Semantic Web

Storage and Querying

Motivation

- Having RDF data available is not enough
  - Need tools to process, transform, and reason with the information
  - Need a way to store the RDF data and interact with it
- Are existing storage systems appropriate to store RDF data?
- Are existing query languages appropriate to query RDF data?

Databases and RDF

- Relational database are a well established technology to store information and provide query support (SQL)
- Relational database have been designed and implemented to store concepts in a predefined (not frequently alterable) schema.

- How can we store the following RDF data in a relational database?

```
<rdf:Description rdf:about="949318">
    <rdf:type rdf:resource="&uni;lecturer"/>
    <uni:name>Dieter Fensel</uni:name>
    <uni:title>University Professor</uni:title>
</rdf:Description>
```

- Several solutions are possible

Databases and RDF

- **Solution 1:** Relational “Traditional” approach

Lecturer		
id	**name**	**title**
949318	Dieter Fensel	University Professor

- **Approach:** We can create a table “Lecturer” to store information about the “Lecturer” RDF Class.

- **Drawbacks:** Every time we need to add new content we have to create a new table -> Not scalable, not dynamic, not based on the RDF principles (TRIPLES)

Databases and RDF

- **Solution 2:** Relational "Triple" based approach

Statement			
Subject	**Predicate**	**ObjectURI**	**ObjectLiteral**
101	102	103	null
101	104		201
101	105		202
103	…	…	null

Resources	
Id	**URI**
101	949318
102	rdf:type
103	uni:lecturer
104	…

Literals	
Id	**Value**
201	Dieter Fensel
202	University Professor
203	…
…	…

- **Approach:** We can create a table to maintain all the triples S P O (and distinguish between URI objects and literals objects).

- **Drawbacks:** We are flexible w.r.t. adding new statements dynamically without any change to the database structure… but what about querying?

Why Native RDF Repositories?

- What happens if I want to find the names of all the lecturers?
- **Solution 1:** Relation “traditional” approach:

```
SELECT NAME FROM LECTURER
```

- We need to query a single table which is easy, quick and performing
- No **JOIN** required (the most expensive operation in a db query)
- **BUT** we already said that Traditional approach is not appropriate

Why Native RDF Repositories?

- What happens if I want to find the names of all the lecturers?
- **Solution 2:** Relational “triple” based approach:

```
SELECT L.Value FROM Literals AS L
  INNER JOIN Statement AS S ON
  S.ObjectLiteral=L.ID
  INNER JOIN Resources AS R ON R.ID=S.Predicate
  INNER JOIN Statement AS S1 ON
  S1.Predicate=S.Predicate
  INNER JOIN Resources AS R1 ON
  R1.ID=S1.Predicate
  INNER JOIN Resources AS R2 ON
  R2.ID=S1.ObjectURI
  WHERE R.URI = “uni:name”
  AND R1.URI = “rdf:type”
  AND R2.URI = “uni:lecturer”
```

Why Native RDF Repositories?

Solution 2

- The query is quite complex: 5 JOINS!
- This require a lot of optimization specific for RDF and triple data storage, that it is not included in Relational DB
- For achieving efficiency a layer on top of a database is required. More, SQL is not appropriate to extract RDF fragments
- *Do we need a new query language?*

Query Languages

- *Querying* and inferencing is the very purpose of information representation in a machine-accessible way

- A query language is a language that allows a user to retrieve information from a “data source”
  - E.g. data sources
    - A simple text file
    - XML file
    - A database
    - The “Web”

- Query languages usually includes *insert* and *update* operations

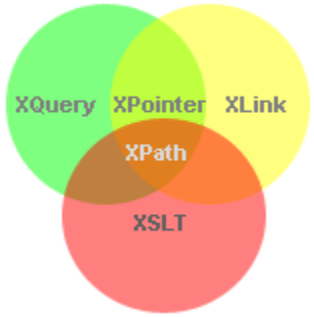
Example of Query Languages

- *SQL*
  - Query language for relational databases
- *XQuery*, *XPointer* and *XPath*
  - Query languages for XML data sources
- *SPARQL*
  - Query language for RDF graphs
- *RDQL*
  - Query language for RDF in Jena models

