A Distributed SPARQL Query Engine

Lei Zou

Peking University
Institute of Computer Science and Technology

Joint work with Tamer Ozsu (UW), Lei Chen (HKUST), Peng Peng, Shuo Han, Youhuan Li, Dongyan Zhao (PKU)
RDF and Semantic Web

- RDF is a language for the conceptual modeling of information about web resources
- A building block of semantic web
  - Facilitates exchange of information
  - Search engines can retrieve more relevant information
  - Facilitates data integration (mashes)
- Machine understandable
  - Understand the information on the web and the interrelationships among them
What’s Sematic Web: A Simple Example (RDFa)

The traditional Web (HTML) only considers the display of the content.

*How is the page displayed, such as which font and the format of the pictures?*

```html
<html>
  <font size="3" color="red"> Lei Zou </font>
  Email: <a href="mailto: zoulei@pku.edu.cn">zoulei@pku.edu.cn</a>
  <p>
    <font size="3" color="black">Publications: </font>
  </p>
  <div>
    Lei Zou, Jinhui Mo, Lei Chen, M. Tamer Ozsu, Dongyan Zhao, gStore: Answering SPARQL Queries Via Subgraph Matching, VLDB, 2011
  </div>
</html>
```
What’s Semantic Web: A Simple Example (RDFa)

Semantic Web considers the **semantics** of the content.

*What does the content in the page mean? e.g., What are the mean of “zoulei@pku.edu.cn” and “VLDB”?*

```html
<html>
<div resource="#me" typeof="Person">
  <font size="3" color="red">Lei Zou</font>
  <br/>
  <a property="http://xmlns.com/foaf/0.1/mbox" href="mailto: zoulei@pku.edu.cn ">
zoulei@pku.edu.cn
  </a>
<p>
  <font size="3" color="black">Publications: </font>
</p>
<div resource="www.vldb.org/pvldb/vol4/p482-zou.pdf">
  Lei Zou, Jinghui Mo, Lei Chen, M. Tamer Özsu, Dongyan Zhao, gStore: Answering SPARQL Queries Via Subgraph Matching,
  VLDB 2011
</div>
</div>
</html>
```
What’s Semantic Web: Google Snippet
### Extracted structured data

<table>
<thead>
<tr>
<th>rdfa-node</th>
<th>relationship:</th>
</tr>
</thead>
<tbody>
<tr>
<td>name:</td>
<td>mbox</td>
</tr>
<tr>
<td>value:</td>
<td><a href="mailto:zoulei@pku.edu.cn">zoulei@pku.edu.cn</a></td>
</tr>
<tr>
<td>href:</td>
<td>mailto:%<a href="mailto:20zoulei@pku.edu.cn">20zoulei@pku.edu.cn</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>property:</th>
</tr>
</thead>
<tbody>
<tr>
<td>name:</td>
</tr>
<tr>
<td>contributor:</td>
</tr>
<tr>
<td>contributor:</td>
</tr>
<tr>
<td>contributor:</td>
</tr>
<tr>
<td>contributor:</td>
</tr>
<tr>
<td>contributor:</td>
</tr>
<tr>
<td>title:</td>
</tr>
<tr>
<td>publisher:</td>
</tr>
<tr>
<td>date:</td>
</tr>
</tbody>
</table>
What’s Semantic Web: Google Snippet

My name is Bob Smith, but people call me Smithy. Here is my homepage: www.example.com. I live in Albuquerque, NM and work as an engineer at ACME Corp. My friends: Darryl, Edna.
What’s Sematic Web: Google Snippet

Enhanced search result preview

Disclaimer: this preview is only shown as an example of what a search engine might display. It is to the discretion of each search engine provider to decide whether your page will be displayed as an enhanced search result or not in their search results pages.

Bob Smith
interstructured-data.org/examples/google-rs/people/md.html

Albuquerque, NM - engineer, ACME Corp

an actual search result may display other content relating to your search terms here.

Raw structured data extracted from the page:

<table>
<thead>
<tr>
<th>rdf:type</th>
<th>vmd:Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>vmd:address</td>
<td>rdf:type</td>
</tr>
<tr>
<td>vmd:locality</td>
<td>vmd:Address</td>
</tr>
<tr>
<td>vmd:region</td>
<td>ACME Corp</td>
</tr>
<tr>
<td>vmd:affiliation</td>
<td>Bob Smith</td>
</tr>
<tr>
<td>vmd:name</td>
<td>Smithy</td>
</tr>
<tr>
<td>vmd:nickname</td>
<td>engineer</td>
</tr>
<tr>
<td>vmd:url</td>
<td><a href="http://www.example.com">http://www.example.com</a></td>
</tr>
</tbody>
</table>

Parser statistics

| Statements | 12 |
| Templates  | http://data-vocabulary.org/Person
|            | http://data-vocabulary.org/Address |
What’s Semantic Web: Facebook Social Graph

Open Graph Stories
Post rich, structured stories from your app using a strongly typed API.

- Custom Stories in iOS: Publish custom stories from your iOS app.
- Custom Stories in Android: Publish custom stories from your Android app.
- Custom Stories on Web: Publish custom stories from your website or web app.

How It Works
People use stories to share the things they’re doing, the people they’re doing them with and the places where they happen. Let people share stories about your app on Facebook through a structured, strongly typed API.

Publishing Open Graph stories requires you to implement Facebook Login and request the publish_stream permission from users.
What's Semantic Web: From Two Perspectives

Expressiveness

More Semantic; More Powerful Reasoning

RDF, RDFS, OWL, OWL Full, OWL 2

… …

Open Linked Data, Web-scale Triple Store, Semantic Wiki

… …

How to get more data?
How to manage the Web-scale Semantic Data?
What’s Semantic Web: From Two Perspectives

Expressiveness

More Powerful Reasoning

Open Linked Data,
Web-scale Triple Store,
Semantic Wiki

More Interesting
Applications

Apple Siri,
Google Knowledge Graph;
IBM Watson;

How to get more data?
How to manage the Web-scale Semantic Data?

RDF, RDFS, OWL, OWL Full, OWL 2

More Areas

Broadcasting: BBC
Publishing: Thomson Reuters
Life: Eli Lilly and Company

More Areas

More Semantic;
More Powerful Reasoning
Some Interesting Products

IBM Watson
Some Interesting Products

EVI—acquired by Amazon on October 2012.

Some Interesting Products

Google Knowledge Graph

Barack Obama - Wikipedia, the free encyclopedia
Barack Obama is the 44th and current President of the United States. He is the first African American to hold the office. Born in Honolulu, Hawaii, ...
Enter: Barack Obama - Obama administration

News for Obama

Obama Embraces 'Obamacare,' Says It's Here to Stay
TIME - 2 hours ago
(Toledo, Ohio) — In his warm-up for the Democratic National Convention, President Barack Obama is tangling with a couple of rivals, only one ...
Obama heads for Isaac-soaked South, as convention delegates gather in North Carolina
Montreal Gazette - 57 minutes ago
Obama hits Romney on ObamaCare slam, says 'I do care'
The Associated Press - 11 hours ago

Barack Obama
Barack Obama.com is the official re-election campaign website of President Barack Obama. Visit the site for the latest updates from the Obama campaign. ...
Store - Contact Us - Jobs - Volunteer

President Barack Obama | The White House
The White House is the official residence and principal workplace of the President of the United States. It is located in Washington, D.C.

Books
Dreams from My Father
The Audacity of Hope
Of Thee I Sing 2010
Change We Can Believe In
Barack Obama in His Own...
RDF Uses

- Yago and DBPedia extract facts from Wikipedia & represent as RDF → structural queries
- Communities build RDF data
  - E.g., biologists: Bio2RDF and Uniprot RDF
- Web data integration
  - Linked Data Cloud
- ...
RDF Data Volumes . . .

- . . . are growing – and fast
  - Linked data cloud currently consists of 325 datasets with >25B triples
  - Size almost doubling every year
RDF Data Volumes . . .

- . . . are growing – and fast
  - Linked data cloud currently consists of 325 datasets with >25B triples
  - Size almost doubling every year

As of March 2009

March '09: 89 datasets

RDF Data Volumes . . .

- . . . are growing – and fast
  - Linked data cloud currently consists of 325 datasets with >25B triples
  - Size almost doubling every year

September ’10: 203 datasets

Linking Open Data cloud diagram, by Richard Cyganiak and Anja Jentzsch.
http://lod-cloud.net/
RDF Data Volumes . . .

