS

- **FILTER**s evaluate as true, false or error
- Only results evaluating as true are returned
- Can apply AND (&&) or OR (||)
- Can also apply NOT (!)
  - !E → E

A	B	A    B	A && B
T	T	T	T
T	F	T	F
F	T	T	F
F	F	F	F
T	E	T	E
E	T	T	E
F	E	E	F
E	F	E	F
E	E	E	E

# SPARQL OPERATORS

<i>A</i>	Op	<i>B</i>	Return type and value	
	!	BOOL <i>b</i>	BOOL	true if $I_L(b)$ is false; false otherwise
BOOL <i>b</i> <sub>1</sub>		BOOL <i>b</i> <sub>2</sub>	BOOL	true if $I_L(b_1)$ or $I_L(b_2)$ ; false otherwise
BOOL <i>b</i> <sub>1</sub>	&&	BOOL <i>b</i> <sub>2</sub>	BOOL	true if $I_L(b_1)$ and $I_L(b_2)$ ; false otherwise
TERM* <i>t</i> <sub>1</sub>	=	TERM* <i>t</i> <sub>2</sub>	BOOL	true if <i>t</i> <sub>1</sub> same term as <i>t</i> <sub>2</sub> ; false otherwise
TERM* <i>t</i> <sub>1</sub>	!=	TERM* <i>t</i> <sub>2</sub>	BOOL	true if <i>t</i> <sub>1</sub> not same term as <i>t</i> <sub>2</sub> ; false otherwise
COM <i>v</i> <sub>1</sub>	=	COM <i>v</i> <sub>2</sub>	BOOL	true if $I_L(v_1) = I_L(v_2)$ ; false otherwise
COM <i>v</i> <sub>1</sub>	!=	COM <i>v</i> <sub>2</sub>	BOOL	true if $I_L(v_1) \neq I_L(v_2)$ ; false otherwise
COM <i>v</i> <sub>1</sub>	<	COM <i>v</i> <sub>2</sub>	BOOL	true if $I_L(v_1) < I_L(v_2)$ ; false otherwise
COM <i>v</i> <sub>1</sub>	>	COM <i>v</i> <sub>2</sub>	BOOL	true if $I_L(v_1) > I_L(v_2)$ ; false otherwise
COM <i>v</i> <sub>1</sub>	<=	COM <i>v</i> <sub>2</sub>	BOOL	true if $I_L(v_1) \leq I_L(v_2)$ ; false otherwise
COM <i>v</i> <sub>1</sub>	>=	COM <i>v</i> <sub>2</sub>	BOOL	true if $I_L(v_1) \geq I_L(v_2)$ ; false otherwise
	+	NUM <i>n</i>	NUM	<i>n</i>
	-	NUM <i>n</i>	NUM	- <i>n</i>
NUM <i>n</i> <sub>1</sub>	+	NUM <i>n</i> <sub>2</sub>	NUM	$I_L(v_1) + I_L(v_2)$
NUM <i>n</i> <sub>1</sub>	-	NUM <i>n</i> <sub>2</sub>	NUM	$I_L(v_1) - I_L(v_2)$
NUM <i>n</i> <sub>1</sub>	*	NUM <i>n</i> <sub>2</sub>	NUM	$I_L(v_1) \times I_L(v_2)$
NUM <i>n</i> <sub>1</sub>	/	NUM <i>n</i> <sub>2</sub>	NUM	$\frac{I_L(v_1)}{I_L(v_2)}$

- **COM**: a comparable literal value
- **TERM\***: a non-comparable RDF term
- $I_L(\cdot)$ : the value (e.g., 2 not "2")



# SPARQL FUNCTIONS: EXISTENCE, EQUALITY, IF ...

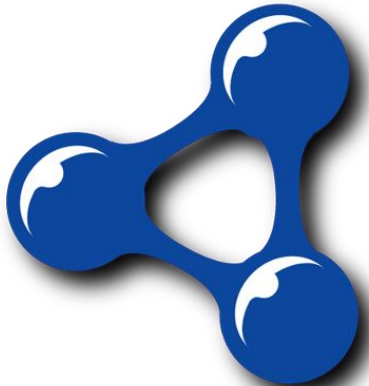
Function	Return type and value
<code>bound</code> ( <small>TERM</small> $t$ )	<small>BOOL</small> true if $t$ is bound; false if unbound
<code>if</code> ( <small>BOOL</small> $b$ , <small>TERM</small> $t_1$ , <small>TERM</small> $t_2$ )	<small>TERM</small> $t_1$ if $b$ is true; $t_2$ otherwise
<code>coalesce</code> ( <small>TERM</small> $t_1, \dots, t_n$ )	<small>TERM</small> first $t_i$ ( $1 \leq i \leq n$ ) that is not an error or unbound
<code>not exists</code> ( <small>SUB</small> $Q$ )	<small>BOOL</small> true if $Q$ has any solution; false otherwise
<code>exists</code> ( <small>SUB</small> $Q$ )	<small>BOOL</small> true if $Q$ has no solution; false otherwise
<code>sameTerm</code> ( <small>TERM</small> $t_1$ , <small>TERM</small> $t_2$ )	<small>BOOL</small> true if $t_1$ same term as $t_2$ ; false otherwise
<small>TERM</small> $t$ <code>in</code> ( <small>TERM</small> $t_1, \dots, t_n$ )	<small>BOOL</small> true if $t = t_i$ for any $t_i \in \{t_1, \dots, t_n\}$ ; false otherwise
<small>TERM</small> $t$ <code>not in</code> ( <small>TERM</small> $t_1, \dots, t_n$ )	<small>BOOL</small> true if $t \neq t_i$ for all $t_i \in \{t_1, \dots, t_n\}$ ; false otherwise



- SUB: a sub-query

# SPARQL FUNCTIONS: TERMS

Function	Return type and value
<code>isIRI</code> ( <i>TERM</i> <i>t</i> )	<b>BOOL</b> true if <i>t</i> is an IRI; false otherwise
<code>isBlank</code> ( <i>TERM</i> <i>t</i> )	<b>BOOL</b> true if <i>t</i> is a blank node; false otherwise
<code>isLiteral</code> ( <i>TERM</i> <i>t</i> )	<b>BOOL</b> true if <i>t</i> is a literal; false otherwise
<code>isNumeric</code> ( <i>TERM</i> <i>t</i> )	<b>BOOL</b> true if <i>t</i> is a numeric value; false otherwise
<code>str</code> ( <i>LIT</i> <i>l</i>   <i>IRI</i> <i>i</i> )	<b>STR</b> lexical value of <i>l</i>   string of <i>i</i>
<code>lang</code> ( <i>LIT</i> <i>l</i> )	<b>STR</b> language tag string of <i>l</i>
<code>datatype</code> ( <i>LIT</i> <i>l</i> )	<b>IRI</b> datatype IRI of <i>l</i>
<code>iri</code> ( <i>STR</i> <i>s</i>   <i>IRI</i> <i>i</i> )	<b>IRI</b> <i>s</i> resolved against the in-scope base IRI   <i>i</i>
<code>bnode</code> ([ <i>STR</i> <i>s</i> ])	<b>BNODE</b> fresh blank node [unique to <i>s</i> ]
<code>strdt</code> ( <i>STR</i> <i>s</i> , <i>IRI</i> <i>i</i> )	<b>LIT</b> " <i>s</i> " <sup>~</sup> < <i>i</i> >
<code>strlang</code> ( <i>STR</i> <i>s</i> , <i>STR</i> <i>l</i> )	<b>LIT</b> " <i>s</i> "@ <i>l</i>
<code>uuid</code> ()	<b>IRI</b> fresh IRI (from UUID URN scheme)
<code>struuid</code> ()	<b>STR</b> fresh string (from UUID URN scheme)



- *a|b* indicates *a* or *b*
- [*a*] indicates *a* optional

# SPARQL FUNCTIONS: STRINGS

---

Function	Return type and value
<code>strlen</code> (STR <i>s</i> )	INT length of string <i>s</i>
<code>substr</code> (STR <i>s</i> , INT <i>b</i> , [INT <i>l</i> ])	STR substring of <i>s</i> from index <i>b</i> [of length <i>l</i> ]
<code>ucase</code> (STR <i>s</i> )	STR uppercase <i>s</i>
<code>lcase</code> (STR <i>s</i> )	STR lowercase <i>s</i>
<code>strstarts</code> (STR <i>s</i> , STR <i>p</i> )	BOOL true if <i>s</i> starts with <i>p</i> ; false otherwise
<code>strends</code> (STR <i>s</i> , STR <i>p</i> )	BOOL true if <i>s</i> ends with <i>p</i> ; false otherwise
<code>strbefore</code> (STR <i>s</i> , STR <i>p</i> )	STR string before first match for <i>p</i> in <i>s</i>
<code>strafter</code> (STR <i>s</i> , STR <i>p</i> )	STR string after first match for <i>p</i> in <i>s</i>
<code>encode_for_iri</code> (STR <i>s</i> )	STR <i>s</i> percent-encoded
<code>concat</code> (STR <i>s</i> <sub>1</sub> , ..., <i>s</i> <sub><i>n</i></sub> )	STR <i>s</i> <sub>1</sub> , ..., <i>s</i> <sub><i>n</i></sub> concatenated
<code>langMatches</code> (STR <i>s</i> , STR <i>l</i> )	BOOL true if <i>s</i> a language tag matching <i>l</i> ; false otherwise
<code>regex</code> (STR <i>s</i> , STR <i>p</i> , [STR <i>f</i> ])	BOOL true if <i>s</i> matches regex <i>p</i> [with flags <i>f</i> ]; false otherwise
<code>replace</code> (STR <i>s</i> , STR <i>p</i> , STR <i>r</i> , [STR <i>f</i> ])	STR <i>s</i> with matches for regex <i>p</i> [with flags <i>f</i> ] replaced by <i>r</i>

---



# SPARQL FUNCTIONS: NUMERICS

Function	Return type and value
<code>abs</code> (NUM <i>n</i> )	NUM absolute value of <i>n</i>
<code>round</code> (NUM <i>n</i> )	NUM round to nearest whole number (towards $+\infty$ for *.5)
<code>ceil</code> (NUM <i>n</i> )	NUM round up (towards $+\infty$ ) to nearest whole number
<code>floor</code> (NUM <i>n</i> )	NUM round down (towards $-\infty$ ) to nearest whole number
<code>rand</code> (NUM <i>n</i> )	NUM random double between 0 (inclusive) and 1 (exclusive)



# SPARQL FUNCTIONS: TEMPORAL

Function	Return type and value	
<code>now()</code>	DT	current date-time
<code>year(DT <i>d</i>)</code>	INT	year of <i>d</i> (as an integer)
<code>month(DT <i>d</i>)</code>	INT	month of <i>d</i> (as an integer)
<code>day(DT <i>d</i>)</code>	INT	day of <i>d</i> (as an integer)
<code>hours(DT <i>d</i>)</code>	INT	hours of <i>d</i> (as an integer)
<code>minutes(DT <i>d</i>)</code>	INT	minutes of <i>d</i> (as an integer)
<code>seconds(DT <i>d</i>)</code>	INT	seconds of <i>d</i> (as an integer)
<code>timezone(DT <i>d</i>)</code>	DTD	timezone of <i>d</i> (as day-time-duration)
<code>tz(DT <i>d</i>)</code>	STR	timezone of <i>d</i> (as a string)