XPath: a simple query language for XML trees

- The basis for most XML query languages
  - Selection of document parts
  - Search context: ordered set of nodes
- Used extensively in XSLT
  - XPath itself has non-XML syntax
- Navigate through the XML Tree
  - Similar to a file system (“/“, “../“, “ ./“, etc.)
  - Query result is the final search context, usually a set of nodes
  - Filters can modify the search context
  - Selection of nodes by element names, attribute names, type, content, value, relations
- Several pre-defined functions
- Version 1.0, **W3C Recommendation 16 November 1999**
- Version 2.0, **W3C Recommendation 23 January 2007**

Other XML Query Languages



- XQuery
  - Building up on the same functions and data types as XPath
  - With XPath 2.0 these two languages get closer
  - XQuery is not XML based, but there is an XML notation (XQueryX)
  - XQuery 1.0, W3C Recommendation 23 January 2007
- XLink 1.0, W3C Recommendation 27 June 2001
  - Defines a standard way of creating hyperlinks in XML documents
- XPointer 1.0, W3C Candidate Recommendation
  - Allows the hyperlinks to point to more specific parts (fragments) in the XML document
- XSLT 2.0, W3C Recommendation 23 January 2007

Why a New Language?

- RDF description (1):

```
<rdf:Description rdf:about="949318">
    <rdf:type rdf:resource="&uni;lecturer"/>
    <uni:name>Dieter Fensel</uni:name>
    <uni:title>University Professor</uni:title>
</rdf:Description>
```

- XPath query:

```
/rdf:Description[rdf:type=
"http://www.mydomain.org/uni-ns#lecturer"]/uni:name
```

Why a New Language?

- RDF description (2):

```
<uni:lecturer rdf:about="949318">
    <uni:name>Dieter Fensel</uni:name>
    <uni:title>University Professor</uni:title>
</uni:lecturer>
```

- XPath query:

```
//uni:lecturer/uni:name
```

Why a New Language?

- RDF description (3):

```
<uni:lecturer rdf:about="949318"
          uni:name="Dieter Fensel"
               uni:title="University Professor"/>
```

- XPath query:

```
//uni:lecturer/@uni:name
```

Why a New Language?

- What is the difference between these three definitions?

- RDF description (1):

```
<rdf:Description rdf:about="949318">
      <rdf:type rdf:resource="&uni;lecturer"/>
      <uni:name>Dieter Fensel</uni:name>
      <uni:title>University Professor</uni:title>
</rdf:Description>
```

- RDF description (2):

```
<uni:lecturer rdf:about="949318">
      <uni:name>Dieter Fensel</uni:name>
      <uni:title>University Professor</uni:title>
</uni:lecturer>
```

- RDF description (3):

```
<uni:lecturer rdf:about="949318"
                    uni:name="Dieter Fensel"
                         uni:title="University Professor"/>
```

Why a New Language?

- All three description denote the same thing:

```
⟨#949318, rdf:type, <uni:lecturer>⟩
⟨#949318, <uni:name>, "Dieter Fensel"⟩
⟨#949318, <uni:title>, "University Professor"⟩
```

- But the queries are different depending on a particular serialization:

```
/rdf:Description[rdf:type=
"http://www.mydomain.org/uni-ns#lecturer"]/uni:name

//uni:lecturer/uni:name

//uni:lecturer/@uni:name
```

RDF REPOSITORIES

Efficient storage of RDF data

Different Architectures

- Based on their implementation, can be divided into 3 broad categories : In-memory, Native, Non-native Non-memory.

- **In – Memory** : RDF Graph is stored as triples in main –memory
  - E.g. Storing an RDF graph using Jena API/ Sesame API.

- **Native** : Persistent storage systems with their own implementation of databases. Provide support for transactions, own query compiler and generally their own procedure language
  - E.g. Sesame Native, Virtuoso, AllegroGraph, Oracle 11g.

- **Non-Native Non-Memory** : Persistent storage systems set-up to run on third party DBs.
  - E.g. Jena SDB.

Implications

- Scalability: In-memory stores come no way near in matching the storage capacity of a Persistent store.
- Different query languages supported to varying degrees.
  - Sesame – SeRQL, SPARQL
  - Oracle 11g – Own query language.
- Different level of inferencing.
  - Sesame supports RDFS inference, AllegroGraph – RDFS++,
  - Oracle 11g – RDFS++, OWL Prime
  - In-memory store usually supports highest degree of reasoning. Any reasoner like Pellet, Jena Reasoner can be used.
- Lack of interoperability and portability.
  - More pronounced in Native stores.

What is OWLIM?