- . . . are growing – and fast
  - Linked data cloud currently consists of 325 datasets with >25B triples
  - Size almost doubling every year

September ’11: 295 datasets

Linking Open Data cloud diagram, by Richard Cyganiak and Anja Jentzsch.
http://lod-cloud.net/
RDF Data Volumes . . .

- . . . are growing – and fast
  - Linked data cloud currently consists of 325 datasets with >25B triples
  - Size almost doubling every year

April ’14: 1091 datasets, ??? triples

Outline

RDF Introduction

gStore: a graph-based SPARQL query engine
Answering SPARQL queries using graph pattern matching [Zou et al., PVLDB 2011, VLDB J 2014]

gStore\textsuperscript{D}: a distributed SPARQL query engine
Answering SPARQL queries in a distributed environment [Peng et al., VLDB J 2016, EDBT 2016]
Outline

RDF Introduction

gStore: a graph-based SPARQL query engine
   Answering SPARQL queries using graph pattern matching [Zou et al., PVLDB 2011, VLDB J 2014]

gStore$^D$: a distributed SPARQL query engine
   Answering SPARQL queries in a distributed environment [Peng et al., VLDB J 2016, EDBT 2016]
RDF Introduction

- Everything is an **uniquely** named resource

http://en.wikipedia.org/wiki/Abraham_Lincoln


y:Abraham Lincoln

Abraham Lincoln:hasName "Abraham Lincoln"

Abraham Lincoln:BornOnDate: "1809-02-12"

Abraham Lincoln:DiedOnDate: "1865-04-15"

y:Washington DC

Abraham Lincoln:DiedIn
RDF Introduction

- Everything is an uniquely named resource
- Namespaces can be used to scope the names
RDF Introduction

- Everything is an **uniquely** named resource
- Namespaces can be used to scope the names
- Properties of resources can be defined

```xml
Abraham_Lincoln:hasName "Abraham Lincoln"
Abraham_Lincoln:BornOnDate: "1809-02-12"
Abraham_Lincoln:DiedOnDate: "1865-04-15"
```

Abraham Lincoln:hasName “Abraham Lincoln”
Abraham Lincoln:BornOnDate: “1809-02-12”
Abraham Lincoln:DiedOnDate: “1865-04-15”
RDF Introduction

- Everything is an **uniquely** named resource
- Namespaces can be used to scope the names
- Properties of resources can be defined
- Relationships with other resources can be defined
RDF Introduction

- Everything is an **uniquely** named resource
- Namespaces can be used to scope the names
- Properties of resources can be defined
- Relationships with other resources can be defined
- Resources can be contributed by different people/groups and can be located anywhere in the web
  - Integrated web “database”
RDF Data Model

- Triple: Subject, Predicate (Property), Object \((s, p, o)\)
  - **Subject**: the entity that is described (URI or blank node)
  - **Predicate**: a feature of the entity (URI)
  - **Object**: value of the feature (URI, blank node or literal)
- \((s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)\)
- Set of RDF triples is called an **RDF graph**

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbrahamLincoln</td>
<td>hasName</td>
<td>“Abraham Lincoln”</td>
</tr>
<tr>
<td>AbrahamLincoln</td>
<td>BornOnDate</td>
<td>“1809-02-12”</td>
</tr>
<tr>
<td>AbrahamLincoln</td>
<td>DiedOnDate</td>
<td>“1865-04-15”</td>
</tr>
</tbody>
</table>

\(U\): set of URIs
\(B\): set of blank nodes
\(L\): set of literals
<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>y: Abraham_Lincoln</td>
<td>hasName</td>
<td>&quot;Abraham Lincoln&quot;</td>
</tr>
<tr>
<td>y: Abraham_Lincoln</td>
<td>BornOnDate</td>
<td>&quot;1809-02-12&quot;</td>
</tr>
<tr>
<td>y: Abraham_Lincoln</td>
<td>DiedOnDate</td>
<td>&quot;1865-04-15&quot;</td>
</tr>
<tr>
<td>y: Abraham_Lincoln</td>
<td>bornIn</td>
<td>y: Hodgenville_KY</td>
</tr>
<tr>
<td>y: Abraham_Lincoln</td>
<td>DiedIn</td>
<td>y: Washington_DC</td>
</tr>
<tr>
<td>y: Abraham_Lincoln</td>
<td>title</td>
<td>&quot;President&quot;</td>
</tr>
<tr>
<td>y: Abraham_Lincoln</td>
<td>gender</td>
<td>&quot;Male&quot;</td>
</tr>
<tr>
<td>y: Washington_DC</td>
<td>hasName</td>
<td>&quot;Washington D.C.&quot;</td>
</tr>
<tr>
<td>y: Hodgenville_KY</td>
<td>hasName</td>
<td>&quot;Hodgenville&quot;</td>
</tr>
<tr>
<td>y: United_States</td>
<td>hasName</td>
<td>&quot;United States&quot;</td>
</tr>
<tr>
<td>y: United_States</td>
<td>foundingYear</td>
<td>&quot;1790&quot;</td>
</tr>
<tr>
<td>y: United_States</td>
<td>locatedIn</td>
<td>y: Washington_DC</td>
</tr>
<tr>
<td>y: Reese_Witherspoon</td>
<td>bornOnDate</td>
<td>&quot;1976-03-22&quot;</td>
</tr>
<tr>
<td>y: Reese_Witherspoon</td>
<td>bornIn</td>
<td>y: New_Orleans_LA</td>
</tr>
<tr>
<td>y: Reese_Witherspoon</td>
<td>hasName</td>
<td>&quot;Reese Witherspoon&quot;</td>
</tr>
<tr>
<td>y: Reese_Witherspoon</td>
<td>gender</td>
<td>&quot;Female&quot;</td>
</tr>
<tr>
<td>y: Reese_Witherspoon</td>
<td>title</td>
<td>&quot;Actress&quot;</td>
</tr>
<tr>
<td>y: New_Orleans_LA</td>
<td>foundingYear</td>
<td>&quot;1718&quot;</td>
</tr>
<tr>
<td>y: New_Orleans_LA</td>
<td>locatedIn</td>
<td>y: United_States</td>
</tr>
<tr>
<td>y: Franklin_Roosevelt</td>
<td>hasName</td>
<td>&quot;Franklin D. Roosevelt&quot;</td>
</tr>
<tr>
<td>y: Franklin_Roosevelt</td>
<td>bornIn</td>
<td>y: Hyde_Park_NY</td>
</tr>
<tr>
<td>y: Franklin_Roosevelt</td>
<td>title</td>
<td>&quot;President&quot;</td>
</tr>
<tr>
<td>y: Franklin_Roosevelt</td>
<td>gender</td>
<td>&quot;Male&quot;</td>
</tr>
<tr>
<td>y: Franklin_Roosevelt</td>
<td>foundingYear</td>
<td>&quot;1810&quot;</td>
</tr>
<tr>
<td>y: Hyde_Park_NY</td>
<td>locatedIn</td>
<td>y: United_States</td>
</tr>
<tr>
<td>y: Marilyn Monroe</td>
<td>hasName</td>
<td>&quot;Marilyn Monroe&quot;</td>
</tr>
<tr>
<td>y: Marilyn Monroe</td>
<td>bornOnDate</td>
<td>&quot;1926-07-01&quot;</td>
</tr>
<tr>
<td>y: Marilyn Monroe</td>
<td>diedOnDate</td>
<td>&quot;1962-08-05&quot;</td>
</tr>
</tbody>
</table>
RDF Query Model

- Query Model - **SPARQL Protocol and RDF Query Language**
- Given $U$ (set of URIs), $L$ (set of literals), and $V$ (set of variables), a SPARQL expression is defined recursively:
  - an atomic triple pattern, which is an element of
    $$(U \cup V) \times (U \cup V) \times (U \cup V \cup L)$$
  - $\texttt{?x hasName "Abraham Lincoln"}$
  - $P \texttt{FILTER } R$, where $P$ is a graph pattern expression and $R$ is a built-in SPARQL condition (i.e., analogous to a SQL predicate)
    - $\texttt{?x price ?p FILTER(?p < 30)}$
  - $P_1 \texttt{AND/OPT/UNION } P_2$, where $P_1$ and $P_2$ are graph pattern expressions
- Example:
  ```sparql
  SELECT ?name
  WHERE {
    ?m <bornIn> ?city. ?m <hasName> ?name.
    ?m<bornOnDate> ?bd. ?city <foundingYear> "1718".
    FILTER(regex(str(?bd),"1976"))
  }
  ```
SPARQL Queries

```
SELECT ?name
WHERE {
  ?m <bornIn> ?city. ?m <hasName> ?name.
  ?m<bornOnDate> ?bd. ?city <foundingYear> "1718".
  FILTER(regex(str(?bd),"1976"))
}
FILTER(regex(str(?bd),"1976"))
```
Naïve Triple Store Design