- DT: date-time
- DTD: day-time-duration

# SPARQL FUNCTIONS: HASHING

Function	Return type and value	
<code>md5</code> ( <i>STR</i> <i>s</i> )	<i>STR</i>	MD5 hash of <i>s</i>
<code>sha1</code> ( <i>STR</i> <i>s</i> )	<i>STR</i>	SHA1 hash of <i>s</i>
<code>sha256</code> ( <i>STR</i> <i>s</i> )	<i>STR</i>	SHA256 hash of <i>s</i>
<code>sha384</code> ( <i>STR</i> <i>s</i> )	<i>STR</i>	SHA384 hash of <i>s</i>
<code>sha512</code> ( <i>STR</i> <i>s</i> )	<i>STR</i>	SHA512 hash of <i>s</i>



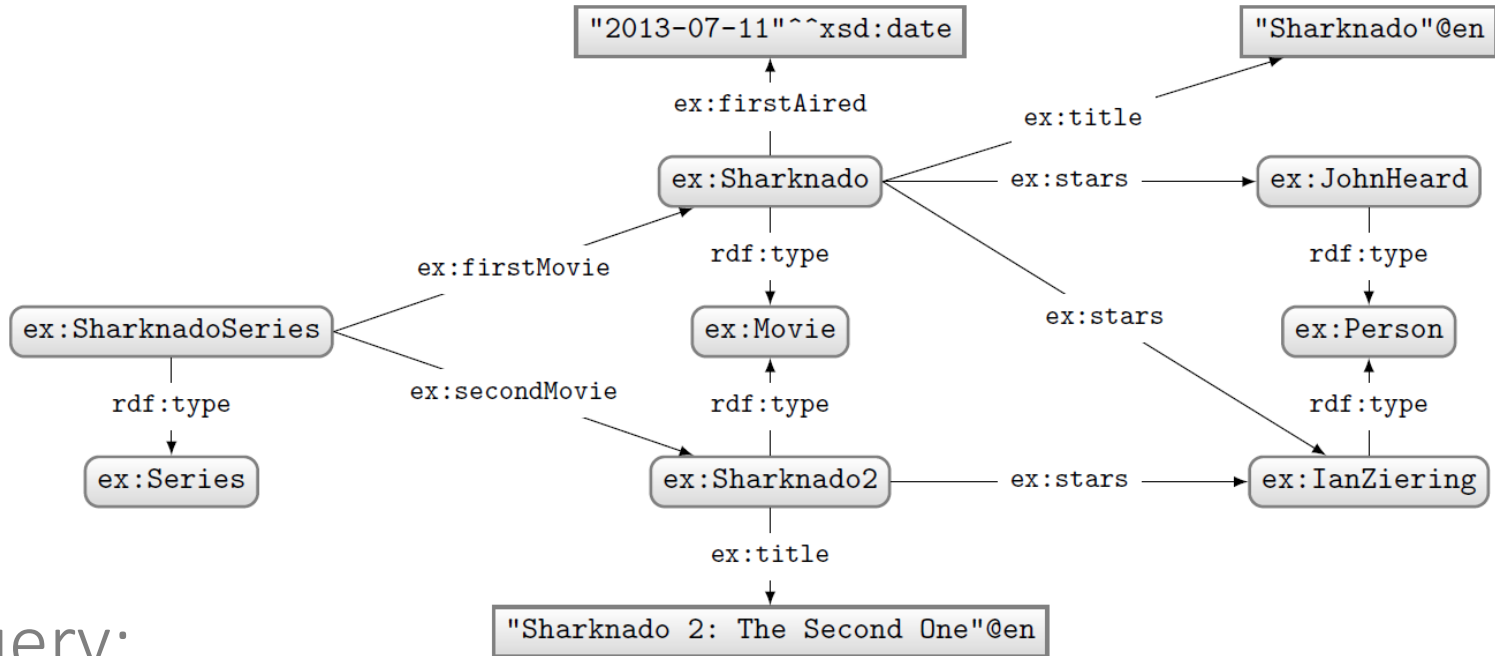
# SPARQL: CASTING BETWEEN TYPES

- **Y**: always allowed
- **N**: never allowed
- **M**: depends on value
  - e.g., "2"^^xsd:string can be mapped to xsd:int but "P"^^xsd:string cannot

From \ To	str	flt	dbl	dec	int	dT	bool
str	Y	M	M	M	M	M	M
flt	Y	Y	Y	M	M	N	Y
dbl	Y	Y	Y	M	M	N	Y
dec	Y	Y	Y	Y	Y	N	Y
int	Y	Y	Y	Y	Y	N	Y
dT	Y	N	N	N	N	Y	N
bool	Y	Y	Y	Y	Y	N	Y
IRI	Y	N	N	N	N	N	N
ltrl	Y	M	M	M	M	M	M

bool = [xsd:boolean](#)  
dbl = [xsd:double](#)  
flt = [xsd:float](#)  
dec = [xsd:decimal](#)  
int = [xsd:integer](#)  
dT = [xsd:dateTime](#)  
str = [xsd:string](#)  
IRI = IRI  
ltrl = simple literal

# SPARQL: WHERE CLAUSE EXAMPLE (I)



Query:

```
PREFIX ex: <http://ex.org/voc#>
SELECT *
WHERE {
  { ex:SharknadoSeries ex:firstMovie ?movie . }
  UNION
  { ex:SharknadoSeries ex:secondMovie ?movie . }
  OPTIONAL
  { ?movie ex:firstAired ?date . }
  ?movie ex:title ?title .
  FILTER(REGEX(STR(?title),"*[0-9]*"))
}
```

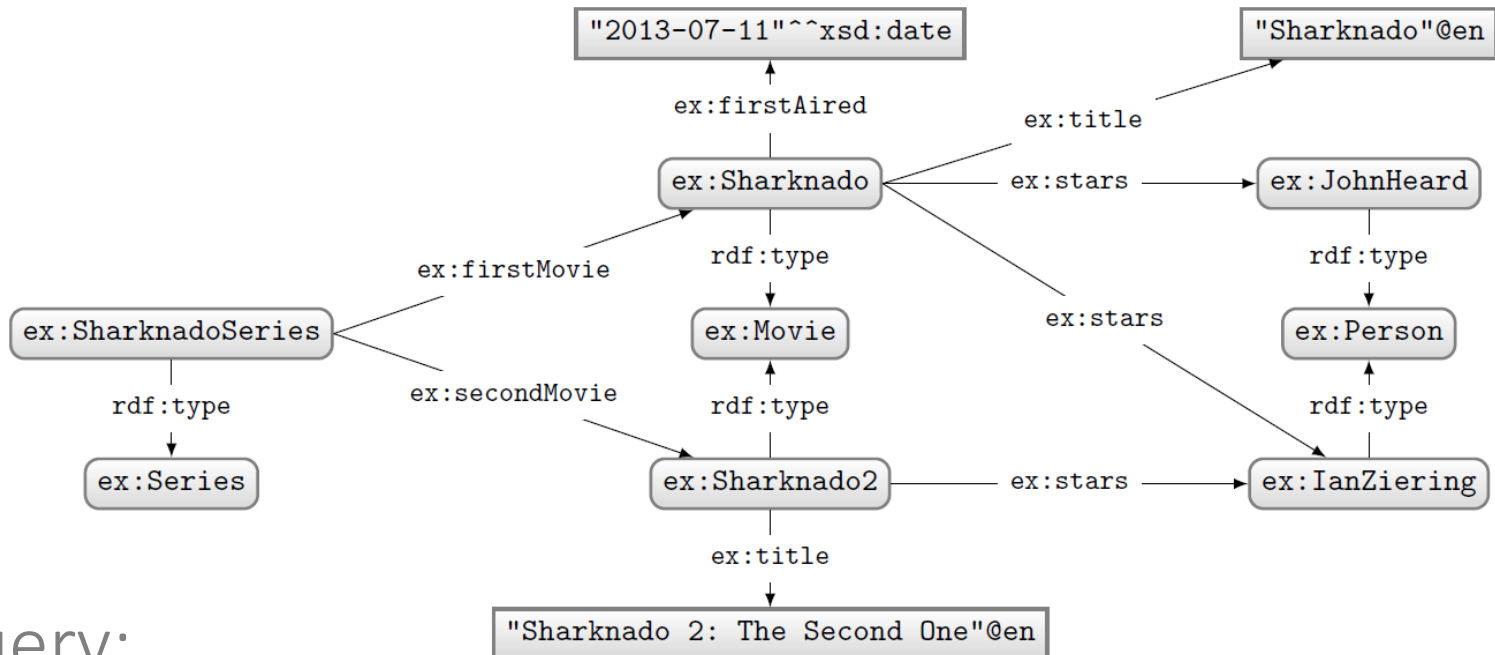
What solutions would this query return?

Solutions:

?movie	?title	?date
ex:Sharknado2	"Sharknado 2: The Second One"@en	



# SPARQL: WHERE CLAUSE EXAMPLE (II)



Query:

```
PREFIX ex: <http://ex.org/voc#>
SELECT *
WHERE {
  ?movie a ex:Movie .
  OPTIONAL
  { ?movie ex:firstAired ?date . }
  FILTER(!BOUND(?date))
}
```

What solutions would this query return?