- OWLIM is a scalable semantic repository which allows
  - Management, integration, and analysis of heterogeneous data
  - Combined with light-weight reasoning capabilities

- OWLIM is RDF database with high-performance reasoning
  - The inference is based on logical rule-entailment
  - Full RDFS and limited OWL Lite and Horst are supported
  - Custom semantics defined via rules and axiomatic triples

Rule-Based Inference

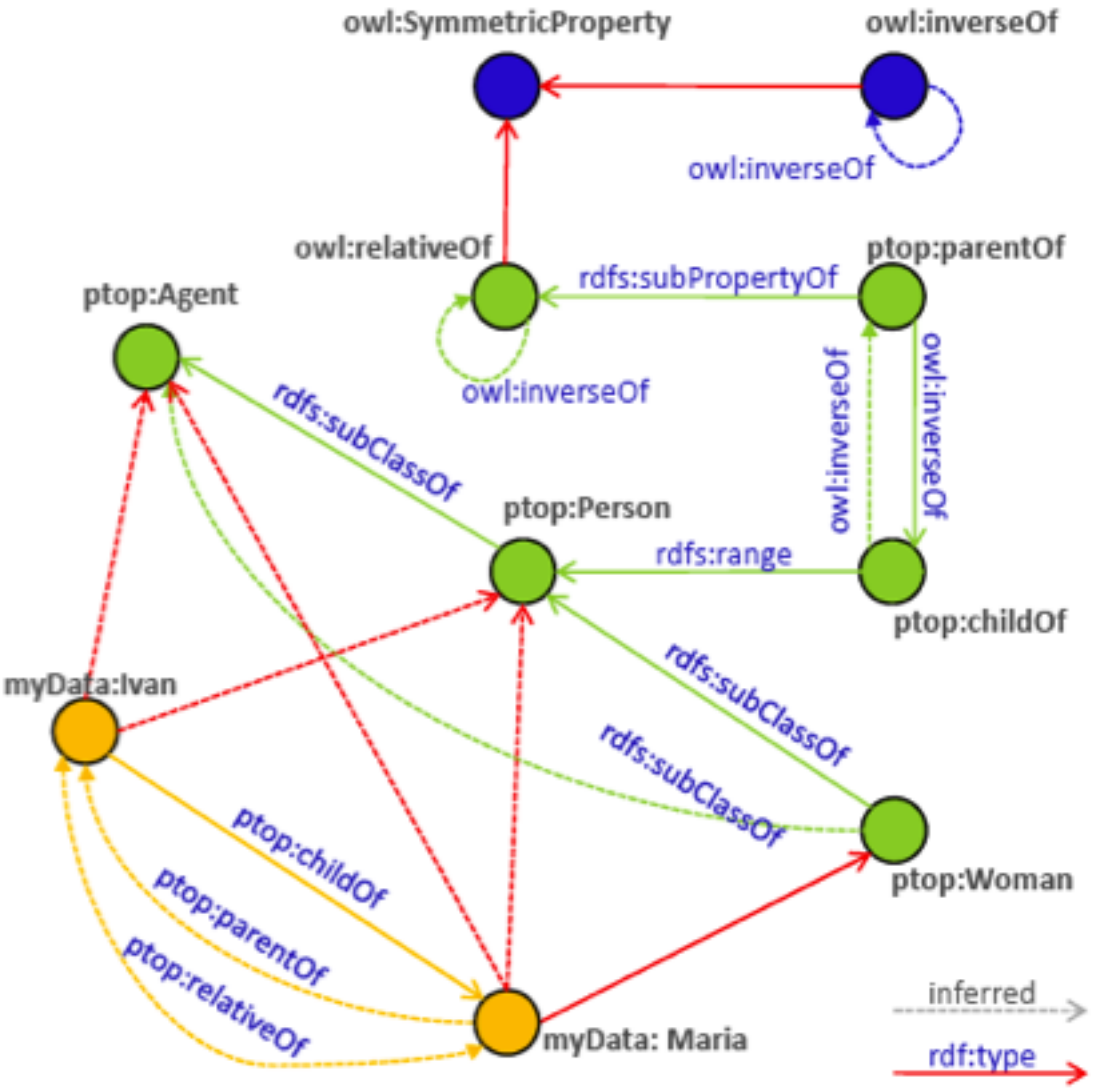
<C1,rdfs:subClassOf,C2>
<C2,rdfs:subClassOf,C3>
=> <C1,rdfs:subClassOf,C3>