SELECT ?name
WHERE {
  ?m <bornIn> ?city. ?m <hasName> ?name.
  ?m<bornOnDate> ?bd. ?city <foundingYear> ‘‘1718’’.
  FILTER(regex(str(?bd), ‘‘1976’’))
}

<table>
<thead>
<tr>
<th>Subject</th>
<th>Property</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:Abraham_Lincoln</td>
<td>hasName</td>
<td>“Abraham Lincoln”</td>
</tr>
<tr>
<td>y:Abraham_Lincoln</td>
<td>bornOnDate</td>
<td>“1809-02-12”</td>
</tr>
<tr>
<td>y:Abraham_Lincoln</td>
<td>diedOnDate</td>
<td>“1865-04-15”</td>
</tr>
<tr>
<td>y:Hodgenville_KY</td>
<td>hasName</td>
<td>“Hodgenville”</td>
</tr>
<tr>
<td>y:Washington_DC</td>
<td>hasName</td>
<td>“Washington D.C.”</td>
</tr>
<tr>
<td>y:Washington_DC</td>
<td>foundingYear</td>
<td>“1790”</td>
</tr>
<tr>
<td>y:United_States</td>
<td>hasName</td>
<td>“United States”</td>
</tr>
<tr>
<td>y:United_States</td>
<td>hasCapital</td>
<td>“Washington_DC”</td>
</tr>
<tr>
<td>y:United_States</td>
<td>foundingYear</td>
<td>“1776”</td>
</tr>
<tr>
<td>y:Reese_Witherspoon</td>
<td>bornOnDate</td>
<td>“1976-03-22”</td>
</tr>
<tr>
<td>y:New_Orleans_LA</td>
<td>hasName</td>
<td>“Reese Witherspoon”</td>
</tr>
<tr>
<td>y:Reese_Witherspoon</td>
<td>gender</td>
<td>“Female”</td>
</tr>
<tr>
<td>y:Reese_Witherspoon</td>
<td>title</td>
<td>“Actress”</td>
</tr>
<tr>
<td>y:New_Orleans_LA</td>
<td>foundingYear</td>
<td>“1718”</td>
</tr>
<tr>
<td>y:Franklin_Roosevelt</td>
<td>hasName</td>
<td>“Franklin D. Roosevelt”</td>
</tr>
<tr>
<td>y:Franklin_Roosevelt</td>
<td>bornOnDate</td>
<td>“1901-07-03”</td>
</tr>
<tr>
<td>y:Franklin_Roosevelt</td>
<td>gender</td>
<td>“Male”</td>
</tr>
<tr>
<td>y:Hyde_Park_NY</td>
<td>foundingYear</td>
<td>“1810”</td>
</tr>
<tr>
<td>y:Hyde_Park_NY</td>
<td>locatedIn</td>
<td>“United States”</td>
</tr>
<tr>
<td>y:Marilyn_Monroe</td>
<td>gender</td>
<td>“Female”</td>
</tr>
<tr>
<td>y:Marilyn_Monroe</td>
<td>hasName</td>
<td>“Marilyn Monroe”</td>
</tr>
<tr>
<td>y:Marilyn_Monroe</td>
<td>bornOnDate</td>
<td>“1926-07-01”</td>
</tr>
<tr>
<td>y:Marilyn_Monroe</td>
<td>diedOnDate</td>
<td>“1962-08-05”</td>
</tr>
</tbody>
</table>
Naïve Triple Store Design

SELECT ?name
WHERE {
  ?m <bornIn> ?city. ?m <hasName> ?name.
  ?m <bornOnDate> ?bd. ?city <foundingYear> "'1718'".
  FILTER(regex(str(?bd), '1976'))
}

<table>
<thead>
<tr>
<th>Subject</th>
<th>Property</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:Abraham_Lincoln</td>
<td>hasName</td>
<td>&quot;Abraham Lincoln&quot;</td>
</tr>
<tr>
<td>y:Abraham_Lincoln</td>
<td>bornOnDate</td>
<td>&quot;1809-02-12&quot;</td>
</tr>
<tr>
<td>y:Abraham_Lincoln</td>
<td>diedOnDate</td>
<td>&quot;1865-04-15&quot;</td>
</tr>
<tr>
<td>y:Abraham_Lincoln</td>
<td>bornIn</td>
<td>y:Hodgenville_KY</td>
</tr>
<tr>
<td>y:Abraham_Lincoln</td>
<td>diedIn</td>
<td>y:Washington_DC</td>
</tr>
<tr>
<td>y:Abraham_Lincoln</td>
<td>title</td>
<td>&quot;President&quot;</td>
</tr>
<tr>
<td>y:Abraham_Lincoln</td>
<td>gender</td>
<td>&quot;Male&quot;</td>
</tr>
<tr>
<td>y:Washington_DC</td>
<td>hasName</td>
<td>&quot;Washington D.C.&quot;</td>
</tr>
<tr>
<td>y:Washington_DC</td>
<td>foundingYear</td>
<td>&quot;1790&quot;</td>
</tr>
<tr>
<td>y:Hodgenville_KY</td>
<td>hasName</td>
<td>&quot;Hodgenville&quot;</td>
</tr>
<tr>
<td>y:United_States</td>
<td>hasName</td>
<td>&quot;United States&quot;</td>
</tr>
<tr>
<td>y:United_States</td>
<td>hasCapital</td>
<td>y:Washington_DC</td>
</tr>
<tr>
<td>y:United_States</td>
<td>foundingYear</td>
<td>&quot;1776&quot;</td>
</tr>
<tr>
<td>y:United_States</td>
<td>locatedIn</td>
<td>y:United_States</td>
</tr>
<tr>
<td>y:Reese_Witherspoon</td>
<td>bornOnDate</td>
<td>y:New_Orleans_LA</td>
</tr>
<tr>
<td>y:Reese_Witherspoon</td>
<td>bornIn</td>
<td>&quot;Reese Witherspoon&quot;</td>
</tr>
<tr>
<td>y:Reese_Witherspoon</td>
<td>gender</td>
<td>&quot;Female&quot;</td>
</tr>
<tr>
<td>y:Reese_Witherspoon</td>
<td>title</td>
<td>&quot;Actress&quot;</td>
</tr>
<tr>
<td>y:New_Orleans_LA</td>
<td>foundingYear</td>
<td>&quot;1718&quot;</td>
</tr>
<tr>
<td>y:New_Orleans_LA</td>
<td>locatedIn</td>
<td>y:United_States</td>
</tr>
</tbody>
</table>
| y:Franklin_Roosevelt| hasName      | "Franklin D. Roo-
                         |    sevelt"          |
| y:Franklin_Roosevelt| bornIn       | y:Hyde_Park_NY     |
| y:Franklin_Roosevelt| title        | "President"        |
| y:Franklin_Roosevelt| gender       | "Male"             |
| y:Hyde_Park_NY      | foundingYear | "1810"             |
| y:Hyde_Park_NY      | locatedIn    | y:United_States    |
| y:Marilyn_Monroe    | hasName      | "Marilyn Monroe"   |
| y:Marilyn_Monroe    | bornOnDate   | "1926-07-01"       |
| y:Marilyn_Monroe    | diedOnDate   | "1962-08-05"       |

SELECT T2.object
FROM T as T1, T as T2, T as T3, T as T4
WHERE T1.property="bornIn"
AND T2.property="hasName"
AND T3.property="bornOnDate"
AND T1.subject=T2.subject
AND T2.subject=T3.subject
AND T4.property="foundingYear"
AND T1.object=T4.subject
AND T4.object="1718"
AND T3.object LIKE '%1976%'
Naïve Triple Store Design

```
SELECT ?name 
WHERE { 
  ?m <bornIn> ?city . ?m <hasName> ?name. 
  ?m<bornOnDate> ?bd . ?city <foundingYear> 
  FILTER(regex(str(?bd),"'1976'"))
}
```