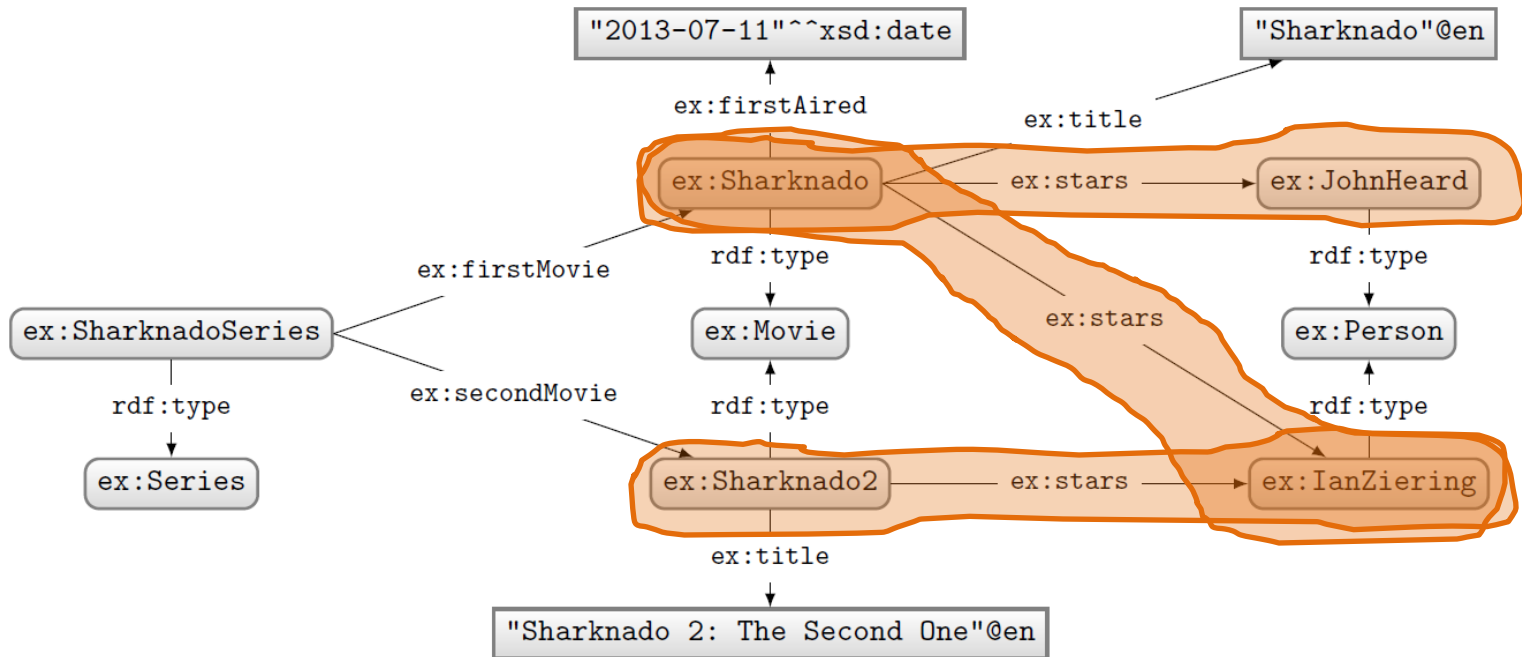
Solutions:

?movie	?date
ex:Sharknado2	

Can do negation!

# SPARQL: QUERY TYPES

# SPARQL: SELECT WITH \*



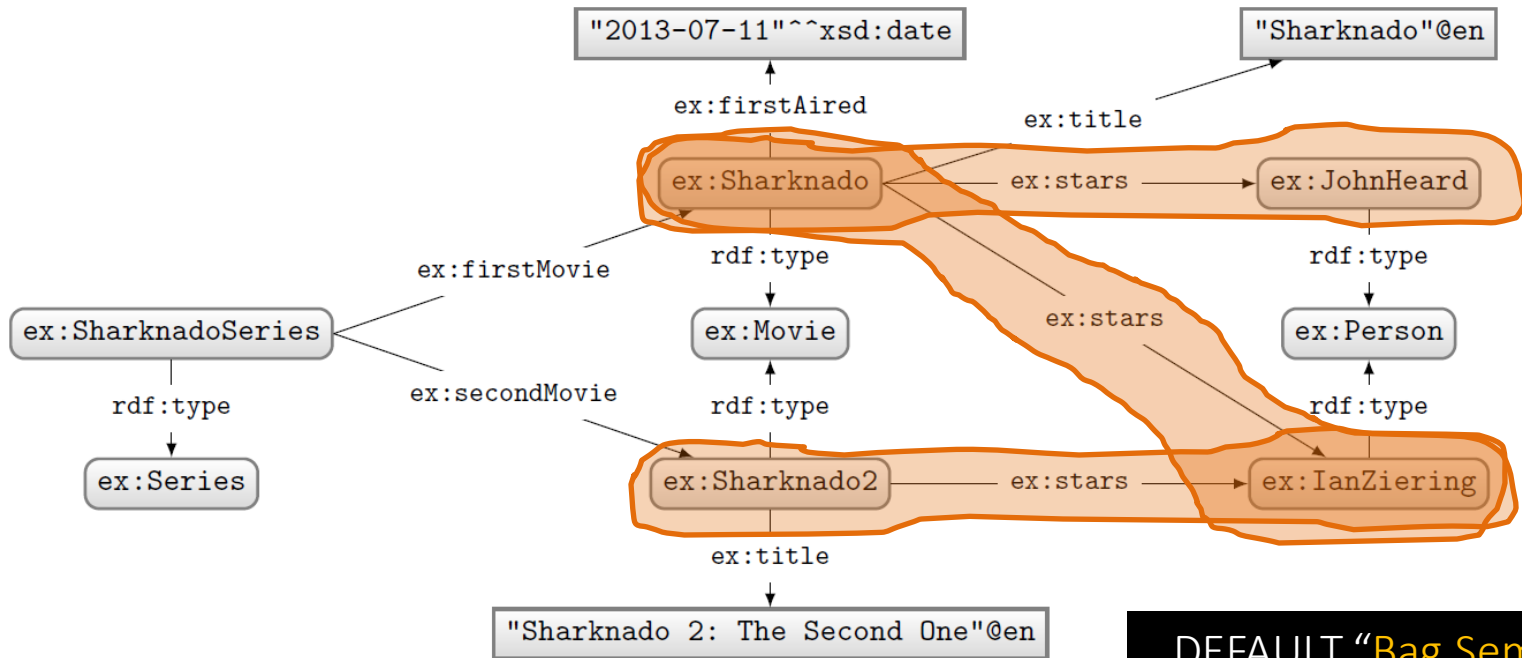
Query:

```
PREFIX ex: <http://ex.org/voc#>
SELECT *
WHERE {
  ?movie a ex:Movie.
  ?movie ex:stars ?star .
}
```

Solutions:

?movie	?star
ex:Sharknado	ex:JohnHeard
ex:Sharknado	ex:IanZiering
ex:Sharknado2	ex:IanZiering

# SPARQL: SELECT WITH PROJECTION



Query:

```
PREFIX ex: <http://ex.org/voc#>
SELECT ?star
WHERE {
  ?movie a ex:Movie.
  ?movie ex:stars ?star .
}
```

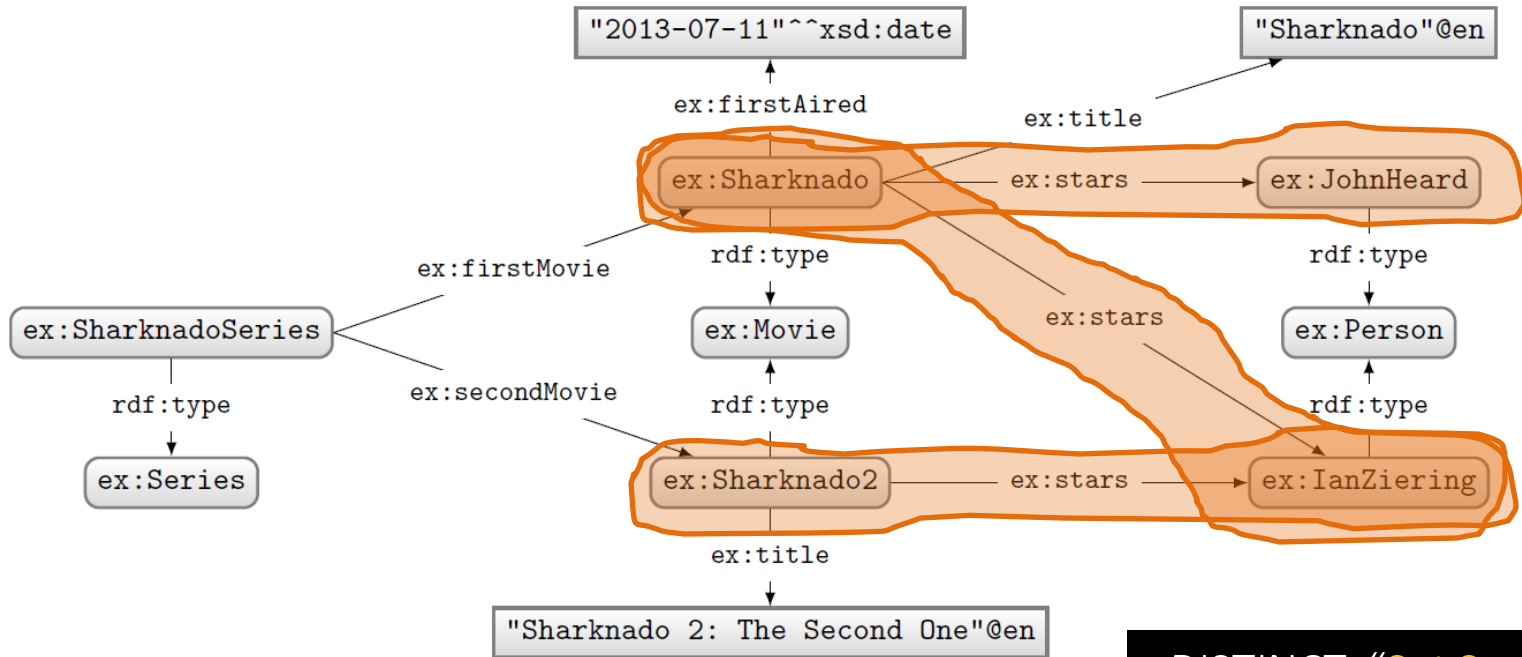
Solutions:

?star
ex:JohnHeard
ex:IanZiering
ex:IanZiering

DEFAULT "Bag Semantics"

(number of results returned must correspond to number of matches in data)

# SPARQL: SELECT WITH DISTINCT



Query:

```
PREFIX ex: <http://ex.org/voc#>
SELECT DISTINCT ?star
WHERE {
  ?movie a ex:Movie.
  ?movie ex:stars ?star .
}
```