<I,rdf:type,C1>
<C1,rdfs:subClassOf,C2>
=> <I,rdf:type,C2>

<I1,P1,I2>
<P1,rdfs:range,C2>
=> <I2,rdf:type,C2>

<P1,owl:inverseOf,P2>
<I1,P1,I2>
=> <I2,P2,I1>

<P1,rdf:type,owl:SymmetricProperty>
=> <P1,owl:inverseOf,P1>



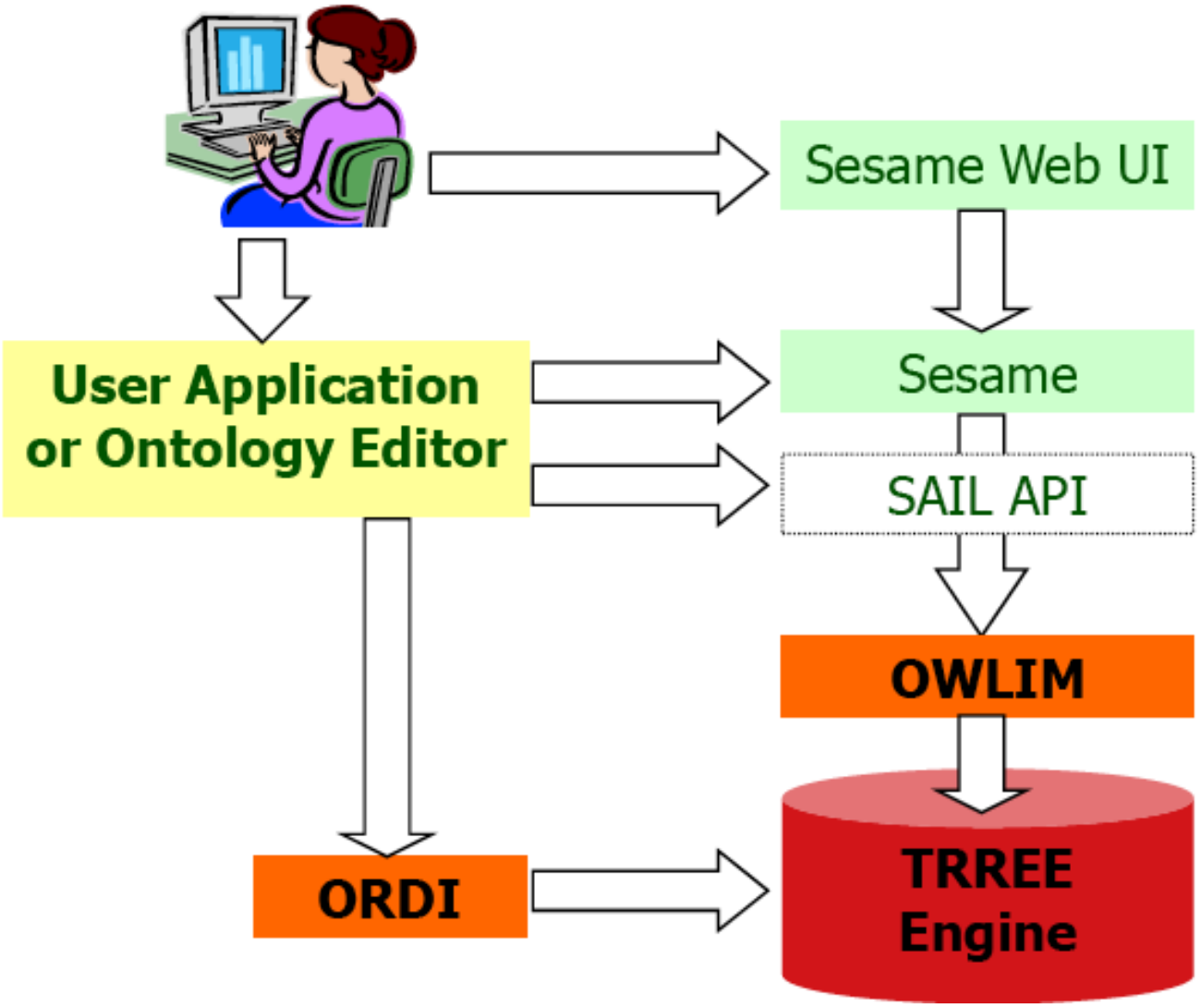
Using OWLIM

- OWLIM is implemented as a storage and inference layer (SAIL) for Sesame

- OWLIM is based on TRREE
  - TRREE = Triple Reasoning and Rule Entailment Engine
  - TRREE takes care of storage, indexing, inference and query evaluation
  - TRREE has different flavors, mapping to different OWLIM species
  - TRREE is also in the hart of ORDI – Ontotext's semantic data integration middleware framework