SELECT T2.object 
FROM T as T1, T as T2, T as T3, T as T4 
WHERE T1.property="bornIn" 
AND T2.property="hasName" 
AND T3.property="bornOnDate" 
AND T1.subject=T2.subject 
AND T2.subject=T3.subject 
AND T4.property="foundingYear" 
AND T1.object=T4.subject 
AND T4.object="1718" 
AND T3.object LIKE '%$1976%'
Existing Solutions

1. Property table
   ▶ Each class of objects go to a different table ⇒ similar to normalized relations
   ▶ Eliminates some of the joins

2. Vertically partitioned tables
   ▶ For each property, build a two-column table, containing both subject and object, ordered by subjects
   ▶ Can use merge join (faster)
   ▶ Good for subject-subject joins but does not help with subject-object joins

3. Exhaustive indexing
   ▶ Create indexes for each permutation of the three columns
   ▶ Query components become range queries over individual relations with merge-join to combine
   ▶ Excessive space usage
Property Tables

- Grouping by entities; Jena [Wilkinson et al., 2003], FlexTable [Wang et al., 2010], DB2-RDF [Bornea et al., 2013]

- **Clustered property table**: group together the properties that tend to occur in the same (or similar) subjects

- **Property-class table**: cluster the subjects with the same type of property into one property table

<table>
<thead>
<tr>
<th>Subject</th>
<th>Property</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:Abraham_Lincoln</td>
<td>hasName</td>
<td>“Abraham Lincoln”</td>
</tr>
<tr>
<td></td>
<td>bornOnDate</td>
<td>“1809-02-12”</td>
</tr>
<tr>
<td></td>
<td>diedOnDate</td>
<td>“1865-04-15”</td>
</tr>
<tr>
<td>y:Washington_DC</td>
<td>hasName</td>
<td>“Washington D.C.”</td>
</tr>
<tr>
<td></td>
<td>foundingYear</td>
<td>“1790”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>hasName</th>
<th>foundingYear</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:Washington_DC</td>
<td>“Washington D.C.”</td>
<td>1790</td>
</tr>
<tr>
<td>y:Hyde_Park_NY</td>
<td>“Hyde Park”</td>
<td>1810'</td>
</tr>
</tbody>
</table>
Property Tables

- Grouping by entities; Jena [Wilkinson et al., 2003], FlexTable [Wang et al., 2010], DB2-RDF [Bornea et al., 2013]
- **Clustered property table**: group together the properties that tend to occur in the same (or similar) subjects
- **Property-class table**: cluster the subjects with the same type of property into one property table

### Advantages

- Fewer joins
- If the data is structured, we have a relational system – similar to normalized relations

<table>
<thead>
<tr>
<th>Subject</th>
<th>hasName</th>
<th>bornOnDate</th>
<th>diedOnDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:Abraham_Lincoln</td>
<td>“Abraham Lincoln”</td>
<td>1809-02-12</td>
<td>1865-04-15</td>
</tr>
<tr>
<td>y:Abraham_Lincoln</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y:Washington_DC</td>
<td>“Washington D.C.”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y:Washington_DC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>hasName</th>
<th>foundingYear</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:Washington_DC</td>
<td>“Washington D.C.”</td>
<td>1790</td>
</tr>
<tr>
<td>y:Hyde_Park_NY</td>
<td>“Hyde Park”</td>
<td>1810</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>hasName</th>
<th>foundingYear</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:Reese_Witherspoon</td>
<td>“Reese Witherspoon”</td>
<td>1976-03-22</td>
</tr>
</tbody>
</table>
Property Tables

- Grouping by entities; Jena [Wilkinson et al., 2003], FlexTable [Wang et al., 2010], DB2-RDF [Bornea et al., 2013]

- **Clustered property table**: group together the properties that tend to occur in the same (or similar) subjects

## Advantages

- Fewer joins
- If the data is structured, we have a relational system – similar to normalized relations

## Disadvantages

- Potentially a lot of NULLs
- Clustering is not trivial
- Multi-valued properties are complicated

<table>
<thead>
<tr>
<th>Subject</th>
<th>hasName</th>
<th>bornOnDate</th>
<th>diedOnDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:Abraham_Lincoln</td>
<td>“Abraham Lincoln”</td>
<td>1809-02-12</td>
<td>1865-04-15</td>
</tr>
<tr>
<td>y:Reese_Witherspoon</td>
<td>“Reese Witherspoon”</td>
<td>1976-03-22</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>hasName</th>
<th>foundingYear</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:Washington_DC</td>
<td>“Washington D.C.”</td>
<td>1790</td>
</tr>
<tr>
<td>y:Hyde_Park_NY</td>
<td>“Hyde Park”</td>
<td>1810’</td>
</tr>
</tbody>
</table>
Binary Tables

- Grouping by properties: For each property, build a two-column table, containing both subject and object, ordered by subjects [Abadi et al., 2009]
- Also called vertical partitioned tables
- \( n \) two column tables (\( n \) is the number of unique properties in the data)

<table>
<thead>
<tr>
<th>Subject</th>
<th>Property</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:Abraham_Lincoln</td>
<td>hasName</td>
<td>“Abraham Lincoln”</td>
</tr>
<tr>
<td>y:Abraham_Lincoln</td>
<td>bornOnDate</td>
<td>“1809-02-12”</td>
</tr>
<tr>
<td>y:Abraham_Lincoln</td>
<td>diedOnDate</td>
<td>“1865-04-15”</td>
</tr>
<tr>
<td>y:Washington_DC</td>
<td>hasName</td>
<td>“Washington D.C.”</td>
</tr>
<tr>
<td>y:Washington_DC</td>
<td>foundingYear</td>
<td>“1790”</td>
</tr>
</tbody>
</table>

**Subject** | **Object**
---|---
y:Abraham_Lincoln | “Abraham Lincoln”
y:Washington_DC | “Washington D.C.”

**Subject** | **Object**
---|---
y:Abraham_Lincoln | 1809-02-12
y:Reese_Witherspoon | 1976-03-22

**Subject** | **Object**
---|---
y:Washington_DC | 1790
y:Hyde_Park_NY | 1810
Binary Tables

- Grouping by properties: For each property, build a two-column table, containing both subject and object, ordered by subjects.

Advantages

- Supports multi-valued properties
- No NULLs
- No clustering
- Read only needed attributes (i.e. less I/O)
- Good performance for subject-subject joins

```
<table>
<thead>
<tr>
<th>Subject</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:Abraham_Lincoln</td>
<td>&quot;Abraham Lincoln&quot;</td>
</tr>
<tr>
<td>y:Washington_DC</td>
<td>&quot;Washington D.C.&quot;</td>
</tr>
<tr>
<td>y:Washington_DC</td>
<td>...</td>
</tr>
<tr>
<td>y:Reese_Witherspoon</td>
<td>1976-03-22</td>
</tr>
<tr>
<td>Subject</td>
<td>Object</td>
</tr>
<tr>
<td>y:Abraham_Lincoln</td>
<td>1809-02-12</td>
</tr>
<tr>
<td>y:Washington_DC</td>
<td>...</td>
</tr>
<tr>
<td>Subject</td>
<td>Founding Year</td>
</tr>
<tr>
<td>y:Washington_DC</td>
<td>“1790”</td>
</tr>
<tr>
<td>y:Hyde_Park_NY</td>
<td>1810</td>
</tr>
</tbody>
</table>
```

Advantages

- Supports multi-valued properties
- No NULLs
- No clustering
- Read only needed attributes (i.e. less I/O)
- Good performance for subject-subject joins
Binary Tables

- Grouping by properties: For each property, build a two-column table, containing both subject and object, ordered by subjects.