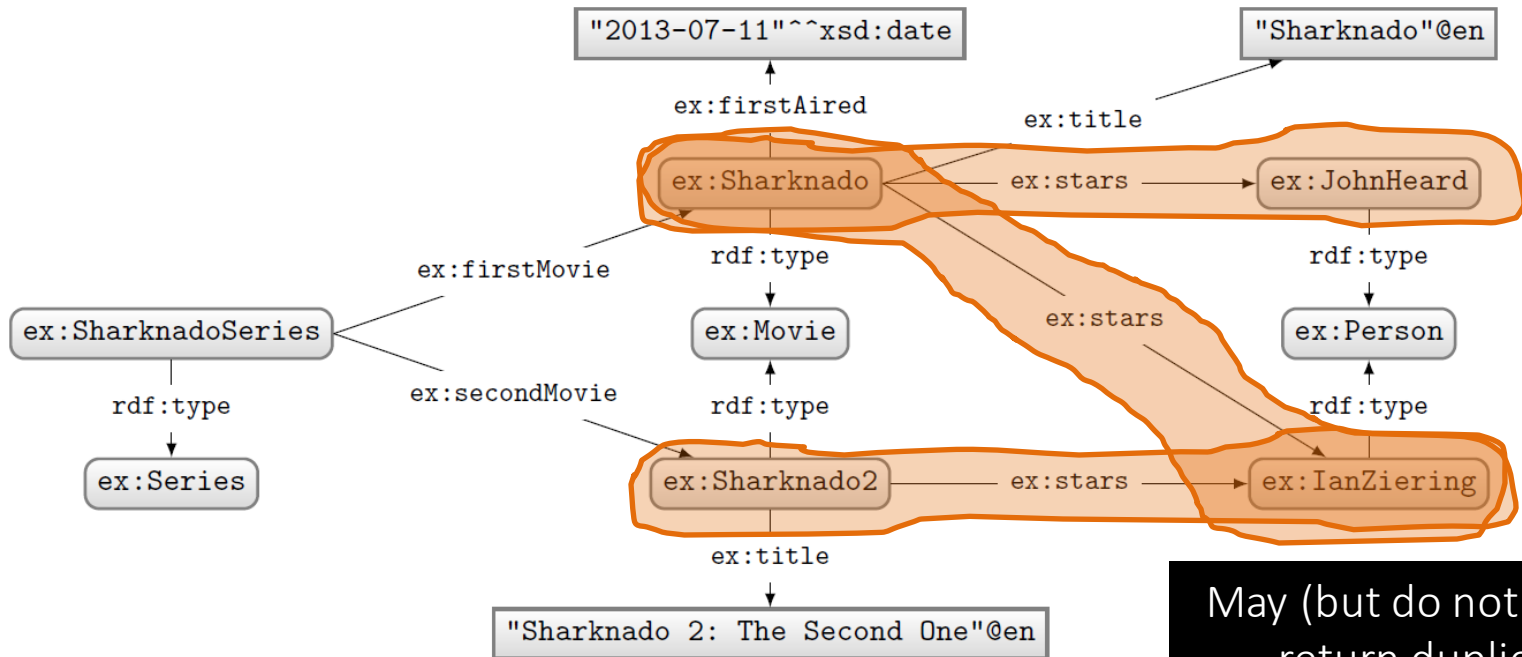
Solutions:

```
?star
ex:JohnHeard
ex:IanZiering
```

DISTINCT: "Set Semantics"

(each result row must be unique)

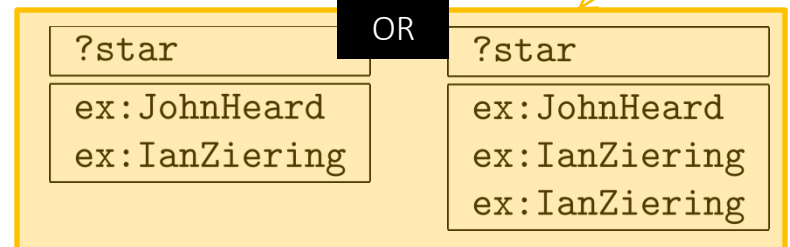
# SPARQL: SELECT WITH REDUCED



Query:

```
PREFIX ex: <http://ex.org/voc#>
SELECT REDUCED ?star
WHERE {
  ?movie a ex:Movie.
  ?movie ex:stars ?star .
}
```

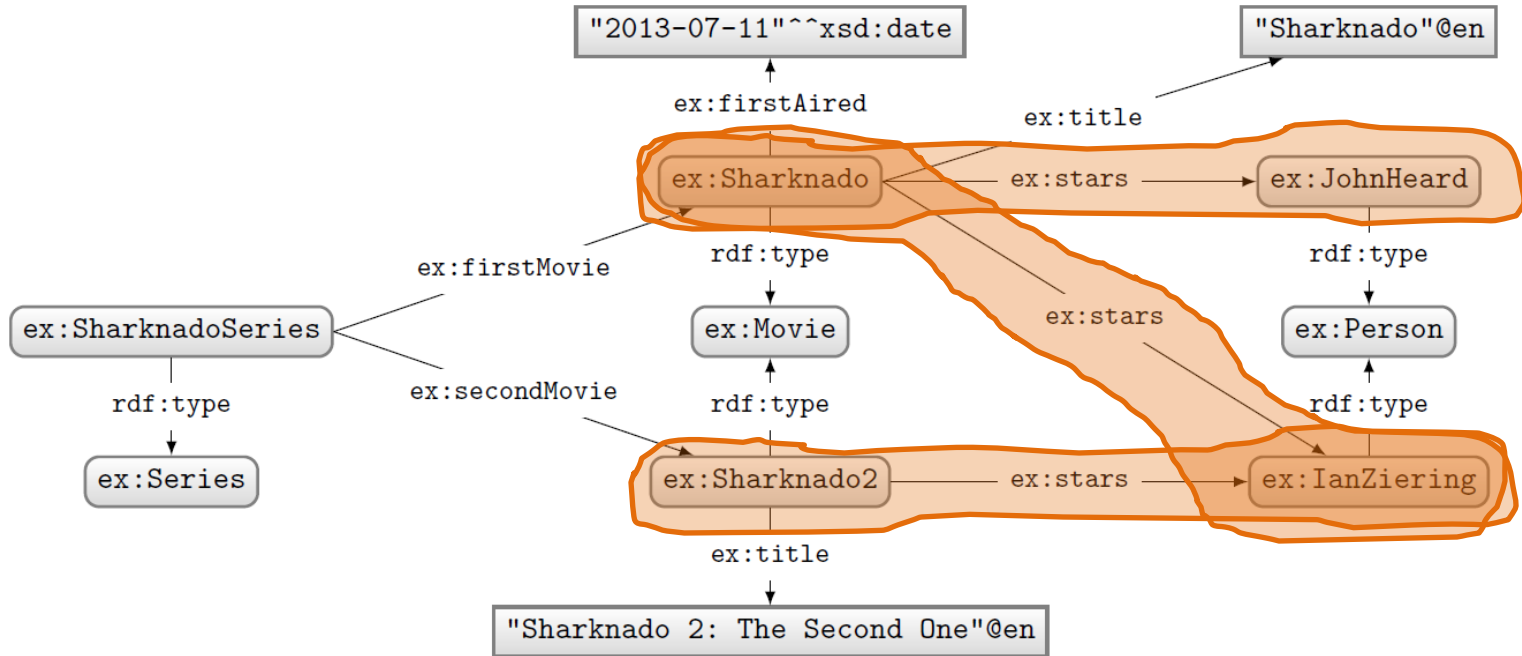
Solutions:



May (but do not need to) return duplicates.

(This allows the engine do whatever is most efficient.)

# SPARQL: ASK



Query:

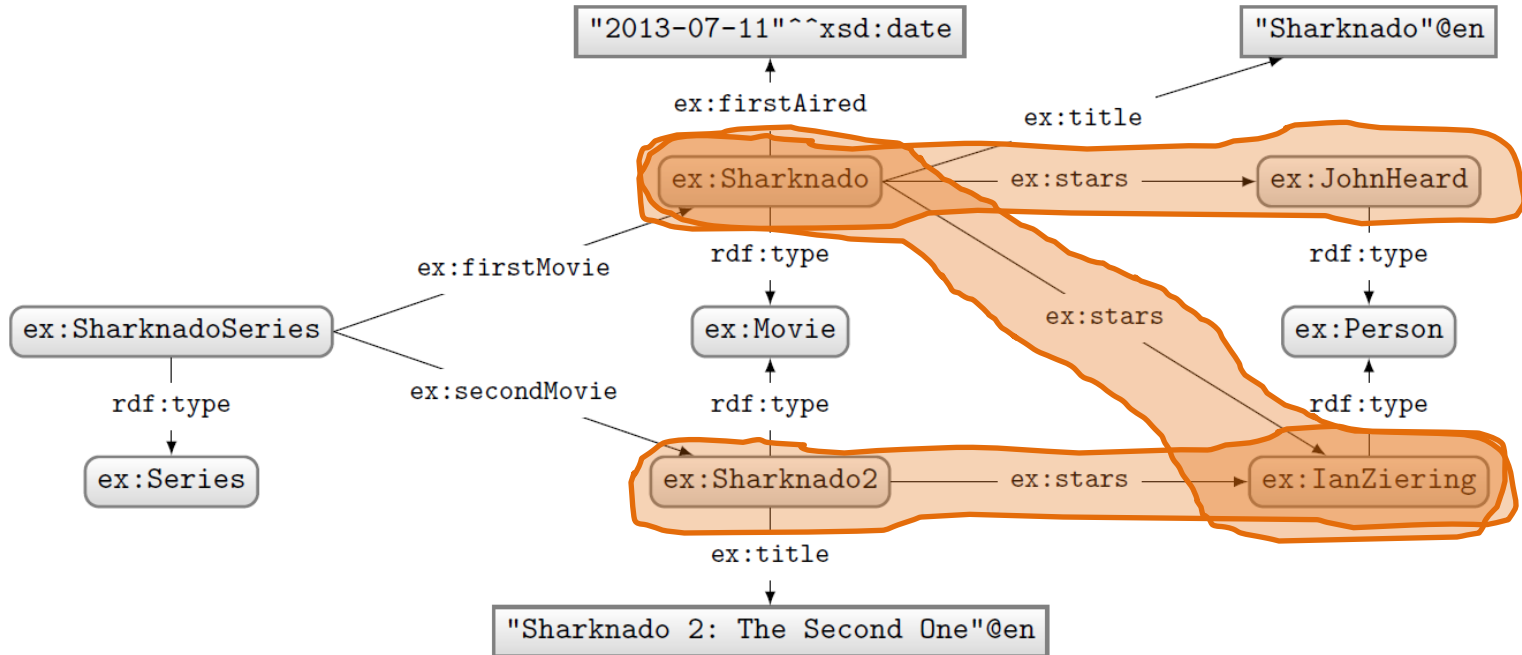
Solutions:

```
PREFIX ex: <http://ex.org/voc#>
ASK
WHERE {
  ?movie a ex:Movie.
  ?movie ex:stars ?star .
}
```

true

Returns true if  
there is a match,  
false otherwise.

