Approaches to RDF Data Management and SPARQL Query Processing

M. Tamer Özsu

University of Waterloo
David R. Cheriton School of Computer Science
tamer.ozsu@uwaterloo.ca

Acknowledgements

This presentation draws upon collaborative research and discussions with the following colleagues (in alphabetical order):

- Güneş Aluç, University of Waterloo; now at SAP
- Khuzaima Daudjee, University of Waterloo
- Olaf Hartig, University of Waterloo; now at Linköping Univ.
- Lei Chen, Hong Kong University of Science & Technology
- Lei Zou, Peking University
RDF and Semantic Web

- RDF is a language for the conceptual modeling of information about resources (web resources in our context)
- A building block of semantic web
  - Facilitates exchange of information
  - Search engine results can be more focused and structured
  - Facilitates data integration (mashes)
- Machine understandable
  - Understand the information on the web and the interrelationships among them
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 Open Data Cloud
- ...
RDF Data Volumes . . .

. . . are growing – and fast

- Linked data cloud currently consists of 3000 datasets with >84B triples
- Size almost doubling every year
RDF Data Volumes . . .

- are growing – and fast
  - Linked data cloud currently consists of 3000 datasets with >84B triples
  - Size almost doubling every year

As of March 2009
Link edCT
Reactome
Tax onomy
KEGG
PubMed
GeneID
Pfam
UniProt
OMIM
PDB
Symbol
ChEBI
Daily

March ’09: 89 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 3000 datasets with >84B 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 3000 datasets with >84B triples
  - Size almost doubling every year


September ’11: 295 datasets, 25B triples
RDF Data Volumes . . .

- . . . are growing – and fast
  - Linked data cloud currently consists of 3000 datasets with >84B triples
  - Size almost doubling every year

April ’14:
570 datasets, ??? triples

Linked Object Data – Closer Look
Globally Distributed Network of Data
Three Approaches

- Data warehousing
  - Consolidate data in a centralized repository and query it

- Distributed SPARQL execution
  - Typical distribution
    - Take a warehouse and distribute
    - For scalability and performance
  - SPARQL federation
    - Leverage query services provided by data publishers

- Live Linked Data querying
  - Navigate through LOD by looking up URIs at query execution time

[Özsu, 2016]
Outline

1. RDF Introduction
2. Data Warehousing Approach
   - Relational Approaches
   - Graph-Based Approaches
3. Distributed RDF Processing
   - Cloud-based Solutions
   - Partition-based Approaches
   - Partial Execution Approach
   - SPARQL Endpoint Federation
4. Live Querying Approach
   - Traversal-based approach
   - Index-based approach
   - Hybrid approach
5. Conclusions
Outline

1. RDF Introduction
2. Data Warehousing Approach
   Relational Approaches
   Graph-Based Approaches
3. Distributed RDF Processing
   Cloud-based Solutions
   Partition-based Approaches
   Partial Execution Approach
   SPARQL Endpoint Federation
4. Live Querying Approach
   Traversal-based approach
   Index-based approach
   Hybrid approach
5. Conclusions
RDF Introduction

- Everything is an **uniquely** named resource

http://data.linkedmdb.org/resource/actor/JN29704

xmlns:y=http://data.linkedmdb.org/resource/actor/
y:JN29704

y:JN29704:hasName "Jack Nicholson"
y:JN29704:BornOnDate "1937-04-22"
y:TS2014:title "The Shining"
y:TS2014:releaseDate "1980-05-23"
y:TS2014

JN29704:movieActor
RDF Introduction

- Everything is an *uniquely* named resource
- Prefixes can be used to shorten the names

xmlns:y=http://data.linkedmdb.org/resource/actor/
y:JN29704

y:JN29704:hasName "Jack Nicholson"
y:JN29704:BornOnDate "1937-04-22"
y:TS2014:title "The Shining"
y:TS2014:releaseDate "1980-05-23"
RDF Introduction

- Everything is an uniquely named resource
- Prefixes can be used to shorten the names
- Properties of resources can be defined

```
xmlns:y=http://data.linkedmdb.org/resource/actor/
y:JN29704

y:JN29704:hasName "Jack Nicholson"
y:JN29704:BornOnDate "1937-04-22"
```
RDF Introduction

- Everything is an **uniquely** named resource
- Prefixes can be used to shorten the names
- Properties of resources can be defined
- Relationships with other resources can be defined

```xml
xmlns:y=http://data.linkedmdb.org/resource/actor/
y:JN29704

y:JN29704:hasName "Jack Nicholson"
y:JN29704:BornOnDate "1937-04-22"

y:TS2014:title "The Shining"
y:TS2014:releaseDate "1980-05-23"
```

Inria/2017-03-09
RDF Introduction

- Everything is an *uniquely* named resource
- Prefixes can be used to shorten the names
- Properties of resources can be defined
- Relationships with other resources can be defined
- Resource descriptions can be contributed by different people/groups and can be located anywhere in the web
  - Integrated web “database”

```
xmlns:y=http://data.linkedmdb.org/resource/actor/
y:JN29704

y:JN29704:hasName "Jack Nicholson"
y:JN29704:BornOnDate "1937-04-22"

y:TS2014:title "The Shining"
y:TS2014:releaseDate "1980-05-23"
```
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

\begin{align*}
\text{Subject} &\quad \text{Predicate} & \quad \text{Object} \\
\text{http://...imdb.../film/2014} &\quad \text{rdfs:label} & \quad \text{“The Shining”} \\
\text{http://...imdb.../film/2014} &\quad \text{movie:releaseDate} & \quad \text{“1980-05-23”} \\
\text{http://...imdb.../29704} &\quad \text{movie:actor_name} & \quad \text{“Jack Nicholson”} \\
\ldots &\quad \ldots & \quad \\end{align*}
## RDF Example Instance

Prefixes:
- `mdb` = http://data.linkedmdb.org/resource/
- `geo` = http://sws.geonames.org/
- `bm` = http://wifo5-03.informatik.uni-mannheim.de/bookmashup/
- `lexvo` = http://lexvo.org/id/
- `wp` = http://en.wikipedia.org/wiki/