- OWLIM can be used and accessed in different ways:
  - By end user: through the web UI routines of Sesame
  - By applications: though the API's of Sesame
  - Applications can either embed it as a library or access it as standalone server

Sesame. TRREE. ORDI. and OWLIM



OWLIM and Sesame

- OWLIM is available as a Storage and Inference Layer (SAIL) for Sesame RDF.

- Benefits:
  - Sesame's infrastructure, documentation, user community, etc.
  - Support for multiple query language (RQL, RDQL, SeRQL)
  - Support for import and export formats (RDF/XML, N-Triples, N3)

OWLIM versions

- Two major OWLIM species: SwiftOWLIM and BigOWLIM
  - Based on the corresponding versions of TRREE
  - Share the same inference and semantics (rule-compiler, etc.)
  - They are identical in terms of usage and integration
    - The same APIs, syntaxes, languages (thanks to Sesame)
    - Different are only the configuration parameters for performance tuning
- SwiftOWLIM is good for experiments and medium-sized data
  - Extremely fast loading of data (incl. inference, storage, etc.)
- BigOWLIM is designed to handle huge volumes of data and intensive querying
  - Query optimizations ensure faster query evaluation on large datasets
  - Scales much better, having lower memory requirements

SwiftOWLIM

- SwiftOWLIM uses SwiftTRREE engine
- It performs in-memory reasoning and query evaluation
  - Based on hash-table-like indices
- Combined with reliable persistence strategy
- Very fast upload, retrieval, query evaluation for huge KB
  - It scales to 10 million statements on a $500-worth PC
  - It loads the 7M statements of LUBM(50,0) dataset in 2 minutes
- Persistency (in SwiftOWLIM 3.0):
  - Binary Persistence, including the inferred statements
  - Allows for instance initialization

BigOWLIM

- BigOWLIM is an enterprise class repository
  - http://www.ontotext.com/owlim/big/
- BigOWLIM is an even more scalable not-in-memory version, based on the corresponding version of the TRREE engine
  - The “light-weight” version of OWLIM, which uses in-memory reasoning and query evaluation is referred as SwiftOWLIM
- BigOWLIM does not need to maintain all the concepts of the repository in the main memory in order to operate
- BigOWLIM stores the contents of the repository (including the “inferred closure”) in binary files
  - This allows instant startup and initialization of large repositories, because it does not need to parse, re-load and re-infer all knowledge from scratch

BigOWLIM vs. SwiftOWLIM

- BigOWLIM uses sorted indices
  - While the indices of SwiftOWLIM are essentially hash-tables
  - In addition to this BigOWLIM maintains data statistics, to allow …
- Database-like query optimizations
  - Re-ordering of the constraints in the query has no impact on the execution time
  - Combined with the other optimizations, this feature delivers dramatic improvements to the evaluation time of “heavy” queries
- Special handling of equivalence classes
  - Large equivalent classes does not cause excessive generation of inferred statements

SwiftOWLIM and BigOWLIM

	SwiftOWLIM	BigOWLIM
Scale (Mil. of explicit statem.)	10 MSt, using 1.6 GB RAM 100 MSt, using 16 GB RAM	130 MSt, using 1.6 GB 1068 MSt, using 8 GB
Processing speed (load + infer + store)	30 KSt/s on notebook 200 KSt/s on server	5KSt/s on notebook 60 KSt/s on server
Query optimization	No	Yes
Persistence	Back-up in N-Triples	Binary data files and indices
License and Availability	Open-source under LGPL; Uses SwiftTRREE that is free, but not open-source	Commercial. Research and evaluation copies provided for free

Which RDF Store to Choose for an App?

- Frequency of loads that the application would perform.
- Single scaling factor and linear load times.
- Level of inferencing.
- Support for which query language. W3C recommendations.
- Special system needs.
  - E.g. Allegrograph needs 64 bit processor.
- There’s no a single answer, the application requirements determine the best choice!