# SPARQL: CONSTRUCT



Query:

```
PREFIX ex: <http://ex.org/voc#>
CONSTRUCT { ?star ex:job ex:Actor }
WHERE {
  ?movie a ex:Movie.
  ?movie ex:stars ?star .
}
```

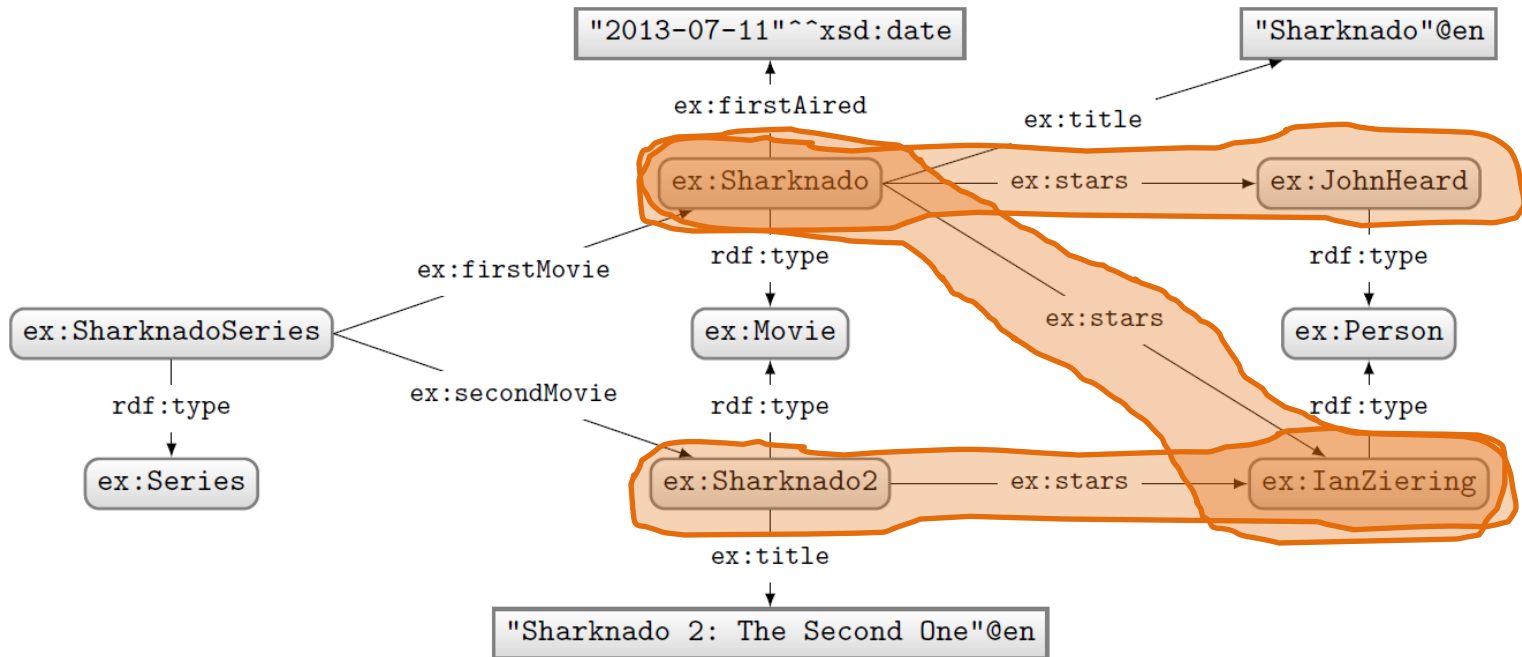
Solutions:

```
@prefix ex: <http://ex.org/voc#> .
ex:JohnHeard ex:job ex:Actor .
ex:IanZiering ex:job ex:Actor .
```

Returns an RDF graph based on the matching CONSTRUCT clause.



# SPARQL: DESCRIBE (NON-NORMATIVE FEATURE)



Query:

```
PREFIX ex: <http://ex.org/voc#>
DESCRIBE ?star
WHERE {
  ?movie a ex:Movie.
  ?movie ex:stars ?star .
}
```

Solutions:

```
@prefix ex: <http://ex.org/voc#> .
ex:JohnHeard a ex:Person .
ex:IanZiering a ex:Person .
```

Returns an RDF graph “describing” the returned results. This is a non-normative feature. What should be returned is left open.

# SPARQL: SOLUTION MODIFIERS

# SOLUTION MODIFIERS

- **ORDER BY (DESC)**
  - Can be used to order results
  - By default ascending (**ASC**), can specify descending (**DESC**)
  - Can order lexicographically on multiple items
- **LIMIT *n***
  - Return only *n* results
- **OFFSET *n***
  - Skip the first *n* results

Without **ORDER BY** results for queries with **LIMIT** or **OFFSET** might be non-deterministic!

How might we ask for the second and third most recently released movies?

```
PREFIX ex: <http://ex.org/voc#>
SELECT ?movie
WHERE { ?movie ex:firstAired ?date . }
ORDER BY DESC(?date)
LIMIT 2
OFFSET 1
```

# SOLUTION MODIFIERS

The order of execution is always:  
ORDER → OFFSET → LIMIT  
Changing the order of LIMIT/OFFSET makes no  
difference to the query solutions.

ORDER BY **must come before** LIMIT/OFFSET  
according to the query syntax

How might we ask for the second and third most recently released movies?

```
PREFIX ex: <http://ex.org/voc#>
SELECT ?movie
WHERE { ?movie ex:firstAired ?date . }
ORDER BY DESC(?date)
LIMIT 2
OFFSET 1
```

≡

```
PREFIX ex: <http://ex.org/voc#>
SELECT ?movie
WHERE { ?movie ex:firstAired ?date . }
ORDER BY DESC(?date)
OFFSET 1
LIMIT 2
```

# SPARQL: NAMED GRAPHS

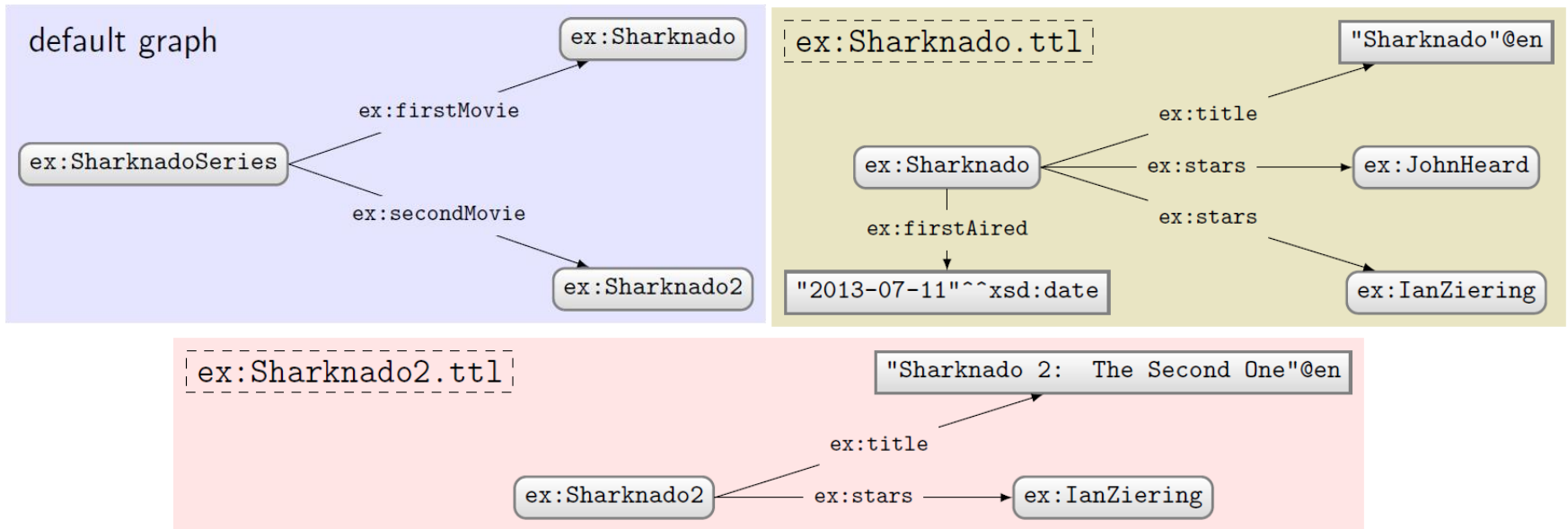
# SPARQL: NAMED GRAPHS

# SPARQL DEFINED OVER A DATASET

- A dataset  $D = (G, \{(G_1, n_1), \dots, (G_k, n_k)\})$
- $G, G_1, \dots, G_k$  are RDF graphs
- $n_1, \dots, n_k$  are pairwise distinct IRIs
- $G$  is called the **default graph**
- each  $(G_i, n_i)$  is a **named graph** ( $1 \leq i \leq n$ )

**Core idea:** SPARQL can support multiple RDF graphs, not just one. When using SPARQL, you can partition your data into multiple graphs. The default graph is chosen if you don't specify a graph. Otherwise you can explicitly select a named graph using its IRI name.

# AN EXAMPLE DATASET

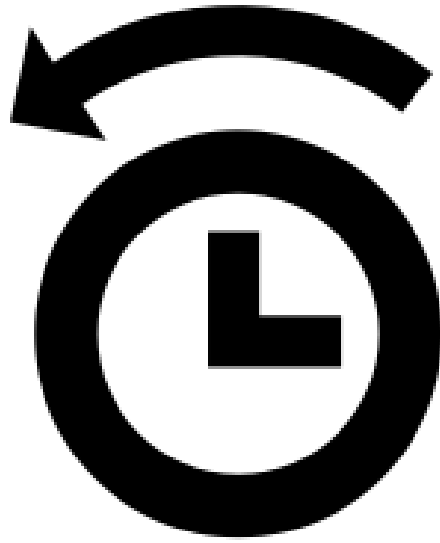




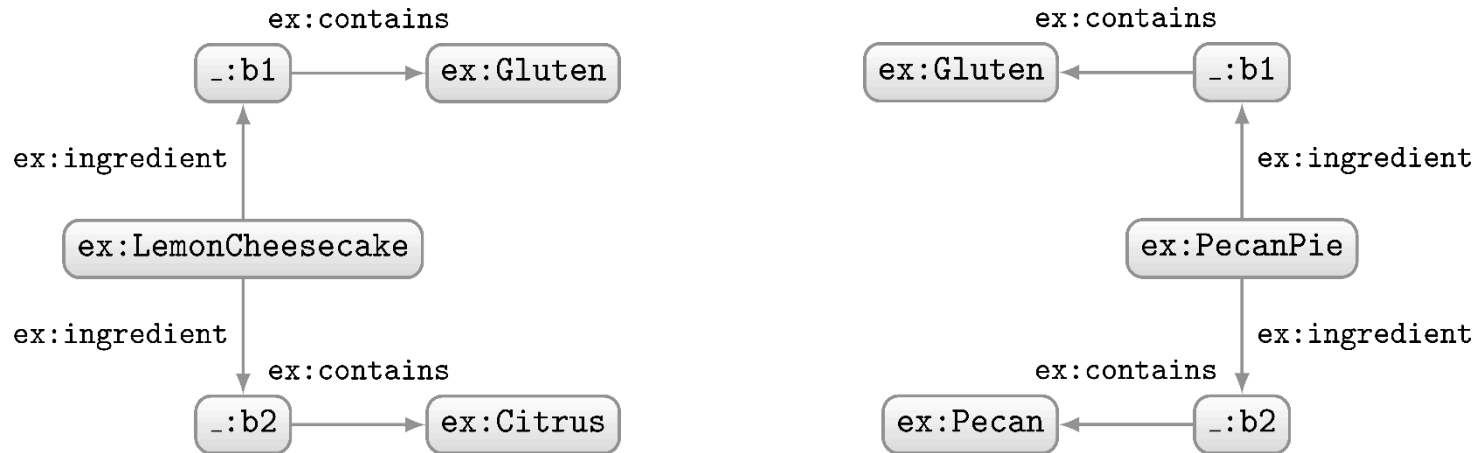
# CREATING A DATASET FOR A QUERY

- Say we have dataset  $D = (G, \{(G_1, n_1), \dots, (G_k, n_k)\})$
- A query can pick an active dataset from the named graphs
- **FROM**
  - Used to define a default graph for the query using graph names
  - If multiple graphs are specified, they are RDF-merged
- **FROM NAMED**
  - Used to select the active named graphs to be used for the query