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mdb:film/2014</code></td>
<td><code>rdfs:label</code></td>
<td>&quot;The Shining&quot;</td>
</tr>
<tr>
<td><code>mdb:film/2014</code></td>
<td><code>movie:director</code></td>
<td><code>mdb:director/8476</code></td>
</tr>
<tr>
<td><code>mdb:film/2014</code></td>
<td><code>movie:director</code></td>
<td><code>mdb:director/8476</code></td>
</tr>
<tr>
<td><code>mdb:film/2685</code></td>
<td><code>movie:director</code></td>
<td></td>
</tr>
<tr>
<td><code>mdb:film/2685</code></td>
<td><code>movie:actor</code></td>
<td></td>
</tr>
<tr>
<td><code>mdb:film/424</code></td>
<td><code>movie:director</code></td>
<td></td>
</tr>
<tr>
<td><code>mdb:film/424</code></td>
<td><code>movie:actor</code></td>
<td></td>
</tr>
<tr>
<td><code>mdb:actor/29704</code></td>
<td><code>movie:actor</code></td>
<td>&quot;Jack Nicholson&quot;</td>
</tr>
<tr>
<td><code>mdb:film/1267</code></td>
<td><code>movie:actor</code></td>
<td>&quot;The Last Tycoon&quot;</td>
</tr>
<tr>
<td><code>mdb:film/1267</code></td>
<td><code>movie:actor</code></td>
<td>&quot;The Passenger&quot;</td>
</tr>
<tr>
<td><code>geo:2635167</code></td>
<td><code>gn:name</code></td>
<td>&quot;United Kingdom&quot;</td>
</tr>
<tr>
<td><code>geo:2635167</code></td>
<td><code>gn:population</code></td>
<td>62348447</td>
</tr>
<tr>
<td><code>geo:2635167</code></td>
<td><code>gn:wikipediaArticle</code></td>
<td><code>wp:United_Kingdom</code></td>
</tr>
<tr>
<td><code>bm:books/0743424425</code></td>
<td><code>dc:creator</code></td>
<td><code>bm:persons/Stephen+King</code></td>
</tr>
<tr>
<td><code>bm:books/0743424425</code></td>
<td><code>rev:rating</code></td>
<td>4.7</td>
</tr>
<tr>
<td><code>bm:books/0743424425</code></td>
<td><code>scom:hasOffer</code></td>
<td><code>bm:offers/0743424425amazonOffer</code></td>
</tr>
<tr>
<td><code>lexvo:iso639-3/eng</code></td>
<td><code>rdfs:label</code></td>
<td>&quot;English&quot;</td>
</tr>
<tr>
<td><code>lexvo:iso639-3/eng</code></td>
<td><code>lvont:usedIn</code></td>
<td><code>lexvo:iso3166/CA</code></td>
</tr>
<tr>
<td><code>lexvo:iso639-3/eng</code></td>
<td><code>lvont:usesScript</code></td>
<td><code>lexvo:script/Latn</code></td>
</tr>
</tbody>
</table>
UniProt in RDF

- UniProt collects data from >150 biological resources
- Claim: “lack of a common standard to represent and link information makes data integration an expensive business” \implies RDF can help
UniProt in RDF – What does the data look like?

- UniProt accession for the human CYP51 protein – Q16850
- Encode it as RDF: http://purl.uniprot.org/uniprot/Q16850.rdf
UniProt in RDF – What does the data look like?

- UniProt accession for the human CYP51 protein – Q16850
- Encode it as RDF: [http://purl.uniprot.org/uniprot/Q16850.rdf](http://purl.uniprot.org/uniprot/Q16850.rdf)
- XML/RDF format

```xml
<rdf:Description rdf:about="http://purl.uniprot.org/citations/8619637">
  <rdf:type rdf:resource="http://purl.uniprot.org/core/JournalCitation"/>
  <title>The ubiquitously expressed human CYP51 encodes lanosterol 14 alpha-demethylase, a cytochrome P450 whose expression is regulated by oxysterols.</title>
  <author>Stroemstedt M.</author>
  <author>Rozman D.</author>
  <author>Waterman M.R.</author>
  <skos:exactMatch rdf:resource="http://purl.uniprot.org/pubmed/8619637"/>
  <dcterms:identifier doi:10.1006/abbi.1996.0193</dcterms:identifier>
  <date rdf:datatype="http://www.w3.org/2001/XMLSchema#gYear">1996</date>
  <name>Arch. Biochem. Biophys.</name>
  <volume>329</volume>
  <pages>73-81</pages>
</rdf:Description>
```
UniProt in RDF – What does the data look like?

- UniProt accession for the human CYP51 protein – Q16850
- Encode it as RDF: http://purl.uniprot.org/uniprot/Q16850.rdf
- XML/RDF format

```xml
<rdf:Description rdf:about="http://purl.uniprot.org/citations/8619637">
  <rdf:type rdf:resource="http://purl.uniprot.org/core/JournalCitation"/>
  <title>The ubiquitously expressed human CYP51 encodes lanosterol 14 alpha-demethylase, a cytochrome P450 whose expression is regulated by oxysterols.</title>
  <author>Stroemstedt M.</author>
  <author>Rozman D.</author>
  <author>Waterman M.R.</author>
  <skos:exactMatch rdf:resource="http://purl.uniprot.org/pubmed/8619637"/>
  <dcterms:identifier>doi:10.1006/abbi.1996.0193</dcterms:identifier>
  <date rdf:datatype="http://www.w3.org/2001/XMLSchema#gYear">1996</date>
  <name>Arch. Biochem. Biophys.</name>
  <volume>329</volume>
  <pages>73-81</pages>
</rdf:Description>
```

- This can be shown as a table <Subject, Predicate, Object>
**RDF Query Model – SPARQL**

- **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
    \[
    (U \cup V) \times (U \cup V) \times (U \cup V \cup L)
    \]
  - $?x$ rdfs:label “The Shining”
  - $P$ FILTER $R$, where $P$ is a graph pattern expression and $R$ is a built-in SPARQL condition (i.e., analogous to a SQL predicate)
  - $P1$ AND/OPT/UNION $P2$, where $P1$ and $P2$ are graph pattern expressions
RDF Query Model – SPARQL