Advantages

- Supports multi-valued properties
- No NULLs
- No clustering
- Read only needed attributes (i.e. less I/O)
- Good performance for subject-subject joins

Disadvantages

- Not useful for subject-object joins
- Expensive inserts

<table>
<thead>
<tr>
<th>Subject</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:Abraham_Lincoln</td>
<td>“Abraham Lincoln”</td>
</tr>
<tr>
<td>y:Washington_DC</td>
<td>“Washington D.C.”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:Abraham_Lincoln</td>
<td>1809-02-12</td>
</tr>
<tr>
<td>y:Reese_Witherspoon</td>
<td>1976-03-22</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:Washington_DC</td>
<td>1790</td>
</tr>
<tr>
<td>y:Hyde_Park_NY</td>
<td>1810</td>
</tr>
</tbody>
</table>
Exhaustive Indexing

- RDF-3X [Neumann and Weikum, 2008], Hexastore [Weiss et al., 2008]
- Strings are mapped to ids using a mapping table

Original triple table

<table>
<thead>
<tr>
<th>Subject</th>
<th>Property</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:Abraham_Lincoln</td>
<td>hasName</td>
<td>“Abraham Lincoln”</td>
</tr>
<tr>
<td>y:Abraham_Lincoln</td>
<td>bornOnDate</td>
<td>“1809-02-12”</td>
</tr>
<tr>
<td>y:Abraham_Lincoln</td>
<td>diedOnDate</td>
<td>“1865-04-15”</td>
</tr>
<tr>
<td>y:Washington_DC</td>
<td>hasName</td>
<td>“Washington D.C.”</td>
</tr>
<tr>
<td>y:Washington_DC</td>
<td>foundingYear</td>
<td>“1790”</td>
</tr>
</tbody>
</table>

Encoded triple table

<table>
<thead>
<tr>
<th>Subject</th>
<th>Property</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Mapping table

<table>
<thead>
<tr>
<th>ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>y:Abraham_Lincoln</td>
</tr>
<tr>
<td>1</td>
<td>hasName</td>
</tr>
<tr>
<td>2</td>
<td>“Abraham Lincoln”</td>
</tr>
<tr>
<td>3</td>
<td>bornOnDate</td>
</tr>
<tr>
<td>4</td>
<td>“1809-02-12”</td>
</tr>
<tr>
<td>5</td>
<td>diedOnDate</td>
</tr>
<tr>
<td>6</td>
<td>“1865-04-15”</td>
</tr>
<tr>
<td>7</td>
<td>y:Washington_DC</td>
</tr>
<tr>
<td>8</td>
<td>“Washington D.C.”</td>
</tr>
<tr>
<td>9</td>
<td>foundingYear</td>
</tr>
<tr>
<td>10</td>
<td>“1790”</td>
</tr>
</tbody>
</table>
Exhaustive Indexing

- RDF-3X [Neumann and Weikum, 2008], Hexastore [Weiss et al., 2008]
- Strings are mapped to ids using a mapping table
- Triples are indexed in a clustered B+ tree in lexicographic order

<table>
<thead>
<tr>
<th>Subject</th>
<th>Property</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Easy querying through mapping table
Exhaustive Indexing

- RDF-3X [Neumann and Weikum, 2008], Hexastore [Weiss et al., 2008]
- Strings are mapped to ids using a mapping table
- Triples are indexed in a clustered B+ tree in lexicographic order
- Create indexes for permutations of the three columns: SPO, SOP, PSO, POS, OPS, OSP

<table>
<thead>
<tr>
<th>Subject</th>
<th>Property</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Easy querying through mapping table
Exhaustive Indexing–Query Execution

- Each triple pattern can be answered by a range query
- Joins between triple patterns computed using merge join
- Join order is easy due to extensive indexing

<table>
<thead>
<tr>
<th>Subject</th>
<th>Property</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>y:Abraham_Lincoln</td>
</tr>
<tr>
<td>1</td>
<td>hasName</td>
</tr>
<tr>
<td>2</td>
<td>“Abraham Lincoln”</td>
</tr>
<tr>
<td>3</td>
<td>bornOnDate</td>
</tr>
<tr>
<td>4</td>
<td>“1809-02-12”</td>
</tr>
<tr>
<td>5</td>
<td>diedOnDate</td>
</tr>
<tr>
<td>6</td>
<td>“1865-04-15”</td>
</tr>
<tr>
<td>7</td>
<td>y:Washington_DC</td>
</tr>
<tr>
<td>8</td>
<td>“Washington D.C.”</td>
</tr>
<tr>
<td>9</td>
<td>foundingYear</td>
</tr>
<tr>
<td>10</td>
<td>“1790”</td>
</tr>
</tbody>
</table>
Exhaustive Indexing–Query Execution

- Each triple pattern can be answered by a range query
- Joins between triple patterns computed using merge join
- Join order is easy due to extensive indexing

Advantages

- Eliminates some of the joins – they become range queries
- Merge join is easy and fast

<table>
<thead>
<tr>
<th>Subject</th>
<th>Property</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>bornOnDate</td>
</tr>
<tr>
<td>4</td>
<td>“1809-02-12”</td>
</tr>
<tr>
<td>5</td>
<td>diedOnDate</td>
</tr>
<tr>
<td>6</td>
<td>“1865-04-15”</td>
</tr>
<tr>
<td>7</td>
<td>y:Washington_D.C.</td>
</tr>
<tr>
<td>8</td>
<td>“Washington D.C.”</td>
</tr>
<tr>
<td>9</td>
<td>foundingYear</td>
</tr>
<tr>
<td>10</td>
<td>“1790”</td>
</tr>
</tbody>
</table>
Exhaustive Indexing–Query Execution

- Each triple pattern can be answered by a range query
- Joins between triple patterns computed using merge join
- Join order is easy due to extensive indexing

Advantages

-Eliminates some of the joins – they become range queries
-Merge join is easy and fast

Disadvantages

- Space usage

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>“1865-04-15”</td>
</tr>
<tr>
<td>7</td>
<td>y:Washington_DC</td>
</tr>
<tr>
<td>8</td>
<td>“Washington D.C.”</td>
</tr>
<tr>
<td>9</td>
<td>foundingYear</td>
</tr>
<tr>
<td>10</td>
<td>“1790”</td>
</tr>
</tbody>
</table>
RDF Introduction

**gStore**: a graph-based SPARQL query engine
Answering SPARQL queries using graph pattern matching [Zou et al., PVLDB 2011, VLDB J 2014]

**gStore^D**: a distributed SPARQL query engine
Answering SPARQL queries in a distributed environment [Peng et al., VLDB J 2016, EDBT 2016]
gStore [Zou et al., VLDB 11], [Zou et al., VLDB J 14] – General Idea

- We work directly on the RDF graph and the SPARQL query graph
  - Answering SPARQL query \(\equiv\) subgraph matching
  - Subgraph matching is computationally expensive
- Use a signature-based encoding of each entity and class vertex to speed up matching
- Filter-and-evaluate
  - Use a false positive algorithm to prune nodes and obtain a set of candidates; then do more detailed evaluation on those
- We develop an index (VS*-tree) over the data signature graph (has light maintenance load) for efficient pruning
0. Start with RDF Graph $G$

FILTER(\text{regex}(\text{str(?bd)}, "1976" ))
0. Start with RDF Graph $G$

Finding matches over a large graph is not a trivial task!
Outline

RDF Introduction

gStore: a graph-based SPARQL query engine
Answering SPARQL queries using graph pattern matching [Zou et al., PVLDB 2011, VLDB J 2014]

\[ gStore^D: \text{a distributed SPARQL query engine} \]
Answering SPARQL queries in a distributed environment [Peng et al., VLDB J 2016, EDBT 2016]
gStore\textsuperscript{D}—A Distributed RDF Graph Store System [Peng Peng et al., VLDB J 16] – General Idea

- We adopt a two-step framework—“partial evaluation and assembly”:
  - Each RDF graph fragment is resident in each site.
  - **Partial Evaluation**: send the whole SPARQL query graph to each fragment and figure out the local partial matches in each fragment.
  - **Assembly**: combine these local partial matches to form complete matches.
- Use graph structure-based pruning to speed up matching.
- Propose two kinds of assembly methods—the centralized and the distributed strategies.
- Our method *is orthogonal to* any graph partition approach.
- The system is scalable to RDF data sizes and the number of machines.
A Distributed RDF Graph
A Distributed RDF Graph
A Distributed RDF Graph
Background: Partial Evaluation

\[ f(x) \Rightarrow f(s, d) \Rightarrow f''(f'(s), d) \Rightarrow \text{Final Answer} \]
Background: Partial Evaluation

\[ f(x) \Rightarrow f(s, d) \Rightarrow f''(f'(s), d) \Rightarrow \text{Final Answer} \]
Background: Partial Evaluation

\[
f(x) \Rightarrow f(s, d) \Rightarrow f''(f'(s), d) \Rightarrow \text{Final Answer}
\]

known inputs  unknown inputs
Background: Partial Evaluation

\[ f(x) \Rightarrow f(s, d) \Rightarrow f''(f'(s), d) \Rightarrow \text{Final Answer} \]

- known inputs
- unknown inputs
- partial results
What’s the known input in the distributed RDF graph $G$?
Fragment $F_i$ in the local site together with extended vertices.
What's the **known input** in the distributed RDF graph $G$?
Fragment $F_i$ in the local site together with *extended vertices*.
Partial Evaluation: in Distributed RDF Graph

What’s the known input in the distributed RDF graph $G$?