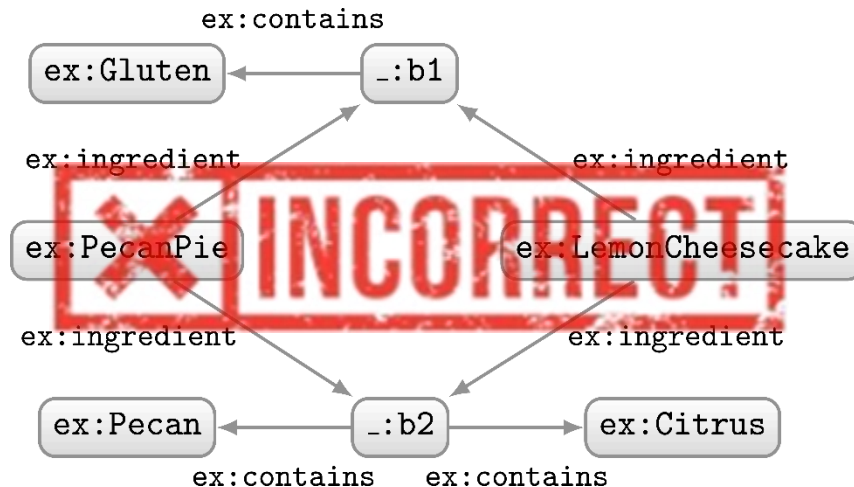
Using either feature clears the index dataset



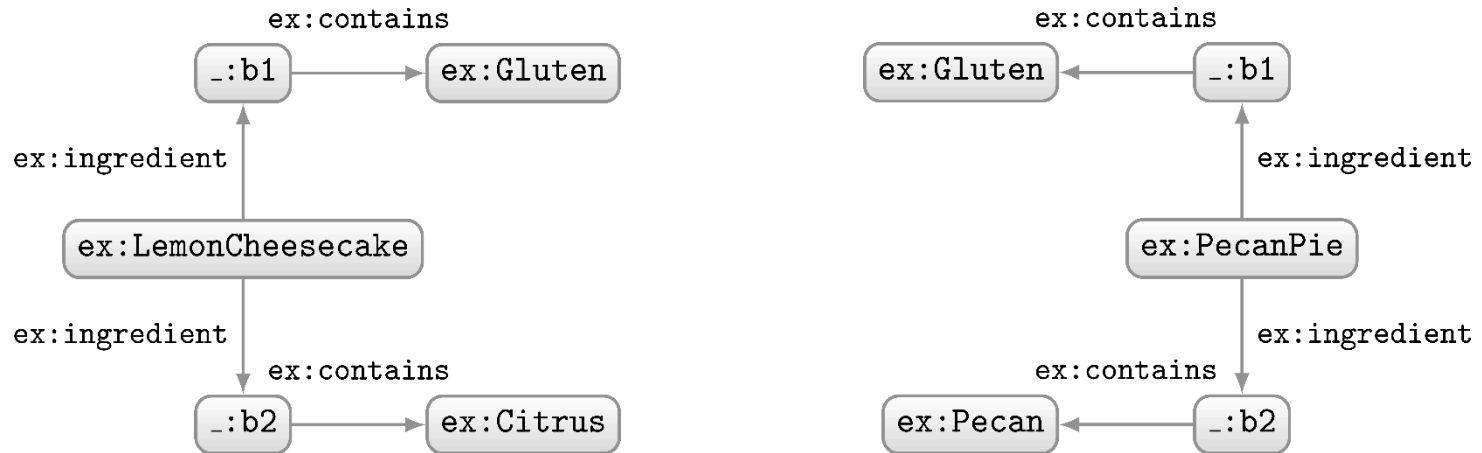
# RDF MERGE: A QUICK REMINDER



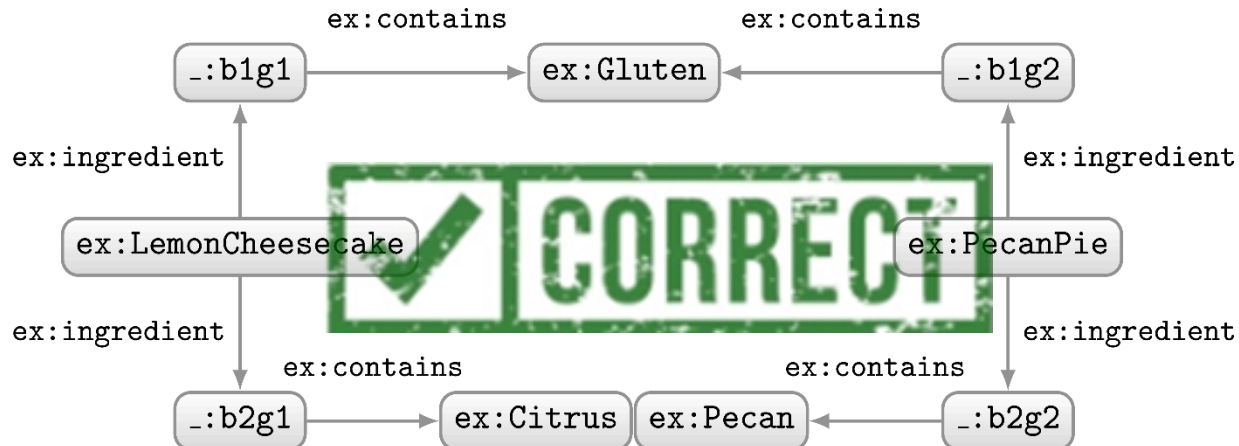
How should we combine these two RDF graphs?



# RDF MERGE: A QUICK REMINDER



How should we combine these two RDF graphs?





# CREATING A DATASET FOR A QUERY

- Indexed dataset:  $D = \{G, (G_1, n_1), \dots, (G_k, n_k)\}$
- Query dataset (no **FROM/FROM NAMED**):  $D$
- Query dataset  $D'$  (example 1):

FROM  $n_1$   
FROM  $n_2$   
FROM NAMED  $n_3$   
FROM NAMED  $n_4$   $\rightarrow D' = \{G_1 \uplus G_2, (G_3, n_3), (G_4, n_4)\}$

- Query dataset  $D'$  (example 2):

FROM  $n_1$   
FROM  $n_2$   $\rightarrow D' = \{G_1 \uplus G_2\}$

( $\uplus$  indicates RDF merge)

# CREATING A DATASET FOR A QUERY

- Indexed dataset:  $D = (G, \{(G_1, n_1), \dots, (G_k, n_k)\})$
- Query dataset (no **FROM**/**FROM NAMED**):  $D$
- Query dataset  $D'$  (example 1):

**FROM**  $n_1$

**FROM**  $n_2$

**FROM NAMED**  $n_3$

**FROM NAMED**  $n_4$

$\rightarrow D' = (G_1 \uplus G_2, \{(G_3, n_3), (G_4, n_4)\})$

- Query dataset  $D'$  (example 2):

**FROM**  $n_1$

**FROM**  $n_2$

$\rightarrow D' = \{G_1 \uplus G_2\}$

( $\uplus$  indicates RDF merge)

# CREATING A DATASET FOR A QUERY

- Indexed dataset:  $D = (G, \{(G_1, n_1), \dots, (G_k, n_k)\})$
- Query dataset (no **FROM**/**FROM NAMED**):  $D$
- Query dataset  $D'$  (example 1):

**FROM**  $n_1$

**FROM**  $n_2$

**FROM NAMED**  $n_3$

**FROM NAMED**  $n_4$

$\rightarrow D' = (G_1 \uplus G_2, \{(G_3, n_3), (G_4, n_4)\})$

- Query dataset  $D'$  (example 2):

**FROM**  $n_1$

**FROM**  $n_2$

$\rightarrow D' = (G_1 \uplus G_2, \{\})$

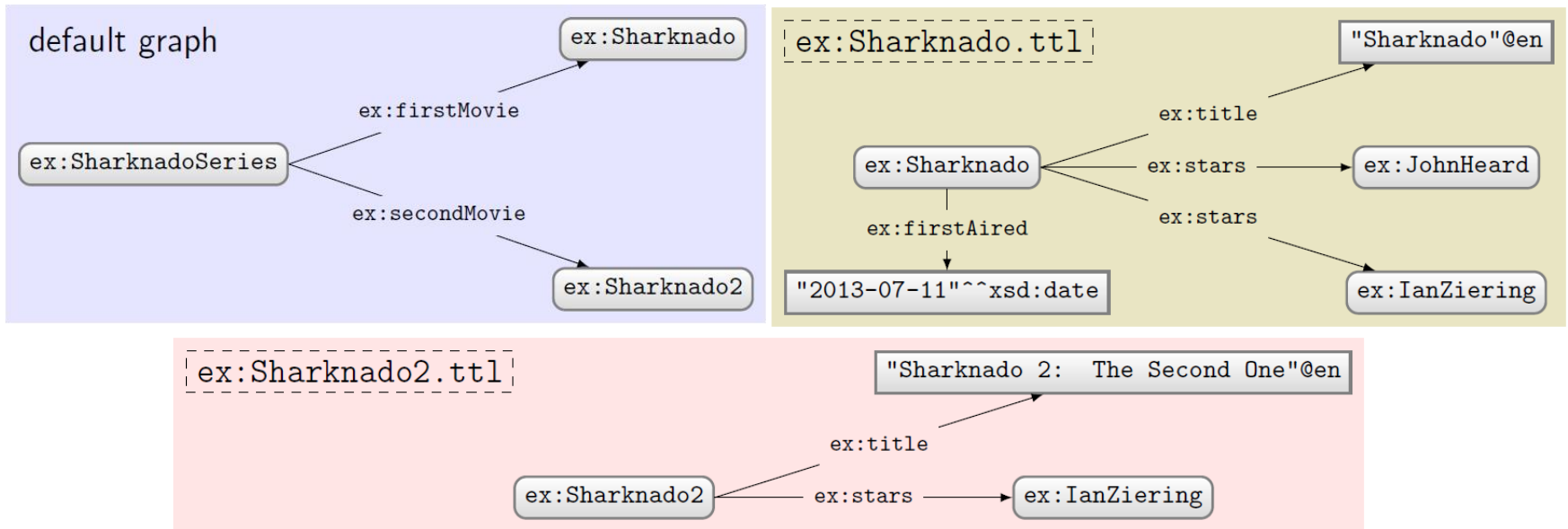
( $\uplus$  indicates RDF merge)