- 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
    $$(U \cup V) \times (U \cup V) \times (U \cup V \cup L)$$
  - $?x rdfs:label “The Shining”
  - $P$ FILTER $R$, where $P$ is a graph pattern expression and $R$ is a built-in SPARQL condition (i.e., analogous to a SQL predicate)
    - $?x \text{rev:rating} ?p \text{FILTER}(?p > 3.0)$
  - $P1 \text{AND/OPT/UNION} P2$, where $P1$ and $P2$ are graph pattern expressions

*Example:*

```sql
SELECT ?name
WHERE {
  ?d movie:director_name "Stanley Kubrick".
  FILTER(?r > 4.0)
}
```
SPARQL Queries

```
SELECT ?name
WHERE {
  ?d movie:director_name "Stanley Kubrick".
  FILTER(?r > 4.0)
}
```

FILTER(?r > 4.0)
UniProt in RDF – The Data Can Be Queried

- RDF encoded UniProt data can be queried using SPARQL:
  http://sparql.uniprot.org/sparql
UniProt in RDF – The Data Can Be Queried

- RDF encoded UniProt data can be queried using SPARQL: http://sparql.uniprot.org/sparql

- Get the GO function for Q16850 (from UniProt SPARQL endpoint)
  
  ```sparql
  PREFIX upc: <http://purl.uniprot.org/core/>
  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  SELECT ?goid ?golabel
  WHERE {
    <http://purl.uniprot.org/uniprot/Q16850> a upc:Protein;
    upc:classifiedWith ?keyword .
    ?keyword rdf:sameAs ?goid .
  }
  ```

- Find the differential expression of probes and the p value that map to Q16850 (from Expression Atlas SPARQL endpoint)

  ```sparql
  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
  PREFIX atlasTerms: <http://rdf.ebi.ac.uk/terms/atlas/>
  SELECT DISTINCT ?valueLabel ?pvalue
  WHERE {
  }
  ORDER BY ASC (?pvalue)
  ```
UniProt in RDF – The Data Can Be Queried

- RDF encoded UniProt data can be queried using SPARQL:
  
  http://sparql.uniprot.org/sparql

- Get the GO function for Q16850 (from UniProt SPARQL endpoint)

  ```
  PREFIX upc: <http://purl.uniprot.org/core/>
  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  SELECT ?goid ?golabel
  WHERE {
    <http://purl.uniprot.org/uniprot/Q16850> a upc:Protein ;
    upc:classifiedWith ?keyword .
    ?keyword rdf:seeAlso ?goid .
  }
  ```

- Find the differential expression of probes and the p value that map to Q16850 (from Expression Atlas SPARQL endpoint)

  ```
  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
  PREFIX atlasterms: <http://rdf.ebi.ac.uk/terms/atlas/>
  SELECT distinct ?valueLabel ?pvalue
  WHERE {
  }
  ORDER BY ASC(?pvalue)
  ```
Outline

1. RDF Introduction
2. Data Warehousing Approach
   - Relational Approaches
   - Graph-Based Approaches
3. Distributed RDF Processing
   - Cloud-based Solutions
   - Partition-based Approaches
   - Partial Execution Approach
   - SPARQL Endpoint Federation
4. Live Querying Approach
   - Traversal-based approach
   - Index-based approach
   - Hybrid approach
5. Conclusions
SELECT ?name
WHERE {
  ?d movie:director_name "Stanley Kubrick".
  FILTER (?r > 4.0)
}

<table>
<thead>
<tr>
<th>Subject</th>
<th>Property</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>mdb:film/2014</td>
<td>rdfs:label</td>
<td>&quot;The Shining&quot;</td>
</tr>
<tr>
<td>mdb:film/2014</td>
<td>movie:director</td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>mdb:film/2014</td>
<td>movie:music_contributor</td>
<td>mdb: music_contributor/4110</td>
</tr>
<tr>
<td>mdb:film/2014</td>
<td>foaf:based_near</td>
<td>geo:2635167</td>
</tr>
<tr>
<td>mdb:director/8476</td>
<td>movie:director_name</td>
<td>&quot;Stanley Kubrick&quot;</td>
</tr>
<tr>
<td>mdb:film/2685</td>
<td>movie:director</td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>mdb:film/2685</td>
<td>rdfs:label</td>
<td>&quot;A Clockwork Orange&quot;</td>
</tr>
<tr>
<td>mdb:film/424</td>
<td>movie:director</td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>mdb:film/424</td>
<td>rdfs:label</td>
<td>&quot;Spartacus&quot;</td>
</tr>
<tr>
<td>mdb:actor/29704</td>
<td>movie:actor_name</td>
<td>&quot;Jack Nicholson&quot;</td>
</tr>
<tr>
<td>mdb:film/1267</td>
<td>movie:actor</td>
<td>mdb:actor/29704</td>
</tr>
<tr>
<td>mdb:film/1267</td>
<td>rdfs:label</td>
<td>&quot;The Last Tycoon&quot;</td>
</tr>
<tr>
<td>mdb:film/3418</td>
<td>movie:actor</td>
<td>mdb:actor/29704</td>
</tr>
<tr>
<td>mdb:film/3418</td>
<td>rdfs:label</td>
<td>&quot;The Passenger&quot;</td>
</tr>
<tr>
<td>geo:2635167</td>
<td>gn:name</td>
<td>&quot;United Kingdom&quot;</td>
</tr>
<tr>
<td>geo:2635167</td>
<td>gn:population</td>
<td>62348447</td>
</tr>
<tr>
<td>geo:2635167</td>
<td>gn:wikipediaArticle</td>
<td>wp:United_Kingdom</td>
</tr>
<tr>
<td>bm:books/0743424425</td>
<td>dc:creator</td>
<td>bm:persons/Stephen+King</td>
</tr>
<tr>
<td>bm:books/0743424425</td>
<td>rev:rating</td>
<td>4.7</td>
</tr>
<tr>
<td>bm:books/0743424425</td>
<td>scom:hasOffer</td>
<td>bm:offers/0743424425amazonOffer</td>
</tr>
<tr>
<td>lexvo:iso639-3/eng</td>
<td>rdfs:label</td>
<td>&quot;English&quot;</td>
</tr>
<tr>
<td>lexvo:iso639-3/eng</td>
<td>lvont:usedIn</td>
<td>lexvo:iso3166/CA</td>
</tr>
<tr>
<td>lexvo:iso639-3/eng</td>
<td>lvont:usesScript</td>
<td>lexvo:script/Latn</td>
</tr>
</tbody>
</table>
Naïve Triple Store Design

