CSC 481: SPARQL - Querying the Semantic Web

Based on Chapter 5 of “Semantic Web for the Working Ontologist - Modeling in RDF, RDFS and OWL” by Allenmag and Hendler

Rodrigo Canaan
Assistant Professor
Computer Science Department
Cal Poly, San Luis Obispo
rcanaan@calpoly.edu

Note: book is available at Cal Poly Library

Note: book is available at Cal Poly Library

- https://csu-calpoly.primo.exlibrisgroup.com/permalink/01CALS_PSU/1qh1nk7/alma9910676171478029O1

Note: book is available at Cal Poly Library

- https://csu-calpoly.primo.exlibrisgroup.com/permalink/01CALS_PSU/1qh1nk7/alma9910676171478029O1

Note: book is available at Cal Poly Library

- https://csu-calpoly.primo.exlibrisgroup.com/permalink/01CALS_PSU/1qh1nk7/alma9910676171478029O1

- https://ebookcentral.proquest.com/lib/calpoly/detail.action?docID=689813&pq-origsite=primo#

Query basics

Query basics

- Consider the following statement: “James Dean played in the movie *Giant*”

Query basics

- Consider the following statement: “James Dean played in the movie *Giant*”
- What are some questions we might want to ask about this statement?

Query basics

- Consider the following statement: “James Dean played in the movie *Giant*”
- What are some questions we might want to ask about this statement?
  - **True/false** questions: Did James Dean play in *Giant*?

Query basics

- Consider the following statement: “James Dean played in the movie *Giant*”
- What are some questions we might want to ask about this statement?
  - **True/false** questions: Did James Dean play in *Giant*?
  - Questions about the **subject**: who played in *Giant*?

Query basics

- Consider the following statement: “James Dean played in the movie *Giant*”
- What are some questions we might want to ask about this statement?
  - **True/false** questions: Did James Dean play in *Giant*?
  - Questions about the **subject**: who played in *Giant*?
  - Questions about the **object:** what movie did James Dean play?

Query basics

- Consider the following statement: “James Dean played in the movie *Giant*”
- What are some questions we might want to ask about this statement?
  - **True/false** questions: Did James Dean play in *Giant*?
  - Questions about the **subject**: who played in *Giant*?
  - Questions about the **object:** what movie did James Dean play?
  - Questions about the **predicate:** what did James Dean do in *Giant*?

SPARQL

SPARQL

- SPARQL = **S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage

SPARQL

- SPARQL = **S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage
  - Yes, the S stands for SPARQL

SPARQL

- SPARQL = **S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage
  - Yes, the S stands for SPARQL
- A language for querying RDF data

SPARQL

- SPARQL = **S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage
  - Yes, the S stands for SPARQL
- A language for querying RDF data
  - Defines *what* a query should return, different implementations (engines) may implement the *how* differently

SPARQL

- SPARQL = **S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage
  - Yes, the S stands for SPARQL
- A language for querying RDF data
  - Defines *what* a query should return, different implementations (engines) may implement the *how* differently
- Uses similar syntax as SQL

SPARQL

- SPARQL = **S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage
  - Yes, the S stands for SPARQL
- A language for querying RDF data
  - Defines *what* a query should return, different implementations (engines) may implement the *how* differently
- Uses similar syntax as SQL
- Full specification: https://www.w3.org/TR/rdf-sparql-query/#select

SPARQL

- SPARQL = **S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage
  - Yes, the S stands for SPARQL
- A language for querying RDF data
  - Defines *what* a query should return, different implementations (engines) may implement the *how* differently
- Uses similar syntax as SQL
- Full specification: https://www.w3.org/TR/rdf-sparql-query/#select
- See https://www.w3.org/wiki/SparqlImplementations for various implementations.

SPARQL Query Forms

SPARQL Query Forms

SPARQL allows 3 main types (“forms”) of query

SPARQL Query Forms

SPARQL allows 3 main types (“forms”) of query

- **ASK** queries: returns whether a pattern exists in the data

SPARQL Query Forms

SPARQL allows 3 main types (“forms”) of query

- **ASK** queries: returns whether a pattern exists in the data
- **SELECT:** returns variables and their bindings that satisfy a pattern

SPARQL Query Forms

SPARQL allows 3 main types (“forms”) of query

- **ASK** queries: returns whether a pattern exists in the data
- **SELECT:** returns variables and their bindings that satisfy a pattern
- **CONSTRUCT:** returns an RDF graph specified by the query template

A simple SELECT example

A simple SELECT example

Given the RDF triple :James_Dean :played_In :Giant

A simple SELECT example

Given the RDF triple :James_Dean :played_In :Giant

A simple SELECT example

Given the RDF triple :James_Dean :played_In :Giant

- **SELECT** ?actor **WHERE** ?actor :played_In :Giant

A simple SELECT example

Given the RDF triple :James_Dean :played_In :Giant

- **SELECT** ?actor **WHERE** ?actor :played_In :Giant
  - Returns :James_Dean

A simple SELECT example

Given the RDF triple :James_Dean :played_In :Giant

- **SELECT** ?actor **WHERE** ?actor :played_In :Giant
  - Returns :James_Dean
- **SELECT** ?movie **WHERE** :James_Dean :played_In ?movie

A simple SELECT example

Given the RDF triple :James_Dean :played_In :Giant

- **SELECT** ?actor **WHERE** ?actor :played_In :Giant
  - Returns :James_Dean
- **SELECT** ?movie **WHERE** :James_Dean :played_In ?movie
  - Returns :Giant

A simple SELECT example

Given the RDF triple :James_Dean :played_In :Giant

- **SELECT** ?actor **WHERE** ?actor :played_In :Giant
  - Returns :James_Dean
- **SELECT** ?movie **WHERE** :James_Dean :played_In ?movie
  - Returns :Giant
- **SELECT** ?role **WHERE** :James_Dean ?role :Giant

A simple SELECT example

Given the RDF triple :James_Dean :played_In :Giant

- **SELECT** ?actor **WHERE** ?actor :played_In :Giant
  - Returns :James_Dean
- **SELECT** ?movie **WHERE** :James_Dean :played_In ?movie
  - Returns :Giant
- **SELECT** ?role **WHERE** :James_Dean ?role :Giant
  - Returns :played_In

A simple SELECT example

Given the RDF triple :James_Dean :played_In :Giant

- **SELECT** ?actor **WHERE** ?actor :played_In :Giant
  - Returns :James_Dean
- **SELECT** ?movie **WHERE** :James_Dean :played_In ?movie
  - Returns :James_Dean
- **SELECT** ?role **WHERE** :James_Dean ?role :Giant
  - Returns :played_In