## QUERYING THE NAMED GRAPHS IN A DATASET

- We can query parts of the dataset using **GRAPH**
  - Specifies the IRI of a named graph over which the pattern is evaluated
  - Can also be a variable that ranges over all named graphs
  - Does not access the default graph!
  - If not specified, default graph is accessed

# AN EXAMPLE QUERY



Query:

```
PREFIX ex: <http://ex.org/voc#>  
SELECT DISTINCT ?s  
WHERE { ?s ?p ?o }
```

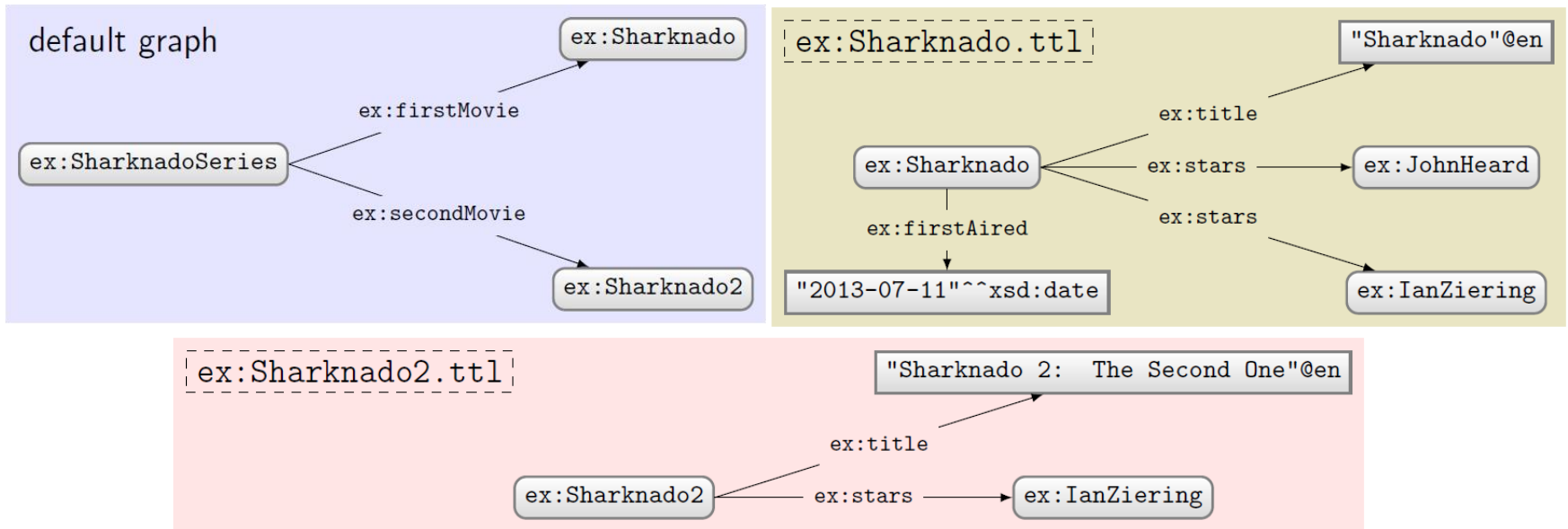
What solutions would this query return?

Solutions:

- `?s`
- `ex:SharknadoSeries`

No GRAPH clause so answers come from default graph only

# USING FROM



Query:

```
PREFIX ex: <http://ex.org/voc#>
FROM ex:Sharknado.ttl
FROM ex:Sharknado2.ttl
SELECT DISTINCT ?s
WHERE { ?s ?p ?o }
```

No GRAPH clause so answers come from default graph defined by FROM (old default graph cleared)

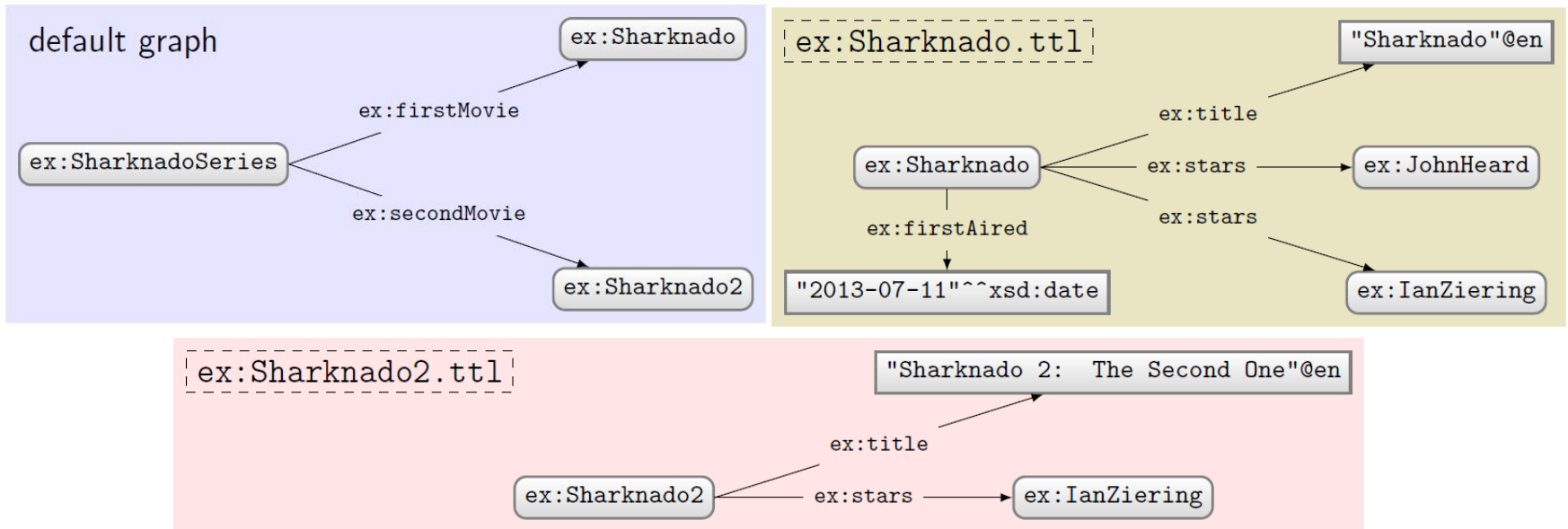
What solutions would this query return?

Solutions:

?s

ex:Sharknado  
ex:Sharknado2

# USING FROM NAMED



Query:

```
PREFIX ex: <http://ex.org/voc#>  
FROM NAMED ex:Sharknado.ttl  
SELECT DISTINCT ?s  
WHERE { ?s ?p ?o }
```

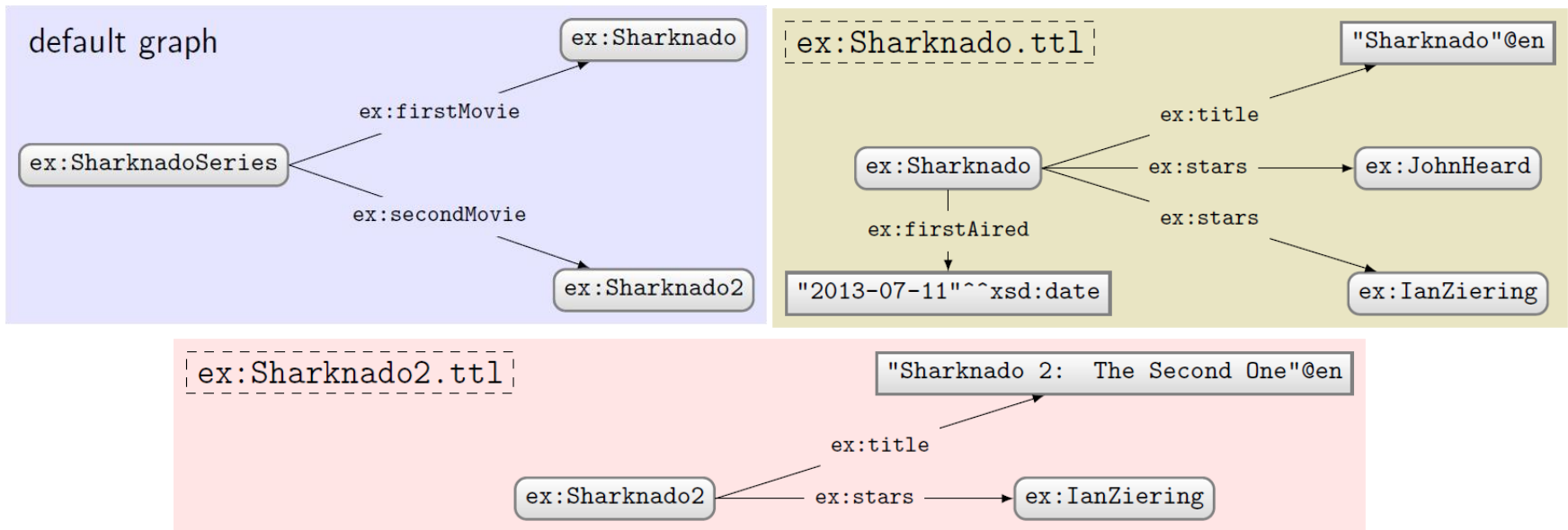
What solutions would this query return?

Solutions:

?s

No GRAPH clause so answers come from default graph, which is empty (since old default graph cleared)!

# USING GRAPH WITH A VARIABLE



Query:

```
PREFIX ex: <http://ex.org/voc#>  
SELECT DISTINCT ?s ?g  
WHERE { GRAPH ?g { ?s ?p ?o } }
```

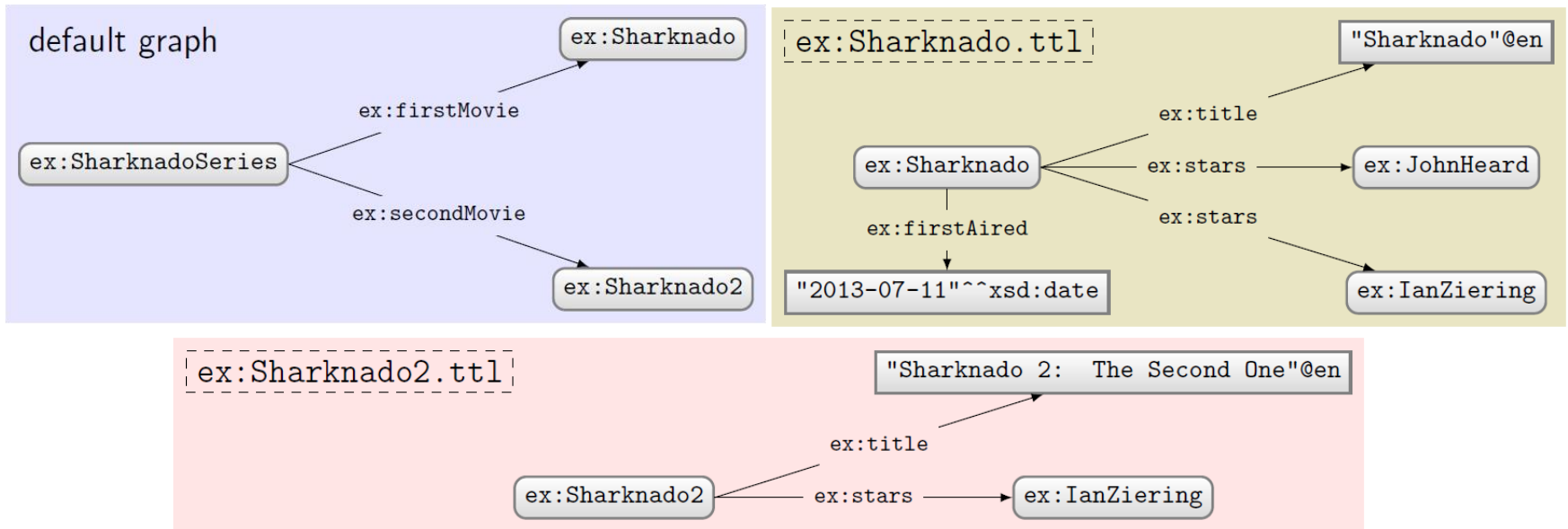
What solutions would this query return?