SELECT ?name
WHERE {
  ?m rdfs:label ?name .
  ?m movie:director ?d .
  ?d movie:director_name "Stanley Kubrick" .
  ?b rev:rating ?r .
  FILTER (?r > 4.0)
}

SELECT ?object
FROM T as T1, T as T2, T as T3, T as T4, T as T5
WHERE T1.p="rdfs:label"
AND T2.p="movie:relatedBook"
AND T3.p="movie:director"
AND T4.p="rev:rating"
AND T5.p="movie:director_name"
AND T1.s=T2.s
AND T1.s=T3.s
AND T2.s=T4.s
AND T3.s=T5.s
AND T4.o > 4.0
AND T5.o="Stanley Kubrick"
SELECT ?name
WHERE {
  ?d movie:director_name "Stanley Kubrick".
  FILTER (?r > 4.0)
}

Easy to implement but too many self-joins!
Exhaustive Indexing

- RDF-3X [Neumann and Weikum, 2008, 2009], 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>mdb:director/8476</td>
<td>movie:director_name</td>
<td>“Stanley Kubrick”</td>
</tr>
<tr>
<td>mdb:film/2685</td>
<td>movie:director</td>
<td></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>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

Mapping table

<table>
<thead>
<tr>
<th>ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>mdb: film/2014</td>
</tr>
<tr>
<td>1</td>
<td>rdfs:label</td>
</tr>
<tr>
<td>2</td>
<td>“The Shining”</td>
</tr>
<tr>
<td>3</td>
<td>movie:initial_release_date</td>
</tr>
<tr>
<td>4</td>
<td>“1980-05-23”</td>
</tr>
<tr>
<td>5</td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>6</td>
<td>movie:director_name</td>
</tr>
<tr>
<td>7</td>
<td>“Stanley Kubrick”</td>
</tr>
<tr>
<td>8</td>
<td>mdb:film/2685</td>
</tr>
<tr>
<td>9</td>
<td>movie:director</td>
</tr>
</tbody>
</table>
Exhaustive Indexing

- RDF-3X [Neumann and Weikum, 2008, 2009], 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>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Easy querying through mapping table
Exhaustive Indexing

- RDF-3X [Neumann and Weikum, 2008, 2009], 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>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>5</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>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>5</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>mdb: film/2014</td>
</tr>
<tr>
<td>1</td>
<td>rdfs:label</td>
</tr>
<tr>
<td>2</td>
<td>“The Shining”</td>
</tr>
<tr>
<td>3</td>
<td>movie:initial_release_date</td>
</tr>
<tr>
<td>4</td>
<td>“1980-05-23”</td>
</tr>
<tr>
<td>5</td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>6</td>
<td>movie:director_name</td>
</tr>
<tr>
<td>7</td>
<td>“Stanley Kubrick”</td>
</tr>
<tr>
<td>8</td>
<td>mdb:film/2685</td>
</tr>
<tr>
<td>9</td>
<td>movie:director</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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>movie:initial_release_date</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>“1980-05-23”</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>mdb:director/8476</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>movie:director_name</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>“Stanley Kubrick”</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>mdb:film/2685</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>movie:director</td>
<td></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

<table>
<thead>
<tr>
<th>ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>mdb: film/2014</td>
</tr>
<tr>
<td>1</td>
<td>rdfs:label</td>
</tr>
<tr>
<td>2</td>
<td>&quot;The Shining&quot;</td>
</tr>
<tr>
<td>3</td>
<td>movie:initial release date</td>
</tr>
<tr>
<td>4</td>
<td>&quot;1980-05-23&quot;</td>
</tr>
<tr>
<td>5</td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>6</td>
<td>movie:director name</td>
</tr>
<tr>
<td>7</td>
<td>&quot;Stanley Kubrick&quot;</td>
</tr>
<tr>
<td>8</td>
<td>mdb:film/2685</td>
</tr>
<tr>
<td>9</td>
<td>movie:director</td>
</tr>
</tbody>
</table>

Advantages

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

Disadvantages

- Space usage
- Expensive updates
Property Tables

- Grouping by entities; Jena [Wilkinson, 2006], 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>mdb:film/2014</td>
<td>movie:director</td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>mdb:film/2685</td>
<td>movie:director</td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>mdb:film/2685</td>
<td>rdfs:label</td>
<td>“A Clockwork Orange”</td>
</tr>
<tr>
<td>mdb:actor/29704</td>
<td>movie:actor_name</td>
<td>“Jack Nicholson”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>refs:label</th>
<th>movie:director</th>
<th>Subject</th>
<th>movie:actor_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>mob:film/2685</td>
<td>“The Clockwork Orange”</td>
<td>mob:director/8476</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Property Tables

- Grouping by entities; Jena [Wilkinson, 2006], 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

---

<table>
<thead>
<tr>
<th>Subject</th>
<th>refs:label</th>
<th>movie:director</th>
</tr>
</thead>
<tbody>
<tr>
<td>mob:film/2685</td>
<td>“The Clockwork Orange”</td>
<td>mob:director/8476</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“A Clockwork Orange”</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Subject</th>
<th>movie:actor_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>mob:actor/29704</td>
<td>“Jack Nicholson”</td>
</tr>
<tr>
<td>mob:actor/2685</td>
<td>“Jack Nicholson”</td>
</tr>
</tbody>
</table>