Note: the meaning of the variables **?actor**, **?movie** and **?role** derive from their position in the graph pattern specified by the WHERE clause, not by the meaning of their names in English

Parallel with Prolog

triple (JamesDean, playedIn, Giant)

Given the RDF triple :James_Dean :played_In :Giant

?- triple(X, playedIn, Giant)

- **SELECT** ?actor **WHERE** ?actor :played_In :Giant
  - Returns :James_Dean

?- triple(JamesDean, playedIn, X)

- **SELECT** ?movie **WHERE** :James_Dean :played_In ?movie
  - Returns :James_Dean

?- triple(JamesDean, X, Giant)

- **SELECT** ?role **WHERE** :James_Dean ?role :Giant
  - Returns :played_In

More complex queries and Graph patterns

Consider the graph below:

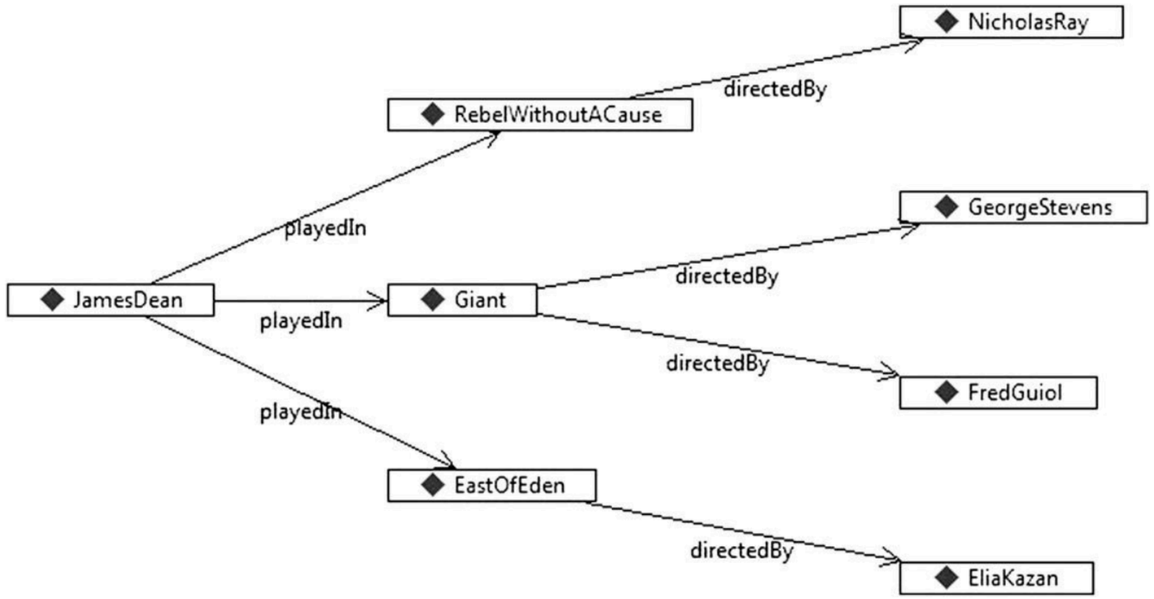


FIGURE 5.2

James Dean's movies and their directors.

More complex queries and Graph patterns

Consider the graph below:

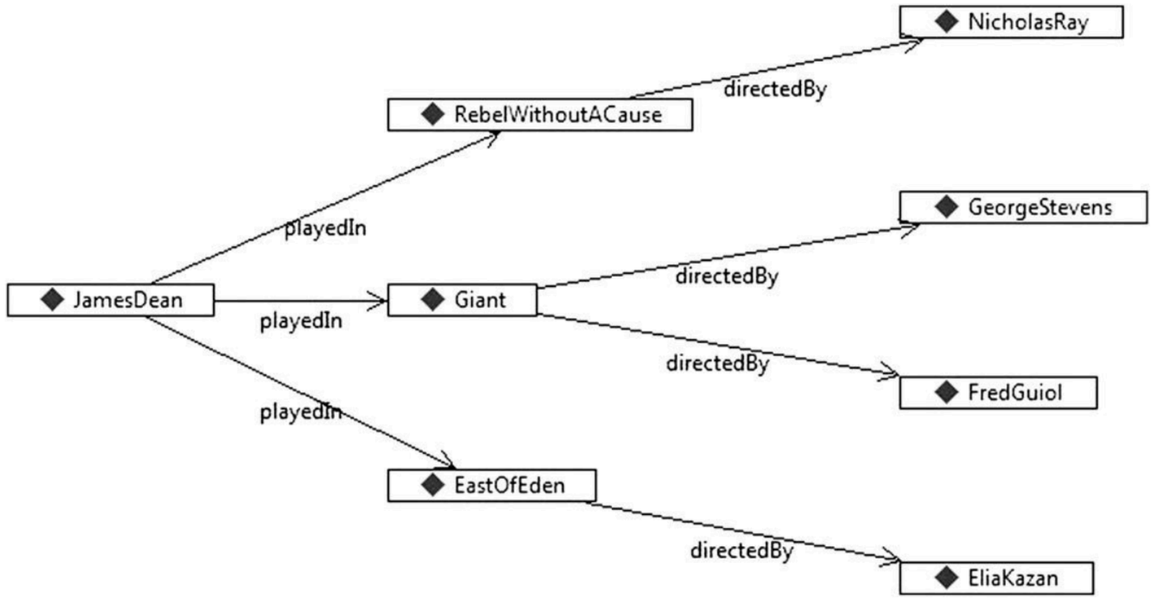


FIGURE 5.2

James Dean's movies and their directors.