SPARQL

A language to query RDF data

Querying RDF

- SPARQL
  - RDF Query language
  - Based on RDQL
  - Uses SQL-like syntax

- Example:

```
PREFIX uni: <http://example.org/uni/>

SELECT ?name
FROM <http://example.org/personal>
WHERE { ?s uni:name ?name.
?s rdf:type uni:lecturer }
```

SPARQL Queries

```
PREFIX uni: <http://example.org/uni/>
SELECT ?name
FROM <http://example.org/personal>
WHERE { ?s uni:name ?name. ?s rdf:type uni:lecturer }
```

- PREFIX
  - Prefix mechanism for abbreviating URIs
- SELECT
  - Identifies the variables to be returned in the query answer
  - SELECT DISTINCT
  - SELECT REDUCED
- FROM
  - Name of the graph to be queried
  - FROM NAMED
- WHERE
  - Query pattern as a list of triple patterns
- LIMIT
- OFFSET
- ORDER BY

SPARQL Query keywords

- PREFIX: based on namespaces

- DISTINCT: The DISTINCT solution modifier eliminates duplicate solutions. Specifically, each solution that binds the same variables to the same RDF terms as another solution is eliminated from the solution set.

- REDUCED: While the DISTINCT modifier ensures that duplicate solutions are eliminated from the solution set, REDUCED simply permits them to be eliminated. The cardinality of any set of variable bindings in an REDUCED solution set is at least one and not more than the cardinality of the solution set with no DISTINCT or REDUCED modifier.

- LIMIT: The LIMIT clause puts an upper bound on the number of solutions returned. If the number of actual solutions is greater than the limit, then at most the limit number of solutions will be returned.

SPARQL Query keywords

- OFFSET: OFFSET causes the solutions generated to start after the specified number of solutions. An OFFSET of zero has no effect.

- ORDER BY: The ORDER BY clause establishes the order of a solution sequence.

- Following the ORDER BY clause is a sequence of order comparators, composed of an expression and an optional order modifier (either ASC() or DESC()). Each ordering comparator is either ascending (indicated by the ASC() modifier or by no modifier) or descending (indicated by the DESC() modifier).

Example RDF Graph

```
<http://example.org/#john> <http://.../vcard-rdf/3.0#FN> "John Smith"

<http://example.org/#john> <http://.../vcard-rdf/3.0#N> :_X1
_:X1 <http://.../vcard-rdf/3.0#Given> "John"
_:X1 <http://.../vcard-rdf/3.0#Family> "Smith"

<http://example.org/#john> <http://example.org/#hasAge> "32"

<http://example.org/#john> <http://example.org/#marriedTo> <#mary>

<http://example.org/#mary> <http://.../vcard-rdf/3.0#FN> "Mary Smith"

<http://example.org/#mary> <http://.../vcard-rdf/3.0#N> :_X2
_:X2 <http://.../vcard-rdf/3.0#Given> "Mary"
_:X2 <http://.../vcard-rdf/3.0#Family> "Smith"

<http://example.org/#mary> <http://example.org/#hasAge> "29"
```

SPARQL Queries: All Full Names

"Return the full names of all people in the graph"

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?fullName
WHERE {?x vCard:FN ?fullName}
```

result:

```
fullName
==================
"John Smith"
"Mary Smith"
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Properties

"Return the relation between John and Mary"

```
PREFIX ex: <http://example.org/#>
SELECT ?p
WHERE {ex:john ?p ex:mary}
```

result:

```
p
==================
<http://example.org/#marriedTo>
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Complex Patterns

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
PREFIX ex: <http://example.org/#>
SELECT ?y
WHERE {?x vCard:FN "John Smith".
       ?x ex:marriedTo ?y}
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Complex Patterns

"Return the spouse of a person by the name of John Smith"

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
PREFIX ex: <http://example.org/#>
SELECT ?y
WHERE {?x vCard:FN "John Smith".
       ?x ex:marriedTo ?y}
```

result:

```
y
==================
<http://example.org/#mary>
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Blank Nodes

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?name, ?firstName
WHERE {?x vCard:N ?name .
      ?name vCard:Given ?firstName}
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Blank Nodes

"Return the first name of all people in the KB"

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?name, ?firstName
WHERE {?x vCard:N ?name .
       ?name vCard:Given ?firstName}
```

result:

```
name firstName
================
_:a "John"
_:b "Mary"
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Building RDF Graph

"Rewrite the naming information in original graph by using the **`foaf:name`**"

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>

CONSTRUCT { ?x foaf:name ?name }
WHERE  { ?x vCard:FN ?name }
```