---
Property Tables

- Grouping by entities; Jena [Wilkinson, 2006], DB2-RDF [Bornea et al., 2013]

  - Clustered property table: group together the properties that

<table>
<thead>
<tr>
<th>Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fewer joins</td>
</tr>
<tr>
<td>If the data is structured, we have a relational system – similar to normalized relations</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Potentially a lot of NULLs</td>
</tr>
<tr>
<td>Clustering is not trivial</td>
</tr>
<tr>
<td>Multi-valued properties are complicated</td>
</tr>
</tbody>
</table>

- | Subject | Property | Object |
  - | mdb:film/2014 | rdfs:label | "The Shining" |
  - | mdb:film/2685 | movie:director | mdb:director/8476 |
  - | mdb:film/2685 | rdfs:label | "A Clockwork Orange" |
  - | mdb:actor/29704 | movie:actor | name | "Jack Nicholson" |
Binary Tables

- Grouping by properties: For each property, build a two-column table, containing both subject and object, ordered by subjects [Abadi et al., 2007, 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>mdb:film/2014</td>
<td>movie:director</td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>mdb:film/2685</td>
<td>movie:director</td>
<td>“A Clockwork Orange”</td>
</tr>
<tr>
<td>mdb:actor/29704</td>
<td>movie:actor_name</td>
<td>“Jack Nicholson“</td>
</tr>
</tbody>
</table>

refs:label

movie:director

movie:actor_name
Binary Tables

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

[Abadi et al., 2007, 2009]

Also called vertical partitioned tables

\[ \text{subject} \times \text{object} \quad (n \text{ is the number of unique properties in the data}) \]

### 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>mob:film/2685</td>
<td>“The Clockwork Orange”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>mdb:film/2014</td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>mdb:film/2685</td>
<td>mdb:director/8476</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>mdb:actor/29704</td>
<td>“Jack Nicholson”</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.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>mdb:film/2014</td>
<td>rdfs:label</td>
</tr>
<tr>
<td></td>
<td>&quot;The Shining&quot;</td>
</tr>
<tr>
<td>mdb:film/2014</td>
<td>movie:director</td>
</tr>
<tr>
<td></td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>mdb:film/2685</td>
<td>movie:director</td>
</tr>
<tr>
<td></td>
<td>mdb:director/8476</td>
</tr>
<tr>
<td>mdb:film/2685</td>
<td>rdfs:label</td>
</tr>
<tr>
<td></td>
<td>&quot;A Clockwork Orange&quot;</td>
</tr>
<tr>
<td>mdb:actor/29704</td>
<td>movie:actor</td>
</tr>
<tr>
<td></td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>&quot;Jack Nicholson&quot;</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

## Disadvantages

- Not useful for subject-object joins
- Expensive inserts
Graph-based Approach

- Answering SPARQL query $\equiv$ subgraph matching using homomorphism
- gStore [Zou et al., 2011, 2014], chameleon-db [Aluç et al., 2013]
Graph-based Approach

- Answering SPARQL query \( \equiv \) subgraph matching using homomorphism
- gStore [Zou et al., 2011, 2014], chameleon-db [Aluç et al., 2013]

Advantages

- Maintains the graph structure
- Full set of queries can be handled
Graph-based Approach

- Answering SPARQL query $\equiv$ subgraph matching using homomorphism
- gStore [Zou et al., 2011, 2014], chameleon-db [Aluç et al., 2013]

Advantages

- Maintains the graph structure
- Full set of queries can be handled

Disadvantages

- Graph pattern matching is expensive
Two Systems

<table>
<thead>
<tr>
<th>gStore Offline</th>
<th>gStore Online</th>
<th>chameleon-db</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>RDF data</td>
<td>Plan Generation</td>
</tr>
<tr>
<td>RDF Parser</td>
<td>RDF Triples</td>
<td>Evaluation</td>
</tr>
<tr>
<td>RDF Graph Builder</td>
<td>RDF Graph</td>
<td></td>
</tr>
<tr>
<td>Encoding Module</td>
<td>Signature Graph</td>
<td></td>
</tr>
<tr>
<td>VS*-tree builder</td>
<td>Filter Module</td>
<td></td>
</tr>
<tr>
<td>Storage</td>
<td>VS*-tree Store</td>
<td></td>
</tr>
<tr>
<td>Storage</td>
<td>Key-Value Store</td>
<td></td>
</tr>
</tbody>
</table>

Let $|S|$ be $M$. Using an appropriate hash function, we set $m$ out of $M$ bits in $S$ to be '1'.

Queries with di...
Two Systems

gStore

- 12,000 lines of C++ code under Linux (plus code for SPARQL parser)
- Encode each vertex of RDF graph as a bit array capturing the neighbourhood relationship ($G^*$)
- Build a multilevel summary tree index (VS*-tree) to capture “connections”
Two Systems

**gStore**

<table>
<thead>
<tr>
<th>Offline</th>
<th>Online</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td><strong>Input</strong></td>
</tr>
<tr>
<td>RDF data</td>
<td>SPARQL Query</td>
</tr>
<tr>
<td>RDF Parser</td>
<td>RDF Trips</td>
</tr>
<tr>
<td><strong>RDF Graph Builder</strong></td>
<td>Query Graph</td>
</tr>
<tr>
<td>RDF Graph</td>
<td>Encoding Module</td>
</tr>
<tr>
<td>Encoding Module</td>
<td>Signature Graph</td>
</tr>
<tr>
<td><strong>Signature Graph</strong></td>
<td>Filter Module</td>
</tr>
<tr>
<td>VS*-tree builder</td>
<td>Node Candidate</td>
</tr>
<tr>
<td>VS*-tree</td>
<td><strong>Join Module</strong></td>
</tr>
<tr>
<td><strong>Storage</strong></td>
<td></td>
</tr>
<tr>
<td>VS*-tree Store</td>
<td>Key-Value Store</td>
</tr>
</tbody>
</table>

- Encode the query graph similarly ($Q^*$)
- Find candidate matching nodes of $Q^*$ in $G^*$ using VS*-tree
- Multiway join of the candidates