A more complex query

```
:JamesDean :playedIn ?what .
?what :directedBy ?who .
```

More complex queries and Graph patterns

Consider the graph below:

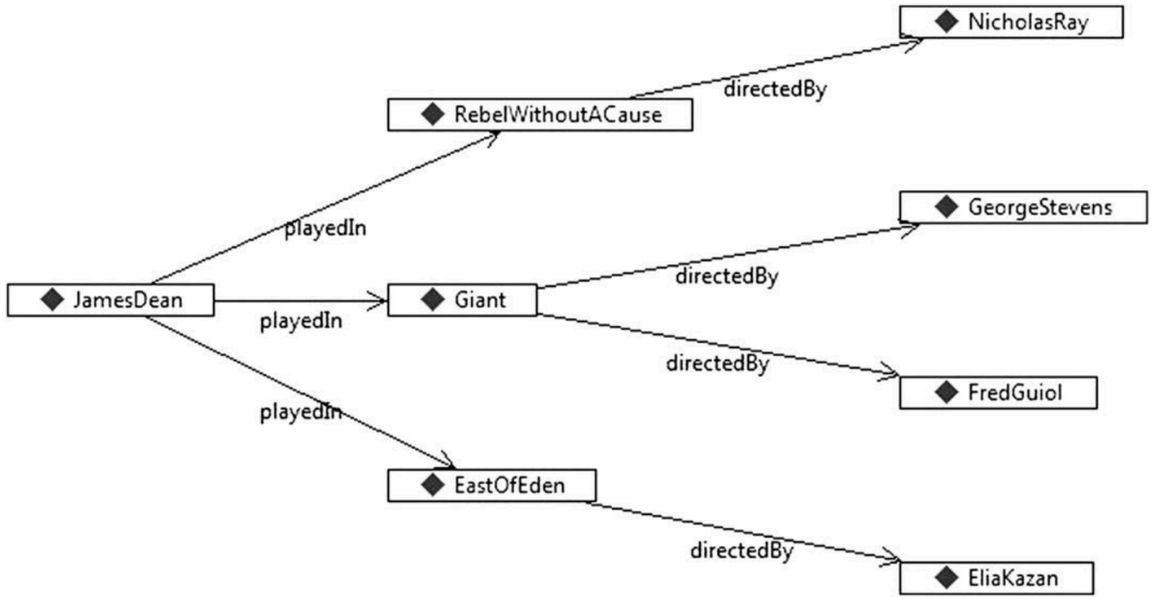


FIGURE 5.2

James Dean's movies and their directors.

A more complex query

```
:JamesDean :playedIn ?what .
?what :directedBy ?who .
```

The corresponding graph pattern

FIGURE 5.3

Graphic version of a query to find James Dean's directors.

More complex queries and Graph patterns

Ask:

```
SELECT ?who
WHERE {:JamesDean :playedIn ?what .
       ?what :directedBy ?who .}
```

Answer:

?who
:GeorgeStevens
:EliaKazan
:NicholasRay
:FredGuiol

Ask:

```
SELECT ?what ?who
WHERE {:JamesDean :playedIn ?what .
       ?what :directedBy ?who .}
```

?what	?who
:Giant	:GeorgeStevens
:Giant	:FredGuiol
:EastOfEden	:EliaKazan
:RebelWithoutaCause	:NicholasRay

More complex queries and Graph patterns

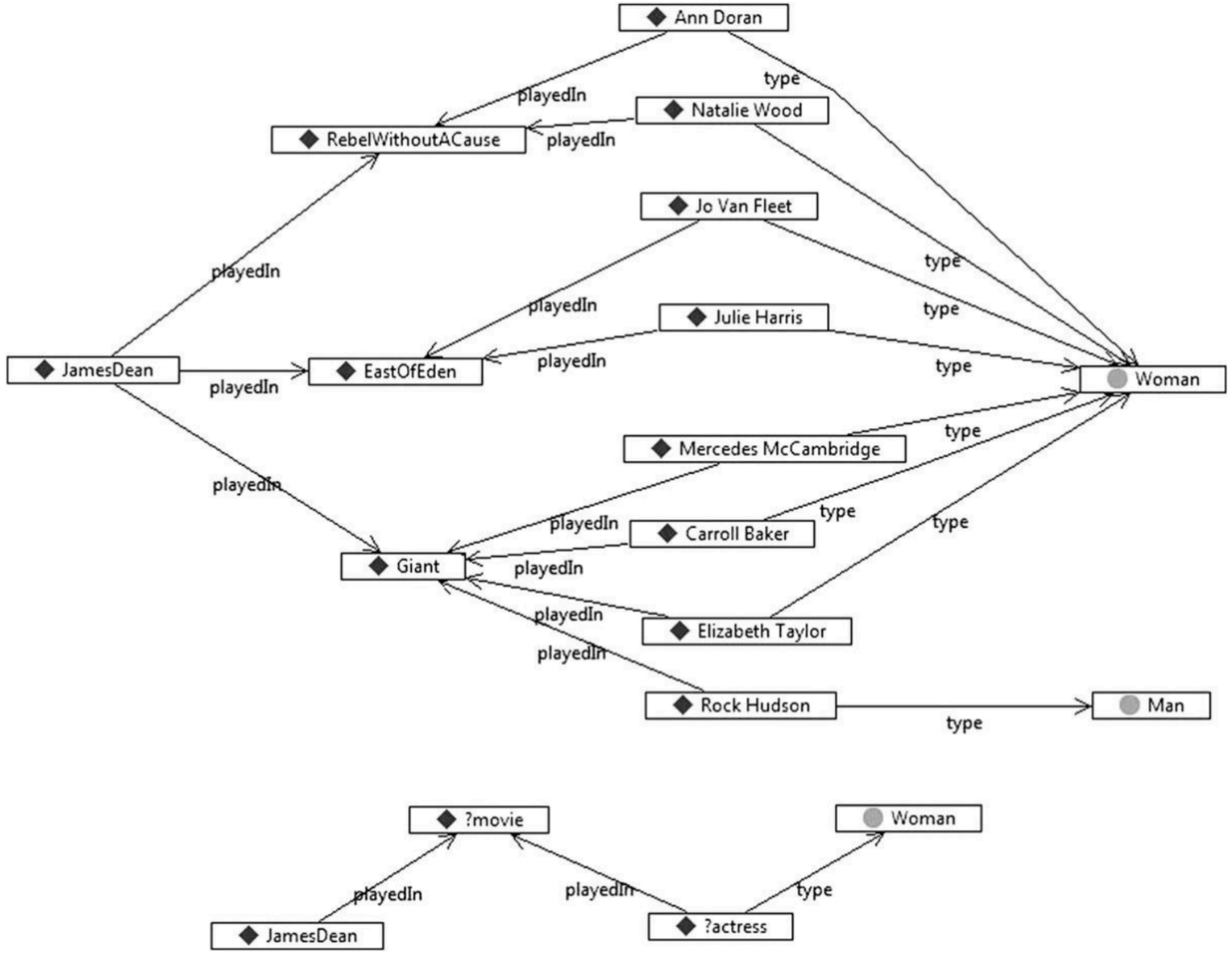
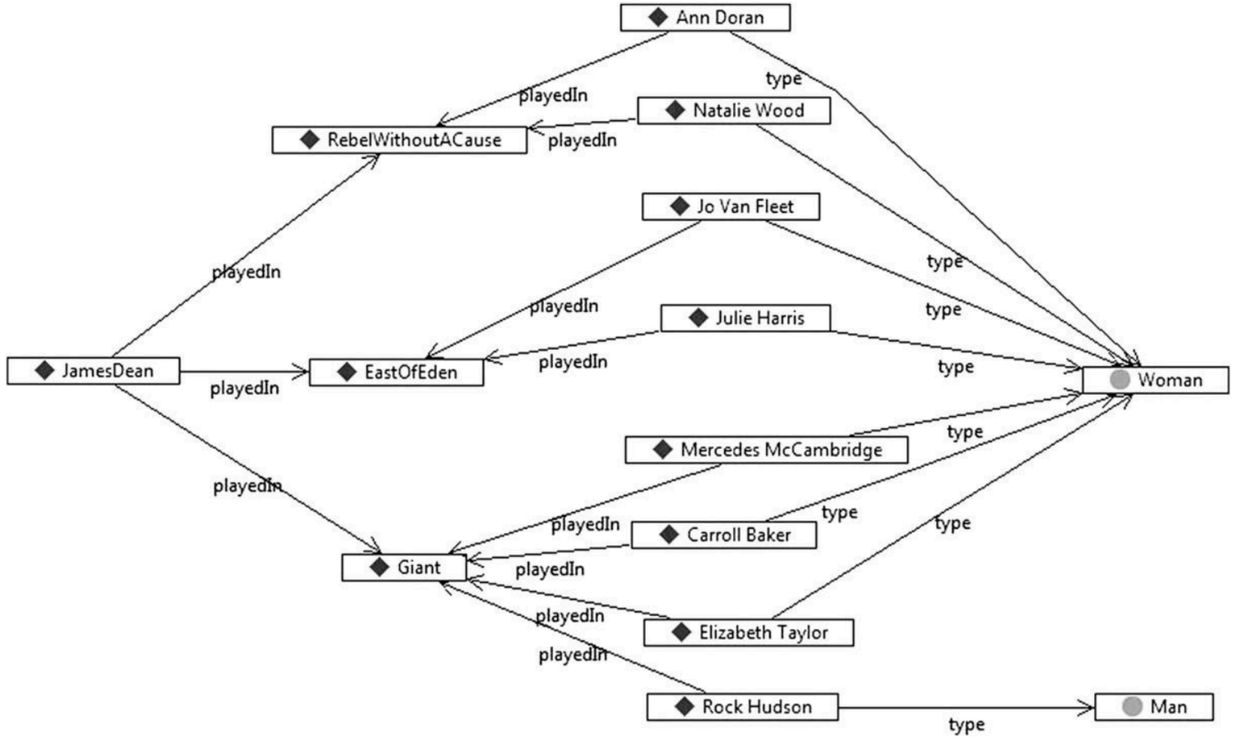