Solutions:

?s	?g
ex:Sharknado	ex:Sharknado.ttl
ex:Sharknado2	ex:Sharknado2.ttl

GRAPH clause only ranges over the named graphs.

# USING GRAPH WITH A NAME



Query:

```
PREFIX ex: <http://ex.org/voc#>
SELECT DISTINCT ?s
WHERE {
  GRAPH ex:Sharknado.ttl { ?s ?p ?o }
}
```

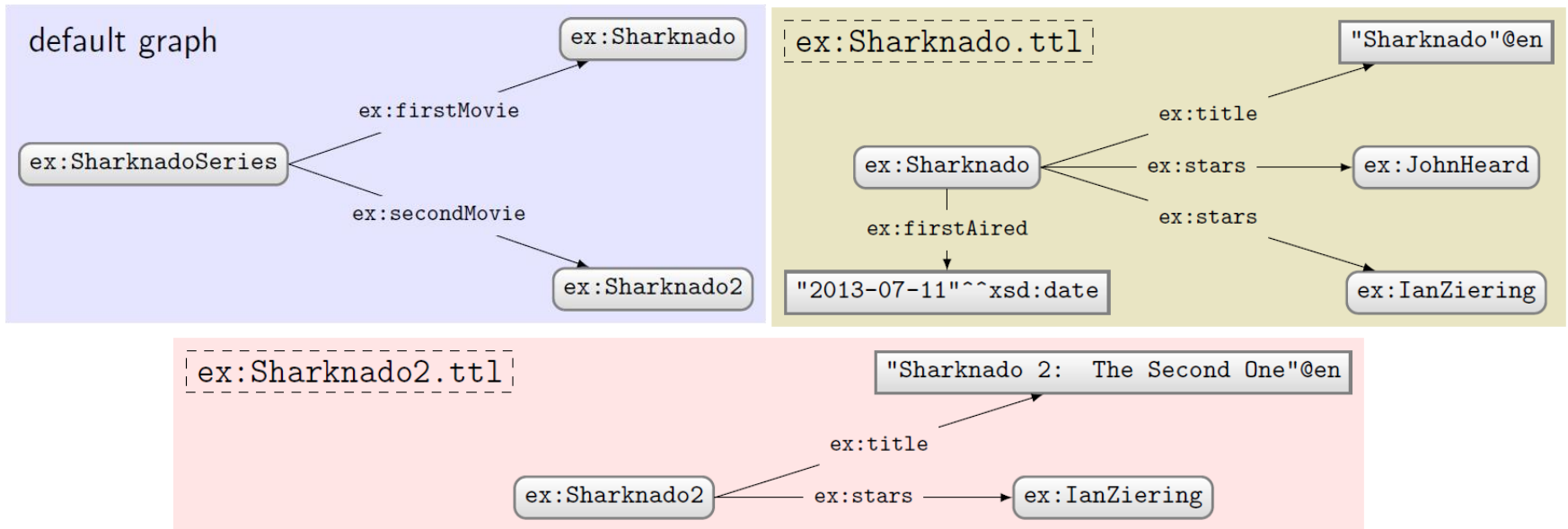
What solutions would this query return?

Solutions:

`?s`

`ex:Sharknado`

# USING GRAPH WITH FROM



Query:

```
PREFIX ex: <http://ex.org/voc#>  
FROM ex:Sharknado.ttl  
SELECT DISTINCT ?s ?g  
WHERE {  
  GRAPH ?g { ?s ?p ?o }  
}
```

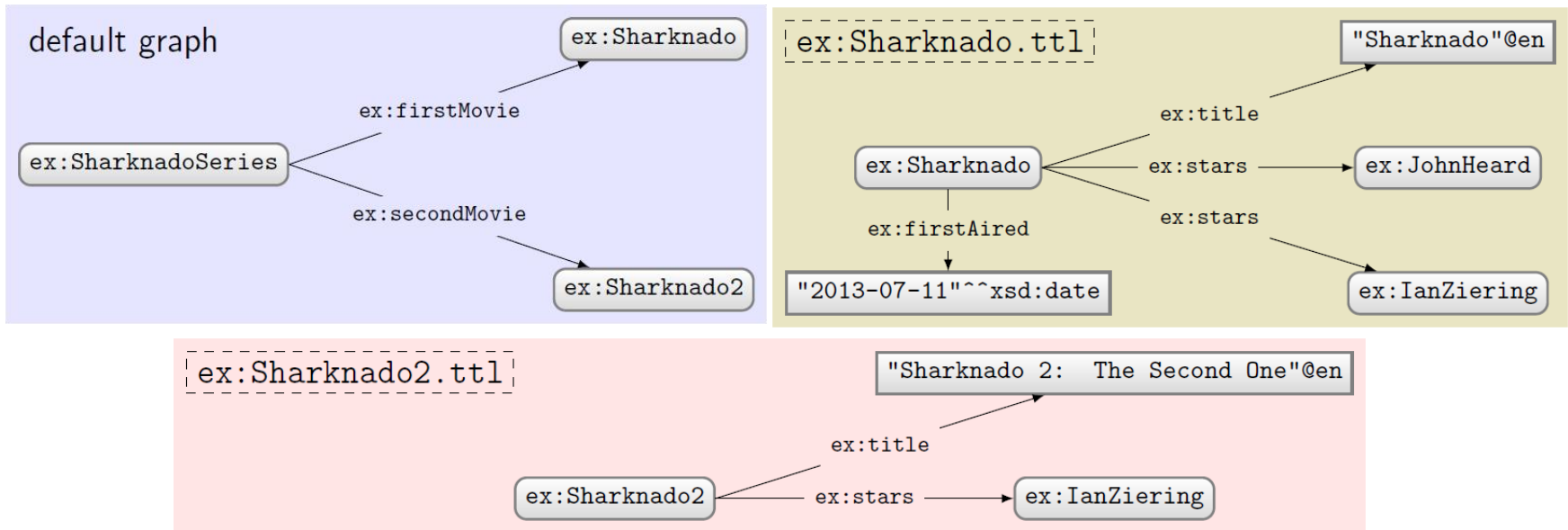
What solutions would this query return?

Solutions:

?s	?g
----	----

No named graphs specified!

# USING GRAPH WITH FROM NAMED



Query:

```
PREFIX ex: <http://ex.org/voc#>  
FROM NAMED ex:Sharknado.ttl  
SELECT DISTINCT ?s ?g  
WHERE {  
  GRAPH ?g { ?s ?p ?o }  
}
```

What solutions would this query return?

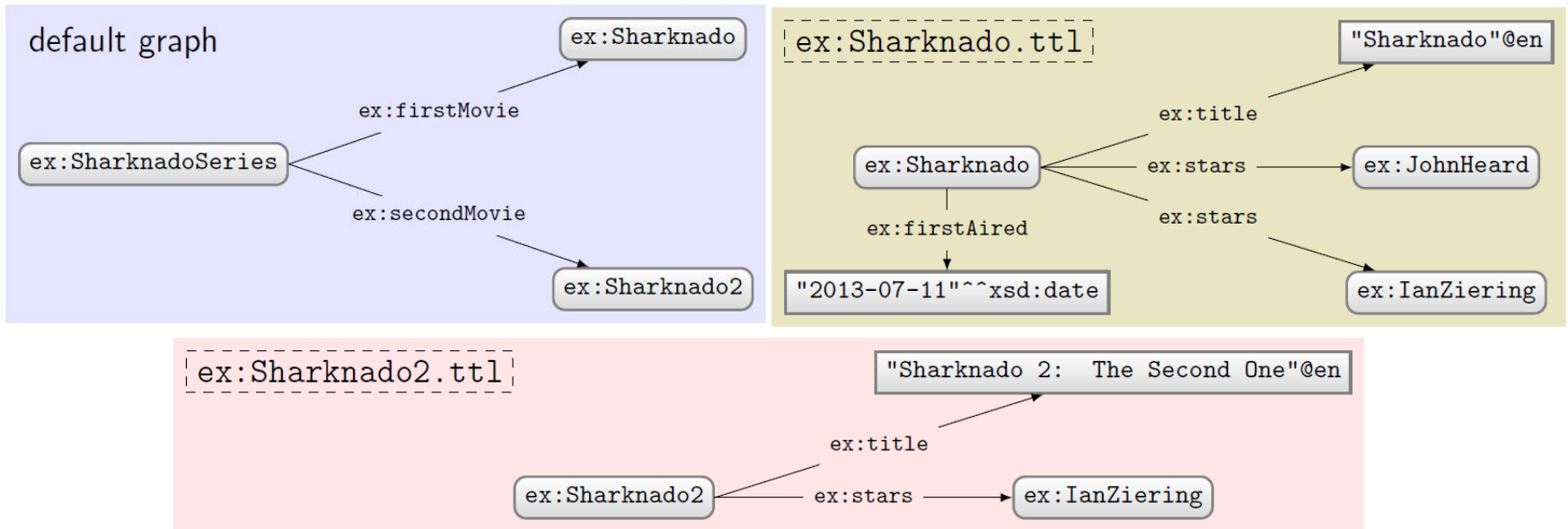
Solutions:

?s	?g
ex:Sharknado	ex:Sharknado.ttl

GRAPH accesses the one and only named graph



# USING GRAPH WITH FROM AND FROM NAMED



Query:

```
PREFIX ex: <http://ex.org/voc#>
FROM ex:Sharknado2.ttl
FROM NAMED ex:Sharknado.ttl
SELECT DISTINCT ?x ?q
WHERE {
  GRAPH ?g { ?s ?p ?o }
  ?x ?q ?o .
}
```

What solutions would this query return?

Solutions:

?x	?q
ex:Sharknado2	ex:stars



First SPARQL (1.0)  
Then SPARQL 1.1

QUESTIONS?