- 12,000 lines of C++ code under Linux (plus code for SPARQL parser)
- Encode each vertex of RDF graph as a bit array capturing the neighbourhood relationship ($G^*$)
- Build a multilevel summary tree index (VS*-tree) to capture “connections”
Two Systems

- 35,000 lines of C++ code under Linux (plus code for SPARQL 1.0 parser)
- Adaptivity to workload due to variability of Web workloads and the variability of composition of SPARQL triple patterns
- An experiment [Aluç et al., 2014a]
  - No single system is a sole winner across all queries
  - No single system is the sole loser across all queries, either
  - 2–5 orders of magnitude difference in the performance between the best and the worst system for a given query
  - The winner in one query may timeout in another
  - Performance difference widens as dataset size increases

- Group-by-query approach [Aluç et al., 2014b]
Outline

1 RDF Introduction
2 Data Warehousing Approach
   Relational Approaches
   Graph-Based Approaches
3 Distributed RDF Processing
   Cloud-based Solutions
   Partition-based Approaches
   Partial Execution Approach
   SPARQL Endpoint Federation
4 Live Querying Approach
   Traversal-based approach
   Index-based approach
   Hybrid approach
5 Conclusions
Remember the Environment

- Distributed environment
- Some of the data sites can process SPARQL queries – SPARQL endpoints
- Not all data sites can process queries
Remember the Environment

- Distributed environment
- Some of the data sites can process SPARQL queries – SPARQL endpoints
- Not all data sites can process queries
- Alternatives
Remember the Environment

- Distributed environment
- Some of the data sites can process SPARQL queries – SPARQL endpoints
- Not all data sites can process queries
- Alternatives
  - Cloud-based approaches
Remember the Environment

- Distributed environment
- Some of the data sites can process SPARQL queries – SPARQL endpoints
- Not all data sites can process queries
- Alternatives
  - Cloud-based approaches
  - Data re-distribution + query decomposition
Remember the Environment

- Distributed environment
- Some of the data sites can process SPARQL queries – SPARQL endpoints
- Not all data sites can process queries
- Alternatives
  - Cloud-based approaches
  - Data re-distribution + query decomposition
  - Data re-distribution + partial evaluation
Remember the Environment

- Distributed environment
- Some of the data sites can process SPARQL queries – SPARQL endpoints
- Not all data sites can process queries
- Alternatives
  - Cloud-based approaches
  - Data re-distribution + query decomposition
  - Data re-distribution + partial evaluation
  - SPARQL federation: just process at SPARQL endpoints
Cloud-based Solutions [Kaoudi and Manolescu, 2015]

- RDF data warehouse $D$ is partitioned ($\{D_1, \ldots, D_n\}$) and placed on cloud platforms (such as HDFS, HBase)

- High scalability and fault-tolerance
- Possibly low performance since MapReduce is not suitable for graph processing
Cloud-based Solutions

- RDF data warehouse $D$ is partitioned ($\{D_1, \ldots, D_n\}$) and placed on cloud platforms (such as HDFS, HBase).
- SPARQL query is run through MapReduce jobs.
- Data parallel execution.

[Kaoudi and Manolescu, 2015]
Cloud-based Solutions

- RDF data warehouse \( D \) is partitioned (\( \{D_1, \ldots, D_n\} \)) and placed on cloud platforms (such as HDFS, HBase)
- SPARQL query is run through MapReduce jobs
- Data parallel execution
- Examples: HARD [Rohloff and Schantz, 2010], HadoopRDF [Husain et al., 2011], EAGRE [Zhang et al., 2013] and JenaHBase [Khadilkar et al., 2012]

- High scalability and fault-tolerance
- Possibly low performance since MapReduce is not suitable for graph processing
Cloud-based Solutions [Kaoudi and Manolescu, 2015]

- RDF data warehouse $D$ is partitioned ($\{D_1, \ldots, D_n\}$) and placed on cloud platforms (such as HDFS, HBase)
- SPARQL query is run through MapReduce jobs
- Data parallel execution
- Examples: HARD [Rohloff and Schantz, 2010], HadoopRDF [Husain et al., 2011], EAGRE [Zhang et al., 2013] and JenaHBase [Khadilkar et al., 2012]

- High scalability and fault-tolerance
- Possibly low performance since MapReduce is not suitable for graph processing
Partition-based Approaches

- (Offline) Partition an RDF data warehouse (graph) into several fragments that are distributed to sites
  - RDF data \( D = \{ D_1, \ldots, D_n \} \)
  - Allocate each \( D_i \) to a site

Partitioning alternatives

- Table-based (e.g., [Husain et al., 2011])
- Graph-based (e.g., [Huang et al., 2011; Zhang et al., 2013])
- Unit-based (e.g., [Gurajada et al., 2014; Lee and Liu, 2013])

(Online) SPARQL query decomposed

\[ Q = \{ Q_1, \ldots, Q_k \} \Rightarrow \] query graph is decomposed

Distributed execution of \{ \( Q_1, \ldots, Q_k \) \} over \{ \( D_1, \ldots, D_n \) \}

Examples: GraphPartition [Huang et al., 2011], WARP [Hose and Schenkel, 2013], Partout [Galarraga et al., 2014], Vertex-block [Lee and Liu, 2013]

▶ High performance
▶ Great for parallelizing centralized RDF data
▶ May not be possible to re-partition and re-allocate Web data (i.e., LOD)
▶ Each approach requires a specific partitioning strategy – no generic partitioning
▶ Query decomposition may not be easy
Partition-based Approaches

(Offline) Partition an RDF data warehouse (graph) into several fragments that are distributed to sites

- RDF data \( D = \{ D_1, \ldots, D_n \} \)
- Allocate each \( D_i \) to a site

Partitioning alternatives

- Table-based (e.g., [Husain et al., 2011])
- Graph-based (e.g., [Huang et al., 2011; Zhang et al., 2013])
- Unit-based (e.g., [Gurajada et al., 2014; Lee and Liu, 2013])
Partition-based Approaches