FIGURE 5.4

Information about James Dean's co-stars, and a query to fetch some of it.

More complex queries and Graph patterns



1. How would you write a query to find all the **actresses** and **movies** that match this pattern?
2. What would the result be?

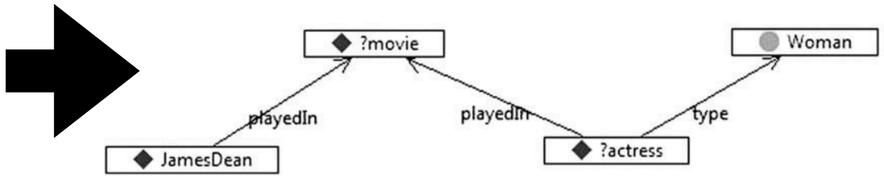


FIGURE 5.4

Information about James Dean's co-stars, and a query to fetch some of it.

More complex queries and Graph patterns

SELECT ?movie ?actress **WHERE**

:James_Dean :played_In ?movie.

?actress :played_In ?movie.

?actress :type :woman.

Querying for properties and schema

Querying for properties and schema

SPARQL lets you ask open-ended queries such as “what do you know about James dean”?

Ask:

```
SELECT ?property ?value
WHERE {:JamesDean ?property ?value}
```

Answer:

?property	?value
bornOn	1931-02-08
diedOn	1955-09-30
playedIn	RebelWithoutaCause
playedIn	EastOfEden
playedIn	Giant
rdf:type	Man
rdfs:label	James Dean

Querying for properties and schema

You can also ask for meta-data: “what are the sorts of things you know about James Dean?”

Ask:

```
SELECT ?property
WHERE {:JamesDean ?property ?value}
```

Answer:

?property
bornOn
diedOn
playedIn
playedIn
playedIn
rdf:type
rdfs:label

Querying for properties and schema

You can also ask for meta-data: “what are the sorts of things you know about James Dean?”

Ask:

```
SELECT ?property
WHERE {:JamesDean ?property ?value}
```

Answer:

?property
bornOn
diedOn
playedIn
playedIn
playedIn
rdf:type
rdfs:label

Ask:

```
SELECT DISTINCT ?property
WHERE {:JamesDean ?property ?value}
```

Answer:

?property
bornOn
diedOn
playedIn
rdf:type
rdfs:label

Querying for properties and schema

You can ask for properties of any actor instead

Ask:

```
SELECT DISTINCT ?property
WHERE {?q0 a :Actor .
       ?q0 ?property ?object .}
```

Answer:

?property
bornOn
diedOn
playedIn
rdf:type
rdfs:label
produced
sang
wrote

Querying for properties and schema

You can also ask for all subclasses of person, or for all classes or properties in general: RDF/SPARQL is “self-describing”

Ask:

```
SELECT DISTINCT ?class
WHERE {?class rdfs:subClassOf :Person}
```

Answer:

?class
:Actor
:Actress
:Man
:Woman
:Politician
:Producer

```
SELECT DISTINCT ?class
WHERE {?q0 a ?class}
```

```
SELECT DISTINCT ?property
WHERE {?q0 ?property ?q1}
```

Bindings and filters

The FILTER keyword

You can use FILTER to define a boolean test for whether or not to include a result

Ask:

```
SELECT ?actor
WHERE {?actor :playedIn :Giant .
       ?actor :diedOn ?deathdate .
       FILTER (?deathdate > "1961-11-24"^^xsd:date)}
```

Operators used in FILTER include arithmetic comparison operators, boolean functions and regex matching.

The FILTER keyword

Be careful with un-instantiated variables!