Fragment $F_i$ in the local site together with extended vertices.

Fragment $F_1$

Extended vertices

<table>
<thead>
<tr>
<th>s1:act4</th>
<th>Hank Azaria</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref:label</td>
<td></td>
</tr>
</tbody>
</table>

| s1:fil7 |
| Mystery Men |
| ref:label |

| s1:dir1 |
| Woody Allen |
| rdfs:label |

| s1:fil1 |
| Sleeper |
| rdfs:label |

| s1:fil2 |
| Small Time Crooks |
| rdfs:label |

<table>
<thead>
<tr>
<th>s3:act3</th>
<th>hasWonPrize</th>
</tr>
</thead>
<tbody>
<tr>
<td>s3:awa1</td>
<td></td>
</tr>
</tbody>
</table>

| s2:act1 |
| A Very Yong Lady |
| rdfs:label |

| s3:act2 |
| s1:fil5 |
| rdfs:label |

| v1 | actedIn |
| ?a |

| v2 | directed |
| ?d |

| v3 | ref:label |
| ?f1 |

| v4 | ref:label |
| ?f2 |

| v5 | ref:label |
| ?n1 |

| v6 | ref:label |
| ?n2 |
Partial Evaluation: in Distributed RDF Graph

What’s the known input in the distributed RDF graph \( G \)?

Fragment \( F_i \) in the local site together with extended vertices.

Fragment \( F_1 \)

Extended vertices
Partial Evaluation: in Distributed RDF Graph

What’s the known input in the distributed RDF graph $G$?
Fragment $F_i$ in the local site together with extended vertices.

[Diagram showing a graph with labeled nodes and edges representing relationships such as "actedIn", "isMarriedTo", and labels like "Sleeper", "Woody Allen", "Mystery Men", "Sleeper", "Woody Allen", "Mystery Men", etc.]

Extended vertices
Partial Evaluation: in Distributed RDF Graph

What’s the **known input** in the distributed RDF graph \( G \)?

Fragment \( F_i \) in the local site together with *extended vertices*.
Assembly: An Example
Assembly: An Example

Sharing Common Crossing Edges

Sleeper
rdfs:label
s1:fil1
directed
s1:dir1
s1:dir1
directed
s2:act1
directed
s1:dir1
s2:act1
directed
Mary Hartman
rdfs:label
s2:fil3
Assembly: An Example

Sharing Common Crossing Edges

Sleeper

s1:fil1

Sleeper

s1:fil1

Mary Hartman

s2:act1

s1:dir1

directed

s2:fil3

Mary Hartman

s2:fil3

s2:act1

s1:dir1

directed
System Framework: Overview

SPARQL Query $Q$ → Local Matches in site $S_1$ → Local Matches in site $S_2$ → Local Matches in site $S_3$ → Assemble All Local Partial Matches → SPARQL answers
System Framework: Overview

SPARQL Query $Q$

Local Matches in site $S_1$

Local Matches in site $S_2$

Local Matches in site $S_3$

Assemble All Local Partial Matches

SPARQL answers
System Framework: Overview

SPARQL Query $Q$

- Local Matches in site $S_1$
- Local Matches in site $S_2$
- Local Matches in site $S_3$

Assemble All Local Partial Matches

SPARQL answers

Assembly
Local Evaluation—Definition

(Local Partial Match) Given a SPARQL query graph $Q$ with $n$ vertices \( \{v_1, \ldots, v_n\} \) and a connected subgraph $PM$ with $m$ vertices \( \{u_1, \ldots, u_m\} \) \((m \leq n)\) in a fragment $F_k = (V_k \cup V^e_k, E_k \cup E^c_k, \Sigma_k)$, $PM$ is a local partial match in fragment $F_k$ if and only if there exists a function $f : \{v_1, \ldots, v_n\} \rightarrow \{u_1, \ldots, u_m\} \cup \{NULL\}$, where the following conditions hold:

1. If $v_i$ is not a variable, $f(v_i)$ and $v_i$ have the same URI or literal or $f(v_i) = NULL$.
2. If $v_i$ is a variable, $f(v_i) \in \{u_1, \ldots, u_m\}$ or $f(v_i) = NULL$.
3. If there exists an edge $\rightarrow v_i v_j$ in $Q$ \((1 \leq i \neq j \leq n)\), there also exists an edge $\rightarrow \rightarrow f(v_i) f(v_j)$ in $PM$ with property $p$ unless $f(v_i)$ and $f(v_j)$ are both in $V^e_k$ or $f(v_i) = NULL \lor f(v_j) = NULL$.
4. $PM$ contains at least one crossing edge.
5. If $f(v_i) \in V_k$ \((i.e., f(v_i) is an internal vertex in F_k)\) and $\exists \rightarrow v_i v_j \in Q$ \((or \rightarrow v_j v_i \in Q)\), there must exist $f(v_j) \neq NULL$ and $\exists \rightarrow \rightarrow f(v_i) f(v_j) \in PM$ \((or \exists \rightarrow \rightarrow f(v_j) f(v_i) \in PM)\). Furthermore, if $\rightarrow v_i v_j$ \((or \rightarrow v_j v_i)\) has a property $p$, $\rightarrow \rightarrow f(v_i) f(v_j)$ \((or \rightarrow \rightarrow f(v_j) f(v_i))\) has the same property $p$.
6. Any two vertices $v_i$ and $v_j$ \((in query Q)\), where $f(v_i)$ and $f(v_j)$ are both internal vertices in $PM$, are weakly connected in $Q$. 
Local Evaluation—Definition

(Local Partial Match) Given a SPARQL query graph $Q$ with $n$ vertices $\{v_1, ..., v_n\}$ and a connected subgraph $PM$ with $m$ vertices $\{u_1, ..., u_m\}$ ($m \leq n$) in a fragment $F_k = (V_k \cup V^e_k, E_k \cup E^c_k, \Sigma_k)$, $PM$ is a local partial match in fragment $F_k$ if and only if there exists a function $f : \{v_1, ..., v_n\} \rightarrow \{u_1, ..., u_m\} \cup \{NULL\}$, where the following conditions hold:

1. If $v_i$ is not a variable, $f(v_i)$ and $v_i$ have the same URI or literal or $f(v_i) = NULL$. 


Local Evaluation—Definition

(Local Partial Match) Given a SPARQL query graph \(Q\) with \(n\) vertices \(\{v_1, \ldots, v_n\}\) and a connected subgraph \(PM\) with \(m\) vertices \(\{u_1, \ldots, u_m\}\) \((m \leq n)\) in a fragment \(F_k = (V_k \cup V^e_k, E_k \cup E^c_k, \Sigma_k)\), \(PM\) is a local partial match in fragment \(F_k\) if and only if there exists a function \(f : \{v_1, \ldots, v_n\} \rightarrow \{u_1, \ldots, u_m\} \cup \{NULL\}\), where the following conditions hold:

1. If \(v_i\) is not a variable, \(f(v_i)\) and \(v_i\) have the same URI or literal or \(f(v_i) = NULL\).
2. If \(v_i\) is a variable, \(f(v_i) \in \{u_1, \ldots, u_m\}\) or \(f(v_i) = NULL\).
**Local Evaluation—Definition**

**(Local Partial Match)** Given a SPARQL query graph \( Q \) with \( n \) vertices \( \{v_1, ..., v_n\} \) and a connected subgraph \( PM \) with \( m \) vertices \( \{u_1, ..., u_m\} \) \((m \leq n)\) in a fragment \( F_k = (V_k \cup V^e_k, E_k \cup E^c_k, \Sigma_k) \), \( PM \) is a *local partial match* in fragment \( F_k \) if and only if there exists a function \( f : \{v_1, ..., v_n\} \rightarrow \{u_1, ..., u_m\} \cup \{NULL\} \), where the following conditions hold:

1. If \( v_i \) is not a variable, \( f(v_i) \) and \( v_i \) have the same URI or literal or \( f(v_i) = NULL \).
2. If \( v_i \) is a variable, \( f(v_i) \in \{u_1, ..., u_m\} \) or \( f(v_i) = NULL \).
3. If there exists an edge \( \overrightarrow{v_i v_j} \) in \( Q \) \((1 \leq i \neq j \leq n)\), there also exists an edge \( \overrightarrow{f(v_i) f(v_j)} \) in \( PM \) with property \( p \) unless \( f(v_i) \) and \( f(v_j) \) are both in \( V^e_k \) or \( f(v_i) = NULL \lor f(v_j) = NULL \). Furthermore, \( \overrightarrow{f(v_i) f(v_j)} \) has the same property as \( \overrightarrow{v_i v_j} \) unless the label of \( \overrightarrow{v_i v_j} \) is a variable.
Local Evaluation—Definition

(Local Partial Match) Given a SPARQL query graph $Q$ with $n$ vertices $\{v_1, \ldots, v_n\}$ and a connected subgraph $PM$ with $m$ vertices $\{u_1, \ldots, u_m\}$ ($m \leq n$) in a fragment $F_k = (V_k \cup V^e_k, E_k \cup E^c_k, \Sigma_k)$, $PM$ is a local partial match in fragment $F_k$ if and only if there exists a function $f : \{v_1, \ldots, v_n\} \rightarrow \{u_1, \ldots, u_m\} \cup \{NULL\}$, where the following conditions hold:

1. If $v_i$ is not a variable, $f(v_i)$ and $v_i$ have the same URI or literal or $f(v_i) = NULL$.
2. If $v_i$ is a variable, $f(v_i) \in \{u_1, \ldots, u_m\}$ or $f(v_i) = NULL$.
3. If there exists an edge $\overrightarrow{v_i v_j}$ in $Q$ ($1 \leq i \neq j \leq n$), there also exists an edge $\overrightarrow{f(v_i) f(v_j)}$ in $PM$ with property $p$ unless $f(v_i)$ and $f(v_j)$ are both in $V^e_k$ or $f(v_i) = NULL \lor f(v_j) = NULL$. Furthermore, $\overrightarrow{f(v_i) f(v_j)}$ has the same property as $\overrightarrow{v_i v_j}$ unless the label of $\overrightarrow{v_i v_j}$ is a variable.
4. $PM$ contains at least one crossing edge.
Local Evaluation—Definition

(Local Partial Match) Given a SPARQL query graph $Q$ with $n$ vertices \{$v_1, ..., v_n$\} and a connected subgraph $PM$ with $m$ vertices \{$u_1, ..., u_m$\} ($m \leq n$) in a fragment $F_k = (V_k \cup V^e_k, E_k \cup E^c_k, \Sigma_k)$, $PM$ is a local partial match in fragment $F_k$ if and only if there exists a function $f : \{v_1, ..., v_n\} \rightarrow \{u_1, ..., u_m\} \cup \{NULL\}$, where the following conditions hold:

1. If $v_i$ is not a variable, $f(v_i)$ and $v_i$ have the same URI or literal or $f(v_i) = NULL$.
2. If $v_i$ is a variable, $f(v_i) \in \{u_1, ..., u_m\}$ or $f(v_i) = NULL$.
3. If there exists an edge $\overline{v_i}v_j$ in $Q$ ($1 \leq i \neq j \leq n$), there also exists an edge $\overline{f(v_i)}f(v_j)$ in $PM$ with property $p$ unless $f(v_i)$ and $f(v_j)$ are both in $V^e_k$ or $f(v_i) = NULL \lor f(v_j) = NULL$. Furthermore, $\overline{f(v_i)}f(v_j)$ has the same property as $\overline{v_i}v_j$ unless the label of $\overline{v_i}v_j$ is a variable.
4. $PM$ contains at least one crossing edge.
5. If $f(v_i) \in V_k$ (i.e., $f(v_i)$ is an internal vertex in $F_k$) and $\exists \overline{v_i}v_j \in Q$ (or $\overline{v_j}v_i \in Q$), there must exist $f(v_j) \neq NULL$ and $\exists \overline{f(v_i)}f(v_j) \in PM$ (or $\exists \overline{f(v_j)}f(v_i) \in PM$). Furthermore, if $\overline{v_i}v_j$ (or $\overline{v_j}v_i$) has a property $p$, $\overline{f(v_i)}f(v_j)$ (or $\overline{f(v_j)}f(v_i)$) has the same property $p$. 
Local Evaluation—Definition

(Local Partial Match) Given a SPARQL query graph $Q$ with $n$ vertices \{${v_1, \ldots, v_n}$\} and a connected subgraph $PM$ with $m$ vertices \{${u_1, \ldots, u_m}$\} ($m \leq n$) in a fragment $F_k = (V_k \cup V^e_k, E_k \cup E^c_k, \Sigma_k)$, $PM$ is a local partial match in fragment $F_k$ if and only if there exists a function $f : \{v_1, \ldots, v_n\} \rightarrow \{u_1, \ldots, u_m\} \cup \{NULL\}$, where the following conditions hold:

1. If $v_i$ is not a variable, $f(v_i)$ and $v_i$ have the same URI or literal or $f(v_i) = NULL$.
2. If $v_i$ is a variable, $f(v_i) \in \{u_1, \ldots, u_m\}$ or $f(v_i) = NULL$.
3. If there exists an edge $\overrightarrow{v_i v_j}$ in $Q$ ($1 \leq i \neq j \leq n$), there also exists an edge $\overrightarrow{f(v_i) f(v_j)}$ in $PM$ with property $p$ unless $f(v_i)$ and $f(v_j)$ are both in $V^e_k$ or $f(v_i) = NULL \lor f(v_j) = NULL$. Furthermore, $\overrightarrow{f(v_i) f(v_j)}$ has the same property as $\overrightarrow{v_i v_j}$ unless the label of $\overrightarrow{v_i v_j}$ is a variable.
4. $PM$ contains at least one crossing edge.
5. If $f(v_i) \in V_k$ (i.e., $f(v_i)$ is an internal vertex in $F_k$) and $\exists \overrightarrow{v_i v_j} \in Q$ (or $\overrightarrow{v_j v_i} \in Q$), there must exist $f(v_i) \neq NULL$ and $\exists f(v_i) f(v_j) \in PM$ (or $\exists f(v_j)f(v_i) \in PM$). Furthermore, if $\overrightarrow{v_i v_j}$ (or $\overrightarrow{v_j v_i}$) has a property $p$, $\overrightarrow{f(v_i) f(v_j)}$ (or $\overrightarrow{f(v_j) f(v_i)}$) has the same property $p$.
6. Any two vertices $v_i$ and $v_j$ (in query $Q$), where $f(v_i)$ and $f(v_j)$ are both internal vertices in $PM$, are weakly connected in $Q$. 
Finding Local Partial Matches (step by step exploration)

Fragment $F_1$

Hank Azaria

Sleeper

Mystery Men

Small Time Crooks

Woody Allen

A Very Yong Lady

Directed vertices

Extended vertices

Local Partial Match
Finding Local Partial Matches (step by step exploration)
Finding Local Partial Matches (step by step exploration)

Fragment $F_1$

Local Partial Match

Extended vertices
Finding Local Partial Matches (step by step exploration)
Finding Local Partial Matches (step by step exploration)

Fragment $F_1$

- Hank Azaria
- Mystery Men
- Woody Allen
- Sleeper
- Small Time Crooks
- A Very Yong Lady

Local Partial Match

Extended vertices
Assembly—Definition

Definition

(Joinable) Given a query graph $Q$ and two fragments $F_i$ and $F_j$ ($i \neq j$), let $PM_i$ and $PM_j$ be two local partial matches over fragments $F_i$ and $F_j$ under functions $f_i$ and $f_j$, respectively. $PM_i$ and $PM_j$ are joinable if and only if the following conditions hold:
Definition

(Joinable) Given a query graph $Q$ and two fragments $F_i$ and $F_j$ ($i \neq j$), let $PM_i$ and $PM_j$ be two local partial matches over fragments $F_i$ and $F_j$ under functions $f_i$ and $f_j$, respectively. $PM_i$ and $PM_j$ are joinable if and only if the following conditions hold:

1. There exist no internal vertices $u$ and $u'$ in $PM_i$ and $PM_j$, respectively, such that $f_i^{-1}(u) = f_j^{-1}(u')$;
Definition (Joinable) Given a query graph $Q$ and two fragments $F_i$ and $F_j$ ($i \neq j$), let $PM_i$ and $PM_j$ be two local partial matches over fragments $F_i$ and $F_j$ under functions $f_i$ and $f_j$, respectively. $PM_i$ and $PM_j$ are joinable if and only if the following conditions hold:

1. There exist no internal vertices $u$ and $u'$ in $PM_i$ and $PM_j$, respectively, such that $f_i^{-1}(u) = f_j^{-1}(u')$;

2. There exists at least one crossing edge $\overrightarrow{uu'}$ such that $u$ is an internal vertex and $u'$ is an extended vertex in $F_i$, while $u$ is an extended vertex and $u'$ is an internal vertex in $F_j$. Furthermore, $f_i^{-1}(u) = f_j^{-1}(u)$ and $f_i^{-1}(u') = f_j^{-1}(u')$. 