- (Offline) Partition an RDF data warehouse (graph) into several fragments that are distributed to sites
  - RDF data $D = \{D_1, \ldots, D_n\}$
  - Allocate each $D_i$ to a site

- Partitioning alternatives
  - Table-based (e.g., [Husain et al., 2011])
  - Graph-based (e.g., [Huang et al., 2011; Zhang et al., 2013])
  - Unit-based (e.g., [Gurajada et al., 2014; Lee and Liu, 2013])

- (Online) SPARQL query decomposed $Q = \{Q_1, \ldots, Q_k\} \Rightarrow$
  query graph is decomposed

- Distributed execution of $\{Q_1, \ldots, Q_k\}$ over $\{D_1, \ldots, D_n\}$
Partition-based Approaches

- **(Offline)** Partition an RDF data warehouse (graph) into several fragments that are distributed to sites
  - RDF data $D = \{D_1, \ldots, D_n\}$
  - Allocate each $D_i$ to a site

- Partitioning alternatives
  - Table-based (e.g., [Husain et al., 2011])
  - Graph-based (e.g., [Huang et al., 2011; Zhang et al., 2013])
  - Unit-based (e.g., [Gurajada et al., 2014; Lee and Liu, 2013])

- **(Online)** SPARQL query decomposed $Q = \{Q_1, \ldots, Q_k\}$ ⇒ query graph is decomposed

- Distributed execution of $\{Q_1, \ldots, Q_k\}$ over $\{D_1, \ldots, D_n\}$

- Examples: GraphPartition [Huang et al., 2011], WARP [Hose and Schenkel, 2013], Partout [Galarraga et al., 2014], Vertex-block [Lee and Liu, 2013]
Partition-based Approaches

- (Offline) Partition an RDF data warehouse (graph) into several fragments that are distributed to sites
  - RDF data $D = \{D_1, \ldots, D_n\}$
  - Allocate each $D_i$ to a site

- Partitioning alternatives
  - Table-based (e.g., [Husain et al., 2011])
  - Graph-based (e.g., [Huang et al., 2011; Zhang et al., 2013])
  - Unit-based (e.g., [Gurajada et al., 2014; Lee and Liu, 2013])

- (Online) SPARQL query decomposed as $Q = \{Q_1, \ldots, Q_k\}$⇒ query graph is decomposed
- Distributed execution of $\{Q_1, \ldots, Q_k\}$ over $\{D_1, \ldots, D_n\}$

Examples: GraphPartition [Huang et al., 2011], WARP [Hose and Schenkel, 2013], Partout [Galarraga et al., 2014], Vertex-block [Lee and Liu, 2013]

- High performance
- Great for parallelizing centralized RDF data
- May not be possible to re-partition and re-allocate Web data (i.e., LOD)
- Each approach requires a specific partitioning strategy – no generic partitioning
- Query decomposition may not be easy
Partial Query Evaluation (PQE)

- RDF data warehouse is partitioned and distributed as before
  - RDF data $D = \{D_1, \ldots, D_n\}$
  - Allocate each $D_i$ to a site
- SPARQL query is not decomposed
- Partial query evaluation – Distributed gStore [Peng et al., 2016]
Partial Query Evaluation (PQE)

- RDF data warehouse is partitioned and distributed as before
  - RDF data \( D = \{D_1, \ldots, D_n\} \)
  - Allocate each \( D_i \) to a site
- SPARQL query is not decomposed
- Partial query evaluation – Distributed gStore [Peng et al., 2016]

\[
\begin{align*}
  f(x) & \Rightarrow f(s, d) \Rightarrow f''(f'(s), d) \Rightarrow \text{Final Answer}
\end{align*}
\]

- known inputs
- unknown inputs
Partial Query Evaluation (PQE)

- RDF data warehouse is partitioned and distributed as before
  - RDF data \( D = \{ D_1, \ldots, D_n \} \)
  - Allocate each \( D_i \) to a site

- SPARQL query is not decomposed

- Partial query evaluation – Distributed gStore [Peng et al., 2016]

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

- known inputs
- unknown inputs
- partial results
Partial Query Evaluation (PQE)

- RDF data warehouse is partitioned and distributed as before
  - RDF data \( D = \{ D_1, \ldots, D_n \} \)
  - Allocate each \( D_i \) to a site

- SPARQL query is not decomposed

- Partial query evaluation – Distributed gStore [Peng et al., 2016]

\[
\begin{align*}
  f(x) & \Rightarrow f(s, d) \Rightarrow f''(f'(s), d)) \Rightarrow \text{Final Answer}
\end{align*}
\]

- Query is the function and each \( D_i \) is the known input

known inputs \quad unknown inputs \quad partial results
Distributed SPARQL Using PQE

Two steps:

1. Evaluate a query at each site to find local matches
   - These are local partial matches
Distributed SPARQL Using PQE

Two steps:

1. Evaluate a query at each site to find local matches
   - These are local partial matches
2. Assemble the partial matches to get final result
   - Crossing match
   - Centralized assembly
   - Distributed assembly
Distributed SPARQL Using PQE

[Two steps:]

1. Evaluate a query at each site to find local matches
   - These are local partial matches

2. Assemble the partial matches to get final result
   - Crossing match
   - Centralized assembly
   - Distributed assembly

- High performance due to parallelization
- Do not have to deal with query decomposition
- May not be possible to re-partition and re-allocate Web data (i.e., LOD)
- RDF storage sites need to be modified to handle partial query processing
SPARQL Endpoint Federation

- Consider only the SPARQL endpoints for query execution
- No data re-partitioning/re-distribution
- Consider $D = D_1 \cup D_2 \cup \ldots \cup D_n$; $D_i$: SPARQL endpoint
SPARQL Endpoint Federation

- Consider only the SPARQL endpoints for query execution
- No data re-partitioning/re-distribution
- Consider $D = D_1 \cup D_2 \cup \ldots \cup D_n$; $D_i$: SPARQL endpoint
- SPARQL query decomposed $Q = \{Q_1, \ldots, Q_k\}$
- Distributed execution of $\{Q_1, \ldots, Q_k\}$ over $\{D_1, \ldots, D_n\}$
SPARQL Endpoint Federation