Ask:

```
SELECT ?actor
WHERE {?actor :playedIn :EastOfEden .
       FILTER (?birthday > "1930-01-01"^^xsd:date)}
```

Answer:
(none)

The FILTER keyword

Be careful with un-instantiated variables!

Ask:

```
SELECT ?actor
WHERE {?actor :playedIn :EastOfEden .
       FILTER (?birthday > "1930-01-01"^^xsd:date)}
```

Answer:
(none)

The FILTER keyword

Be careful with un-instantiated variables!

```
SELECT ?actor
WHERE {?actor :playedIn :EastOfEden .
       FILTER (?q0 > "1930-01-01"^^xsd:date)}
```

Ask:

```
SELECT ?actor
WHERE {?actor :playedIn :EastOfEden .
       FILTER (?birthday > "1930-01-01"^^xsd:date)}
```

Answer:
(none)

The FILTER keyword

Be careful with un-instantiated variables!

```
SELECT ?actor
WHERE {?actor :playedIn :EastOfEden .
       FILTER (?q0 > "1930-01-01"^^xsd:date)}
```

Ask:

```
SELECT ?actor
WHERE {?actor :playedIn :EastOfEden .
       FILTER (?birthday > "1930-01-01"^^xsd:date)}
```

Answer:
(none)

The FILTER keyword

How to write this query correctly?

Ask:

```
SELECT ?actor
WHERE {?actor :playedIn :EastOfEden .
       FILTER (?birthday > "1930-01-01"^^xsd:date)}
```

Answer:
(none)

The FILTER keyword

How to write this query correctly?

Ask:

```
SELECT ?actor
WHERE {?actor :playedIn :EastOfEden .
       FILTER (?birthday > "1930-01-01"^^xsd:date)}
```

Answer:
(none)

```
SELECT ?actor
WHERE {?actor :playedIn :EastOfEden .
       ?actor :bornOn ?birthday .
       FILTER (?birthday > "1930-01-01"^^xsd:date)}
```

Yes/No Queries - the ASK keyword

The ASK keyword

You can use the ASK keyword to ask true/false questions - no variable bindings are returned

The ASK keyword

You can use the ASK keyword to ask true/false questions - no variable bindings are returned

“Did Elizabeth Taylor die”?

```
ASK WHERE {:ElizabethTaylor :diedOn ?any}
```

The ASK keyword

You can use the ASK keyword to ask true/false questions - no variable bindings are returned

"Did Elizabeth Taylor die"?

```
ASK WHERE {:ElizabethTaylor :diedOn ?any}
```

"Is Elizabeth Taylor alive?"

```
ASK WHERE {UNSAID {:ElizabethTaylor :diedOn ?any}}
```

The ASK keyword

You can use the ASK keyword to ask true/false questions - no variable bindings are returned

"Did Elizabeth Taylor die"?

```
ASK WHERE {:ElizabethTaylor :diedOn ?any}
```

"Is Elizabeth Taylor alive?"

```
ASK WHERE {UNSAID {:ElizabethTaylor :diedOn ?any}}
```

Negation as failure: the query succeeds if the subgraph under UNSAID is empty (returns no results)

The ASK keyword

You can use the ASK keyword to ask true/false questions - no variable bindings are returned

"Did Elizabeth Taylor die"?

```
ASK WHERE {:ElizabethTaylor :diedOn ?any}
```

"Is Elizabeth Taylor alive?"

```
ASK WHERE {UNSAID {:ElizabethTaylor :diedOn ?any}}
```

Negation as failure: the query succeeds if the subgraph under UNSAID is empty (returns no results)

Combining ASK with FILTER

```
ASK WHERE {?any :playedIn :Giant.
           ?any :bornOn ?birthday .
           FILTER (?birthday > "1950-01-01"^^xsd:date) }
```

The ASK keyword

You can use the ASK keyword to ask true/false questions - no variable bindings are returned

"Did Elizabeth Taylor die"?

```
ASK WHERE {:ElizabethTaylor :diedOn ?any}
```

"Is Elizabeth Taylor alive?"

```
ASK WHERE {UNSAID {:ElizabethTaylor :diedOn ?any}}
```

Negation as failure: the query succeeds if the subgraph under UNSAID is empty (returns no results)

Combining ASK with FILTER

"Was any actor in Giant born after 1950?"

```
ASK WHERE {?any :playedIn :Giant.
           ?any :bornOn ?birthday .
           FILTER (?birthday > "1950-01-01"^^xsd:date) }
```

CONSTRUCT queries

CONSTRUCT queries

Motivation

SELECT queries return results in the form of a table

Ask:

```
SELECT ?director
WHERE {?m :directedBy ?director}
```

Answer:

?director
:EliaKazan
:FredGuiol
:GeorgeCukor
:GeorgeStevens
:NicholasRay
etc.

This doesn’t leverage all the expressive power of RDF graphs:

- Meaning of each header (e.g. ?director) expressed by its name (not an URI)
- Not very amenable for further SPARQL processing

CONSTRUCT queries

Motivation

CONSTRUCT queries let you instead construct a graph with the results

```
CONSTRUCT {?d rdf:type :Director .
           ?d rdfs:label ?name . }
WHERE {?any :directedBy ?d .
       ?d rdfs:label ?name . }
```