Assembly: An Example
Assembly: An Example

Sharing Common Crossing Edges

Sleeper

s1:fil1 directed

s1:dir1 directed s2:act1

s1:dir1 directed s2:act1

Mary Hartman rdfs:label

s2:fil3
Assembly: An Example

Sharing Common Crossing Edges

Sleeper
  rdfs:label

s1:fil1  directed  s1:dir1  directed  s2:act1

s1:dir1  directed  s2:act1

Mary Hartman  rdfs:label  s2:fil3

Sleeper
  rdfs:label

s1:fil1  directed  s1:dir1  directed  s2:act1

s2:fil3

Mary Hartman  rdfs:label
Centralized Assembly—An Iterative Algorithm
Any two matches in the same group cannot be joinable.

Matching to $v_1$

$PM_2^1$  
\[
[002,001,005,\text{NULL},\text{NULL},\text{NULL}]
\]

$PM_2^2$  
\[
[002,001,009,\text{NULL},018,\text{NULL}]
\]

$PM_2^3$  
\[
[002,001,010,\text{NULL},019,\text{NULL}]
\]

$PM_3^1$  
\[
[004,011,003,\text{NULL},\text{NULL},\text{NULL}]
\]
Optimization—Partitioning-based Join Processing

\[
PM_1^1 [002, NULL, 005, NULL, 027, NULL] ; PM_2^1 [002, 001, 005, NULL, NULL, NULL] ; PM_3^1 [004, 011, 003, NULL, NULL, NULL] ; PM_4^1 [004, 011, 003, NULL, NULL, NULL] ; PM_5^1 [006, NULL, 008, NULL, 015, NULL] ; PM_2^2 [002, 001, NULL, 007, NULL, 014] ; PM_2^3 [002, 001, NULL, 008, NULL, 015] ; PM_2^4 [004, 003, NULL, 012, NULL, 022] ; PM_2^3 [002, 001, 010, NULL, 019, NULL] ;
\]
Optimization—Partitioning-based Join Processing

Definition

**Join Cost.** Given a query graph $Q$ with $n$ vertices $v_1, \ldots, v_n$ and a partitioning $\mathcal{P} = \{ P_{v_1}, \ldots, P_{v_n} \}$ over all local partial matches $\Omega$, the join cost is defined as follows.

$$\text{Cost}(\Omega) = O\left( \prod_{i=1}^{i=n} (|P_{v_i}| + 1) \right)$$ (1)

where $|P_{v_i}|$ is the number of local partial matches in $P_{v_i}$ and 1 is introduced to avoid the “0” element in the product.
Optimization—Partitioning-based Join Processing

Definition

**Join Cost.** Given a query graph $Q$ with $n$ vertices $v_1,\ldots,v_n$ and a partitioning $\mathcal{P} = \{P_{v_1},\ldots,P_{v_n}\}$ over all local partial matches $\Omega$, the join cost is defined as follows.

$$\text{Cost}(\Omega) = O\left(\prod_{i=1}^{i=n} (|P_{v_i}| + 1)\right)$$  \hspace{1cm} (1)

where $|P_{v_i}|$ is the number of local partial matches in $P_{v_i}$ and 1 is introduced to avoid the “0” element in the product.

Theorem

*Finding the optimal partitioning is NP-complete problem.*

Proof.

It can be proven by reducing 0-1 integer planning problem to finding the optimal partitioning.  \hfill \Box
Distributed Assembly—Background: BSP model

Processors

Local Computation

Communication

Barrier Synchronisation
Distributed Assembly—Background: BSP model

\[ \Delta_m^m(F_i) \]

\[ \Omega^m(F_i) \]

\[ \Delta_{out}^m(F_i) \]

\[ \Omega^{m+1}(F_i) = \Omega^m(F_i) \cup \Delta_{out}^m(F_i) \]

sending out \( \Delta_{out}^m(F_i) \)

Buffer \( B_i \)

Monitoring Procedure

Superstep

Local Computation

Communication

Barrier synchronisation

Procedure

Monitoring

Buffer
Experiments: Comparative Study

<table>
<thead>
<tr>
<th>Query</th>
<th>Query Response Time (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_1$</td>
<td>GraphPartition</td>
</tr>
<tr>
<td>$Q_2$</td>
<td>EAGRE</td>
</tr>
<tr>
<td>$Q_3$</td>
<td>TripleGroup</td>
</tr>
<tr>
<td>$Q_4$</td>
<td>PECA</td>
</tr>
<tr>
<td>$Q_5$</td>
<td>PEDA</td>
</tr>
<tr>
<td>$Q_6$</td>
<td></td>
</tr>
<tr>
<td>$Q_7$</td>
<td></td>
</tr>
</tbody>
</table>

- GraphPartition
- EAGRE
- TripleGroup
- PECA
- PEDA
- gStore accepts the standard RDF triple files (in N3 and RDF/XML format) and SPARQL 1.0 standard.
gStore provides **C++** and **JAVA API**—Easy to use gStore in your knowledge graph applications.

### C++ API

```cpp
#include "gstoreConnector.h"
#include<sstream>

// before run this example, you must start up the GStore server at first (see command ./gstore).
int main(int argc, char * argv[]) {
    // initialize the GStore server's IP address and port.
    gstoreConnector gc("127.0.0.1", 33001);
    // build a new database by a RDF file.
    // note that the relative path is related to gstore.
    gc.create("db\data1.rdb", "example/rdf/triple/UNOM10Database.rdb");
    // then you can execute SPARQL query on this database.
    string sparql1 = "select ?a where {
        ?x a rdf:type <uh/UndergraduateStudent>.
        ?y a rdf:type <uh/Department>.
        ?z a rdf:type <uh/FullProfessor>.
        ?y rdfs:label "Course1".
        ?z rdfs:label "Department1".
    }";
    std::string answer = gc.executeQuery(sparql1);
    std::cout << answer << std::endl;
    // unload this database.
    gc.clear();
    // also, you can load some exist database directly and then query.
    gc.load("db\data1.rdb");
    answer = gc.executeQuery(sparql1);
    return 0;
}
```

### JAVA API

```java
import jgro.gstoreConnector;

// before run this example, you must start up the GStore server at first (see command ./gstore).
public class JavaAPIExample {
    public static void main(String[] args) {
        // initialize the GStore server's IP address and port.
        GstoreConnector gc = new GstoreConnector("127.0.0.1", 33001);
        // build a new database by a RDF file.
        // note that the relative path is related to gstore.
        gc.create("db\data1.rdb", "example/rdf/triple/UNOM10Database.rdb");
        // then you can execute SPARQL query on this database.
        String sparql1 = "select ?a where {
            ?x rdf:type <uh/UndergraduateStudent>.
            ?y rdf:type <uh/Department>.
            ?z rdf:type <uh/FullProfessor>.
            ?y rdfs:label "Course1".
            ?z rdfs:label "Department1".
        }";
        String answer = gc.executeQuery(sparql1);
        System.out.println(answer);
    }
}
```
gStore

- gStore is an open source effort @ Github.
gStore

- gStore has the **competitive performance** with other open source and commercial products; it supports **billions** of RDF triples and the **distributed version** comes soon.
Conclusions

▶ Graph Database is a Possible Way for RDF Knowledge Base Management.
Conclusions

▶ Graph Database is a **Possible** Way for RDF Knowledge Base Management.
▶ Subgraph Matching is a **Strong** Tool.
Conclusions

- Graph Database is a Possible Way for RDF Knowledge Base Management.
- Subgraph Matching is a Strong Tool.
- Using RDF repository, how to Provide Knowledge Services for Applications and Common Users?
Thank you!
Reference I