result:

```
#john foaf:name "John Smith"
#marry foaf:name "Marry Smith"
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Building RDF Graph

"Rewrite the naming information in original graph by using the **`foaf:name`**"

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>

CONSTRUCT { ?x foaf:name ?name }
WHERE  { ?x vCard:FN ?name }
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
                                                  ;
                                                h" ] ;
                                                " ;
                                                  ;
                                                h" ] ;
```

```
<rdf:RDF
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:foaf="http://xmlns.com/foaf/0.1/"
      xmlns:ex="http://example.org">
  <rdf:Description rdf:about=ex:john>
       <foaf:name>John Smith</foaf:name>
  </rdf:Description>
  <rdf:Description rdf:about=ex:marry>
       <foaf:name>Marry Smith</foaf:name>
  </rdf:Description>
</rdf:RDF>
```

SPARQL Queries: Testing if the Solution Exists

"Are there any married persons in the KB?"

```
PREFIX ex: <http://example.org/#>
ASK { ?person ex:marriedTo ?spouse }
```

result:

yes

==========

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Constraints (Filters)

"Return all people over 30 in the KB"

```
PREFIX ex: <http://example.org/#>
SELECT ?x
WHERE {?x hasAge ?age .
FILTER(?age > 30)}
```

result:

```
x
==================
<http://example.org/#john>
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Optional Patterns

"Return all people and (optionally) their spouse"

```
PREFIX ex: <http://example.org/#>
SELECT ?person, ?spouse
WHERE {?person ex:hasAge ?age .
OPTIONAL { ?person ex:marriedTo ?spouse } }
```

result:

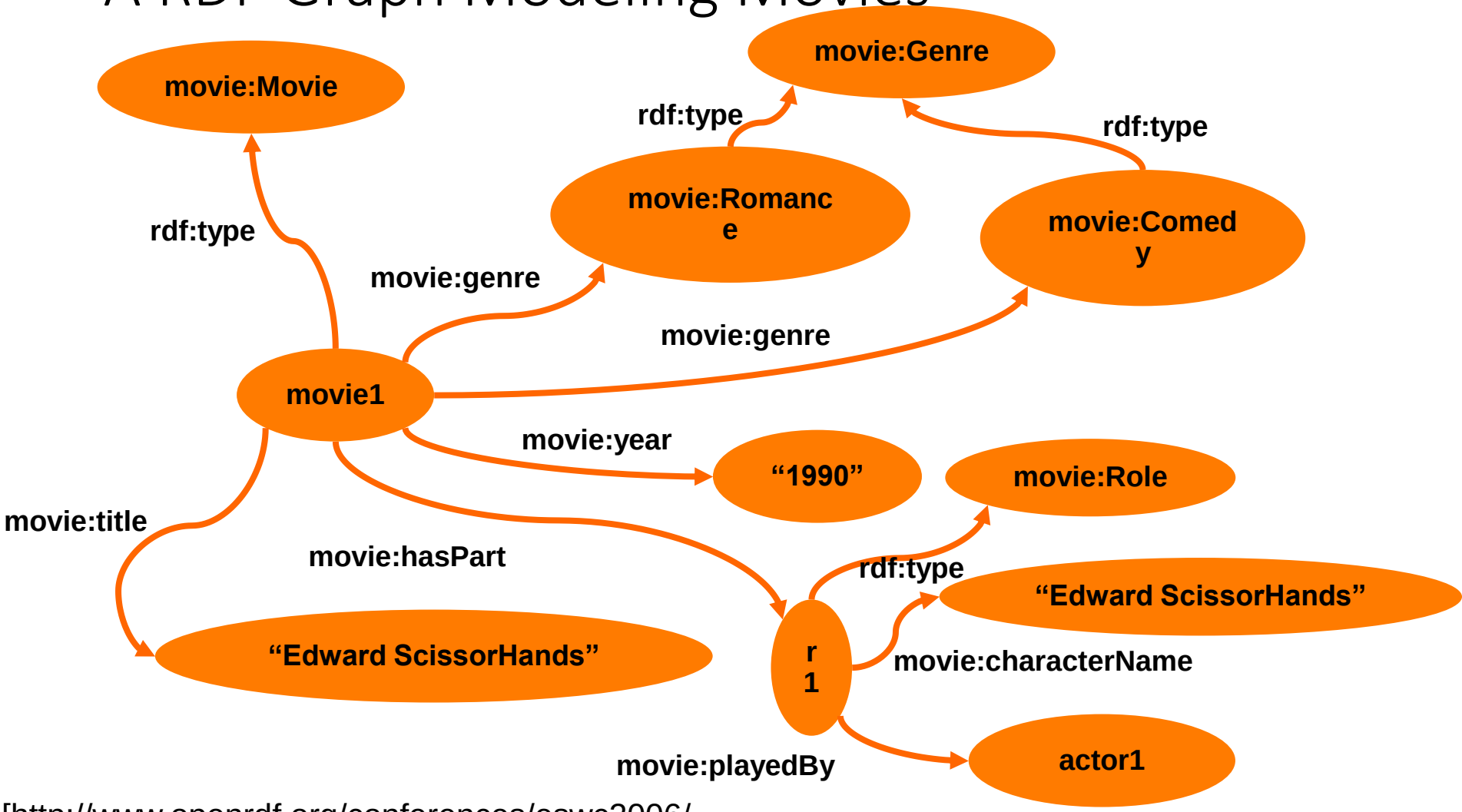
```
?person ?spouse
============================
<http://example.org/#mary>
<http://example.org/#john> <http://example.org/#mary>
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