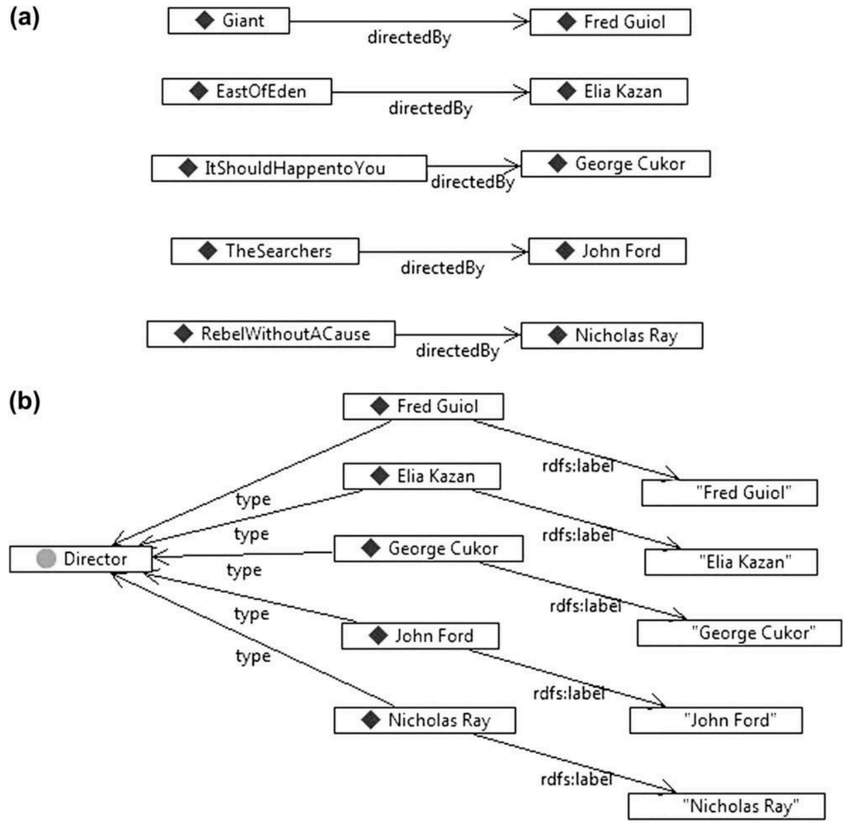


FIGURE 5.8

Constructing a model about directors from a query about movies.

CONSTRUCT queries

Motivation

Graph can then be:

- stored to the same knowledge base
- kept in memory for further processing
- stored to a different knowledge base
- serialized in some format
- provided as a SPARQL endpoint (e.g. https://dbpedia.org/sparql)

CONSTRUCT queries

Using SPARQL as a rule language

CONSTRUCT queries can be used to define new categories and create new rules:

Consider the following data:

```
:John a :Man.
:Joe a :Man.
:Eunice a :Woman .
:Maria a :Woman .
:Caroline a :Woman .
:Ted a :Man .
:Socrates a :Man .
:Caroline :hasFather :John .
:Ted :hasBrother :John .
:John :hasFather :Joe .
:Maria :hasMother :Eunice .
:Maria :hasFather :Sargent .
:Ted :hasSister :Eunice .
```

CONSTRUCT queries

Organizing Hierarchical data

Data from example below (plus a few more triples) is unorganized and inconsistent:

```
:Caroline :hasFather :John .
:John :hasFather :Joe .
:Eunice :hasFather :Joe .
:Maria :hasMother :Eunice .
:Maria :hasFather :Sargent .
:Joe :hasSon :Robert .
:Joe :hasSon :Ted .
:Ted :hasSon :Patrick .
```

CONSTRUCT queries

Organizing Hierarchical data

Data from example below (plus a few more triples) is unorganized and inconsistent:

```
:Caroline :hasFather :John .
:John :hasFather :Joe .
:Eunice :hasFather :Joe .
:Maria :hasMother :Eunice .
:Maria :hasFather :Sargent .
:Joe :hasSon :Robert .
:Joe :hasSon :Ted .
:Ted :hasSon :Patrick .
```

Hierarchical structure is not very evident, relationships run inconsistently in both directions, gender complicates things

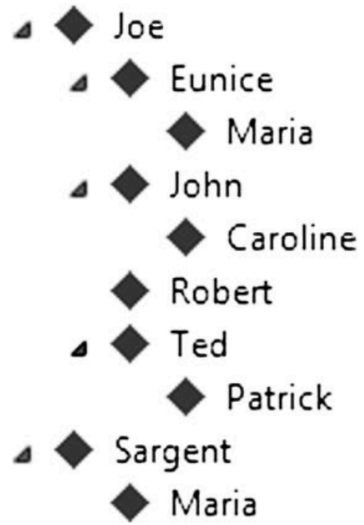
CONSTRUCT queries

Organizing Hierarchical data

We can organize it with a series of CONSTRUCT rules

```
CONSTRUCT {?s :hasParent ?o} WHERE {?s :hasMother ?o}
CONSTRUCT {?s :hasParent ?o} WHERE {?s :hasFather ?o}
CONSTRUCT {?s :hasParent ?o} WHERE {?o :hasSon ?s}
CONSTRUCT {?s :hasParent ?o} WHERE {?o :hasDaughter ?s}
```

```
:Caroline :hasParent :John .
:Eunice :hasParent :Joe .
:John :hasParent :Joe .
:Maria :hasParent :Sargent .
:Maria :hasParent :Eunice .
:Patrick :hasParent :Ted .
:Robert :hasParent :Joe .
:Ted :hasParent :Joe .
```



Transitive queries

Imagine trying to find all members (descendants) of Joe’s family

Transitive queries

Imagine trying to find all members (descendants) of Joe's family

Ask:

```
SELECT ?member
WHERE {?member :hasParent :Joe}
```

Transitive queries

Imagine trying to find all members (descendants) of Joe's family

Ask:

```
SELECT ?member
WHERE {?member :hasParent :Joe}
```

?member
Eunice
John
Robert
Ted

Transitive queries

Imagine trying to find all members (descendants) of Joe's family

Ask:

```
SELECT ?member
WHERE {?member :hasParent :Joe}
```

?member
Eunice
John
Robert
Ted

Ask:

```
SELECT ?member
WHERE {?int :hasParent :Joe .
       ?member :hasParent ?int .}
```

Transitive queries

Imagine trying to find all members (descendants) of Joe's family

Ask:

```
SELECT ?member
WHERE {?member :hasParent :Joe}
```

?member
Eunice
John
Robert
Ted

Ask:

```
SELECT ?member
WHERE {?int :hasParent :Joe .
       ?member :hasParent ?int .}
```

?member
Maria
Caroline
Patrick

Transitive queries

Using the operator *, we can query for chains of 0+ occurrences of a pattern, finding all descendants.

Ask:

```
SELECT ?member
WHERE {?member :hasParent* :Joe .}
```

Answer:

?member
Joe
Eunice
Maria
John
Caroline
Robert
Ted
Patrick

Transitive queries

If we don’t want to return Joe himself, we can use + instead of *

Ask:

```
SELECT ?member
WHERE {?member :hasParent+ :Joe .}
```

Answer:

?member
Eunice
Maria
John
Caroline
Robert
Ted
Patrick

Establishing “sameness”

Establishing “sameness”

In the LinkedMDB database, the same person might appear as an actor or director, with different URIs

Establishing “sameness”

In the LinkedMDB database, the same person might appear as an actor or director, with different URIs

```
actor:29753
      rdf:type linkedmdb:actor ;
      linkedmdb:actor_name "Clint Eastwood" .

film:38599
      rdf:type linkedmdb:film ;
      dc:title "Unforgiven" ;
      linkedmdb:actor  actor:29753 ;
      linkedmdb:director director:8533 .

director:8533
      rdf:type linkedmdb:director ;
      linkedmdb:director_name "Clint Eastwood" .
```

Establishing “sameness”

In the LinkedMDB database, the same person might appear as an actor or director, with different URIs

- This is because the underlying relational DB has an Actor table and a Director table

```
actor:29753
      rdf:type linkedmdb:actor ;
      linkedmdb:actor_name "Clint Eastwood" .

film:38599
      rdf:type linkedmdb:film ;
      dc:title "Unforgiven" ;
      linkedmdb:actor  actor:29753 ;
      linkedmdb:director director:8533 .

director:8533
      rdf:type linkedmdb:director ;
      linkedmdb:director_name "Clint Eastwood" .
```

Establishing “sameness”

In the LinkedMDB database, the same person might appear as an actor or director, with different URIs

- This is because the underlying relational DB has an Actor table and a Director table
- We would like to answer whether actor Clint Eastwood is the same person as director Clint Eastwood

```
actor:29753
      rdf:type linkedmdb:actor ;
      linkedmdb:actor_name "Clint Eastwood" .

film:38599
      rdf:type linkedmdb:film ;
      dc:title "Unforgiven" ;
      linkedmdb:actor  actor:29753 ;
      linkedmdb:director director:8533 .

director:8533
      rdf:type linkedmdb:director ;
      linkedmdb:director_name "Clint Eastwood" .
```

Establishing “sameness”

Establishing “sameness”

Fortunately, there is also a triple for matching them with Freebase resources

Establishing “sameness”

Fortunately, there is also a triple for matching them with Freebase resources

```
actor:29753 foaf:page freebase:9202a8c04000641f800000000056de6 .
director:8533 foaf:page freebase:9202a8c04000641f800000000056de6 .
```

Establishing “sameness”

Fortunately, there is also a triple for matching them with Freebase resources

```
actor:29753 foaf:page freebase:9202a8c04000641f8000000000056de6 .
director:8533 foaf:page freebase:9202a8c04000641f8000000000056de6 .
```

This external service can act as an authoritative source of information about which resources are the same

Establishing “sameness”

Fortunately, there is also a triple for matching them with Freebase resources

```
actor:29753 foaf:page freebase:9202a8c04000641f800000000056de6 .
director:8533 foaf:page freebase:9202a8c04000641f800000000056de6 .
```

This external service can act as an authoritative source of information about which resources are the same

```
CONSTRUCT {?a skos:exactMatch ?b}
WHERE {?a foaf:page ?page .
       ?b foaf:page ?page .}
```

Establishing “sameness”

Fortunately, there is also a triple for matching them with Freebase resources

```
actor:29753 foaf:page freebase:9202a8c04000641f800000000056de6 .
director:8533 foaf:page freebase:9202a8c04000641f800000000056de6 .
```

This external service can act as an authoritative source of information about which resources are the same

```
CONSTRUCT {?a skos:exactMatch ?b}
WHERE {?a foaf:page ?page .
       ?b foaf:page ?page .}
```

```
actor:29753 skos:exactMatch director:8533 .
director:8533 skos:exactMatch actor:29753 .
director:8533 skos:exactMatch director:8533 .
actor:29753 skos:exactMatch actor:29753 .
```

Establishing “sameness”

Establishing “sameness”

We can avoid some duplicates by requiring that the matches don’t be the same LinkedMDB resource:

Establishing “sameness”

We can avoid some duplicates by requiring that the matches don’t be the same LinkedMDB resource:

```
CONSTRUCT {?a skos:exactMatch ?b}
WHERE {?a foaf:page ?page .
       ?b foaf:page ?page .
FILTER (?a != ?b)}
```

Establishing “sameness”

We can avoid some duplicates by requiring that the matches don’t be the same LinkedMDB resource:

```
CONSTRUCT {?a skos:exactMatch ?b}
WHERE {?a foaf:page ?page .
       ?b foaf:page ?page .
FILTER (?a != ?b)}
```

```
actor:29753 skos:exactMatch director:8533 .
director:8533 skos:exactMatch actor:29753 .
```

Establishing “sameness”

We can avoid some duplicates by requiring that the matches don’t be the same LinkedMDB resource:

```
CONSTRUCT {?a skos:exactMatch ?b}
WHERE {?a foaf:page ?page .
       ?b foaf:page ?page .
FILTER (?a != ?b)}
```

```
actor:29753 skos:exactMatch director:8533 .
director:8533 skos:exactMatch actor:29753 .
```

If we further know that the relationship is supposed to be transitive, we can cast the Freebase URL to a string and keep the lowest

Establishing “sameness”

We can avoid some duplicates by requiring that the matches don’t be the same LinkedMDB resource:

```
CONSTRUCT {?a skos:exactMatch ?b}
WHERE {?a foaf:page ?page .
       ?b foaf:page ?page .
FILTER (?a != ?b)}
```

```
actor:29753 skos:exactMatch director:8533 .
director:8533 skos:exactMatch actor:29753 .
```

If we further know that the relationship is supposed to be transitive, we can cast the Freebase URL to a string and keep the lowest

```
CONSTRUCT {?a skos:exactMatch ?b}
WHERE {?a foaf:page ?page .
       ?b foaf:page ?page .
       FILTER (xsd:string (?a) > xsd:string (?b))
}
```

Establishing “sameness”

We can avoid some duplicates by requiring that the matches don’t be the same LinkedMDB resource:

```
CONSTRUCT {?a skos:exactMatch ?b}
WHERE {?a foaf:page ?page .
       ?b foaf:page ?page .
FILTER (?a != ?b)}
```

```
actor:29753 skos:exactMatch director:8533 .
director:8533 skos:exactMatch actor:29753 .
```

If we further know that the relationship is supposed to be transitive, we can cast the Freebase URL to a string and keep the lowest

```
CONSTRUCT {?a skos:exactMatch ?b}
WHERE {?a foaf:page ?page .
       ?b foaf:page ?page .
       FILTER (xsd:string (?a) > xsd:string (?b))
}
```

```
director:8533 skos:exactMatch actor:29753 .
```

Advanced features

Ordering results

SPARQL queries are not ordered by default. We can specify order with ORDER BY

Ask:

```
SELECT ?title ?date
WHERE {:JamesDean :playedIn ?movie.
       ?movie rdfs:label ?title .
       ?movie dc:date  ?date . }
ORDER BY ?date
```

Answer:

?title	?date
EastOfEden	1955
RebelWithoutaCause	1955
Giant	1956

Ask:

```
SELECT ?title ?date
WHERE {:JamesDean :playedIn ?movie.
       ?movie rdfs:label ?title .
       ?movie dc:date  ?date . }
ORDER BY ?title
```

Answer:

?title	?date
EastOfEden	1955
Giant	1956
RebelWithoutaCause	1955

Ordering results

We can also specify how many results to show with LIMIT

-What's the first movie by James Dean?