ILLUSTRATION BY A LARGER EXAMPLE

An example of usage of SPARQL

A RDF Graph Modeling Movies



[http://www.openrdf.org/conferences/eswc2006/Sesame-tutorial-eswc2006.ppt]

Example Query 1

- Select the movies that has a character called “Edward Scissorhands”

```
PREFIX movie: <http://example.org/movies/>

SELECT DISTINCT ?x ?t
WHERE {
            ?x movie:title ?t ;
            movie:hasPart ?y .
            ?y movie:characterName ?z .
            FILTER (?z = “Edward
 Scissorhands”@en)
      }
```

Example Query 1

```
PREFIX movie: <http://example.org/movies/>

SELECT DISTINCT ?x ?t
WHERE {
            ?x movie:title ?t ;
            movie:hasPart ?y .
            ?y movie:characterName ?z .
            FILTER (?z = "Edward Scissorhands"@en)
      }
```

- Note the use of ";" This allows to create triples referring to the previous triple pattern (extended version would be **?x movie:hasPart ?y**)
- Note as well the use of the language speciation in the filter **@en**

Example Query 2

- Create a graph of actors and relate them to the movies they play in (through a new 'playsInMovie' relation)

```
PREFIX movie: <http://example.org/movies/>
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>

CONSTRUCT {
      ?x foaf:firstName ?fname.
      ?x foaf:lastName ?lname.
      ?x movie:playInMovie ?m
      }
WHERE  {
            ?m movie:title ?t ;
            movie:hasPart ?y .
            ?y movie:playedBy ?x .
      ?x foaf:firstName ?fname.
      ?x foaf:lastName ?lname.
}
```

Example Query 3

- Find all movies which share at least one genre with “Gone with the Wind”

```
PREFIX movie: <http://example.org/movies/>

SELECT DISTINCT ?x2 ?t2
WHERE {
                ?x1 movie:title ?t1.
                ?x1 movie:genre ?g1.
                ?x2 movie:genre ?g2.
                ?x2 movie:title ?t2.
                FILTER (?t1 = “Gone with the Wind”@en &&
                ?x1!=?x2 && ?g1=?g2)
      }
```

EXTENSIONS

Massive Data Inference in RDF Repositories

- Present common implementations:
  - Make a number of small queries to propagate the effects of rule firing.
  - Each of these queries creates an interaction with the database.
  - Not very efficient
- Approaches
  - Snapshot the contents of the database-backed model into Main Memory (RAM) for the duration of processing by the inference engine.
  - Performing inferencing in-stream.
    - Precompute the inference closure of ontology and analyze the in-coming data-streams, add triples to it based on your inference closure.
    - Assumes rigid separation of the RDF Data(A-box) and the Ontology data(T-box)
    - Even this may not work for very large ontologies – BioMedical Ontologies

Extending SPARQL

- SPARQL is still under continuous development. Current investigate possible extensions includes:
  - Better support for OWL semantics
  - RDF data insert and update
  - Aggregate functions
  - Standards XPath functions
  - Manipulation of Composite Datasets
  - Access to RDF lists

Questions?