Ask:

```
SELECT ?title
WHERE {:JamesDean :playedIn ?m.
       ?m rdfs:label ?title .
       ?m dc:date  ?date . }
ORDER BY ?date
LIMIT 1
```

Answer:

?title
East Of Eden

Aggregating results

Aggregating results

We can aggregate data with COUNT, MIN, MAX, AVG and SUM

Aggregating results

We can aggregate data with COUNT, MIN, MAX, AVG and SUM

```
SELECT (COUNT (?movie) AS ?howmany)
WHERE {:JamesDean ?playedIn ?movie .}
```

?howmany
3

Aggregating results

We can aggregate data with COUNT, MIN, MAX, AVG and SUM

```
SELECT (COUNT (?movie) AS ?howmany)
WHERE {:JamesDean ?playedIn ?movie .}
```

?howmany
3

Aggregating results

We can aggregate data with COUNT, MIN, MAX, AVG and SUM

```
SELECT (COUNT (?movie) AS ?howmany)
WHERE {:JamesDean ?playedIn ?movie .}
```

?howmany
3

The aggregator keyword (in this case Count) specifies which variable to aggregate

Aggregating results

We can aggregate data with COUNT, MIN, MAX, AVG and SUM

```
SELECT (COUNT (?movie) AS ?howmany)
WHERE {:JamesDean ?playedIn ?movie .}
```

?howmany
3

The aggregator keyword (in this case Count) specifies which variable to aggregate

The variable name after AS specifies how it is displayed

Aggregating results

We can aggregate data with COUNT, MIN, MAX, AVG and SUM

```
SELECT (COUNT (?movie) AS ?howmany)
WHERE {:JamesDean ?playedIn ?movie .}
```

?howmany
3

The aggregator keyword (in this case Count) specifies which variable to aggregate

The variable name after AS specifies how it is displayed

Aggregating results

Consider the following data about sales

Company	Amount	Year
ACME	$1250	2010
PRIME	$3000	2009
ABC	$2500	2009
ABC	$2800	2010
PRIME	$1950	2010
ACME	$2500	2009
ACME	$3100	2010
ABC	$1500	2009
ACME	$1250	2009
PRIME	$2350	2009
PRIME	$1850	2010

```
:row1 a :Sale .
:row1 :company :ACME .
:row1 :amount 1250 .
:row1 :year 2010 .
```

Aggregating results

Consider the following data about sales

Company	Amount	Year
ACME	$1250	2010
PRIME	$3000	2009
ABC	$2500	2009
ABC	$2800	2010
PRIME	$1950	2010
ACME	$2500	2009
ACME	$3100	2010
ABC	$1500	2009
ACME	$1250	2009
PRIME	$2350	2009
PRIME	$1850	2010

Asking for sale totals

Ask:

```
SELECT (SUM (?val) AS ?total)
WHERE {?s a :Sale .
       ?s :amount ?val }
```

Answer:

?total
24050.00

Aggregating results

Consider the following data about sales

Company	Amount	Year
ACME	$1250	2010
PRIME	$3000	2009
ABC	$2500	2009
ABC	$2800	2010
PRIME	$1950	2010
ACME	$2500	2009
ACME	$3100	2010
ABC	$1500	2009
ACME	$1250	2009
PRIME	$2350	2009
PRIME	$1850	2010

Asking for sale totals

Ask:

```
SELECT (SUM (?val) AS ?total)
WHERE {?s a :Sale .
       ?s :amount ?val }
```

Answer:

?total
24050.00

Aggregating results

Consider the following data about sales

Company	Amount	Year
ACME	$1250	2010
PRIME	$3000	2009
ABC	$2500	2009
ABC	$2800	2010
PRIME	$1950	2010
ACME	$2500	2009
ACME	$3100	2010
ABC	$1500	2009
ACME	$1250	2009
PRIME	$2350	2009
PRIME	$1850	2010

Grouping sale totals by year

Ask:

```
SELECT ?year  (SUM (?val) AS ?total)
WHERE {?s a :Sale .
       ?s :amount ?val .
       ?s :year ?year  }
GROUP BY ?year
```

Answer:

?year	?total
2009	13100.00
2010	10950.00

Aggregating results

Consider the following data about sales

Company	Amount	Year
ACME	$1250	2010
PRIME	$3000	2009
ABC	$2500	2009
ABC	$2800	2010
PRIME	$1950	2010
ACME	$2500	2009
ACME	$3100	2010
ABC	$1500	2009
ACME	$1250	2009
PRIME	$2350	2009
PRIME	$1850	2010

Did any company sell > $5000 in a single year?

Ask:

```
SELECT ?year ?company (SUM (?val) AS ?total)
WHERE {?s a :Sale .
       ?s :amount ?val .
       ?s :year ?year  .
       ?s :company ?company .
}
GROUP BY ?year ?company
HAVING (?total > 5000)
```

Answer:

?year	?company	?total
2009	PRIME	5350.00

Other advanced features

- Subqueries: did any company sell more in 2010 than 2009?

Ask:

```
SELECT ?company
WHERE {
    {SELECT ?company ((SUM(?val)) AS ?total09)
        WHERE {
            ?s a :Sale .
            ?s :amount ?val .
            ?s :company ?company .
            ?s :year 2009 . }
        GROUP BY ?company } .
    {SELECT ?company ((SUM(?val)) AS ?total10)
        WHERE {
            ?s a :Sale .
            ?s :amount ?val .
            ?s :company ?company .
            ?s :year 2010 .}
        GROUP BY ?company } .
    FILTER (?total10 > ?total09) . }
```

Answer:

?company
ACME

Other advanced features

Unions

- By default, there's an implicit AND between triples in the query pattern
- Unions let us ask for triples that are in one graph pattern OR another

Ask:

```
SELECT ?actor
WHERE {
       {?actor :playedIn :Giant .}
       UNION
       {?actor :playedIn :RebelWithoutaCause .}
      }
```

Answer:

actor
Ann Doran
Carroll Baker
Elizabeth Taylor
James Dean
James Dean
Jim Backus
Mercedes McCambridge
Natalie Wood
Rock Hudson
Sal Mineo
Sal Mineo