- Consider only the SPARQL endpoints for query execution
- No data re-partitioning/re-distribution
- Consider $D = D_1 \cup D_2 \cup \ldots \cup D_n$; $D_i$: SPARQL endpoint
- SPARQL query decomposed $Q = \{Q_1, \ldots, Q_k\}$
- Distributed execution of $\{Q_1, \ldots, Q_k\}$ over $\{D_1, \ldots, D_n\}$
- Systems
  - DARQ [Quilitz and Leser, 2008], FedEx [Schwarte et al., 2011], SPLENDID [Görlitz and Staab, 2011], ANAPSID [Acosta et al., 2011]
SPARQL Endpoint Federation

- Consider only the SPARQL endpoints for query execution
- No data re-partitioning/re-distribution
- Consider $D = D_1 \cup D_2 \cup \ldots \cup D_n$; $D_i$: SPARQL endpoint
- SPARQL query decomposed $Q = \{Q_1, \ldots, Q_k\}$
- Distributed execution of $\{Q_1, \ldots, Q_k\}$ over $\{D_1, \ldots, D_n\}$
- Systems
  - DARQ [Quilitz and Leser, 2008], FedX [Schwarte et al., 2011], SPLENDID [Görlitz and Staab, 2011], ANAPSID [Acosta et al., 2011]

- Data integration approach
- May be the only way to proceed if data is distributed
- Not all RDF data storage points are SPARQL endpoints
UniProt Federation – EBI RDF Platform

Curated computational models of biological processes
Sample information for reference samples and samples for which data exist in one of the EBI’s assay databases
Curated chemical database of bioactive molecules with drug-like properties

Genome databases for vertebrates and other eukaryotic species

Gene expression data from the Gene Expression Atlas

Curated and peer-reviewed pathways
Federated Access to UniProt Collection

Get the Reactome pathways where Q16850 is associated, then get all the other proteins in that pathway and pull out their expression from the atlas, along with the GO annotations from UniProt

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX biopax3: <http://www.biopax.org/release/biopax-level3.owl#>
PREFIX atlasterm: <http://rdfs.ebi.ac.uk/terms/atlas/>
PREFIX upc:<http://purl.uniprot.org/core/>
SELECT DISTINCT ?pathwayname ?expressionValue ?golabel
WHERE {
  # Get the pathways that reference Q16850
  ?pathway biopax3:pathwayComponent
    [?rel [biopax3:entityReference ?dbXref]] .
  
  # Get the expression for those proteins
  SERVICE <http://www.ebi.ac.uk/rdf/services/atlas/sparql> {
    ?value atlasterm:dbXref ?dbXref .
  }

  # get the GO functions from Uniprot
  SERVICE <http://uniprot.org/sparql> {
    ?dbXref a upc:Protein ;
    upc:classifiedWith ?keyword .
    ?keyword rdfs:seeAlso ?goid .
  }
}
```
Outline

1. RDF Introduction
2. Data Warehousing Approach
   - Relational Approaches
   - Graph-Based Approaches
3. Distributed RDF Processing
   - Cloud-based Solutions
   - Partition-based Approaches
   - Partial Execution Approach
   - SPARQL Endpoint Federation
4. Live Querying Approach
   - Traversal-based approach
   - Index-based approach
   - Hybrid approach
5. Conclusions
Traditional Hypertext-based Web Access

Data exposed to the Web via HTML
Linked Data Publishing Principles

Data model: RDF
Global identifier: URI
Access mechanism: HTTP
Connection: data links
Live Query Processing

- Not all data resides at SPARQL endpoints
- Freshness of access to data important
- Potentially countably infinite data sources
- Live querying
  - On-line execution
  - Only rely on linked data principles
- Alternatives
  - Traversal-based approaches
  - Index-based approaches
  - Hybrid approaches
## Web of Linked Data

Given a finite or countably infinite set \( \mathcal{D} \) of Linked Documents, a Web of Linked Data is a tuple \( W = (\mathcal{D}, \text{adoc, data}) \) where:

- \( \mathcal{D} \subseteq \mathcal{D} \),
- \( \text{adoc} \) is a partial mapping from URIs to \( \mathcal{D} \), and
- \( \text{data} \) is a total mapping from \( \mathcal{D} \) to finite sets of RDF triples.
Linked Data Model

Web of Linked Data

Given a finite or countably infinite set \( \mathcal{D} \) of Linked Documents, a Web of Linked Data is a tuple \( W = (\mathcal{D}, adoc, data) \) where:

- \( D \subseteq \mathcal{D} \),
- \( adoc \) is a partial mapping from URIs to \( D \), and
- \( data \) is a total mapping from \( D \) to finite sets of RDF triples.

Data Links

A Web of Linked Data \( W = (\mathcal{D}, adoc, data) \) contains a data link from document \( d \in \mathcal{D} \) to document \( d' \in \mathcal{D} \) if there exists a URI \( u \) such that:

- \( u \) is mentioned in an RDF triple \( t \in data(d) \), and
- \( d' = adoc(u) \).
SPARQL Query Semantics in Live Querying

- Full-web semantics
  - Scope of evaluating a SPARQL expression is all Linked Data
  - Query result completeness cannot be guaranteed by any (terminating) execution
SPARQL Query Semantics in Live Querying

- **Full-web semantics**
  - Scope of evaluating a SPARQL expression is all Linked Data
  - Query result completeness cannot be guaranteed by any (terminating) execution

- **Reachability-based query semantics**
  - Query consists of a SPARQL expression, a set of seed URIs $S$, and a reachability condition $c$
  - Scope: all data along paths of data links that satisfy the condition
  - Computationally feasible
Traversing Approaches

- Discover relevant URIs recursively by traversing (specific) data links at query execution runtime [Hartig, 2013; Ladwig and Tran, 2011]
- Implements reachability-based query semantics
  - Start from a set of seed URIs
  - Recursively follow and discover new URIs
- Important issue is selection of seed URIs
- Retrieved data serves to discover new URIs and to construct result
Traversing Approaches

- Discover relevant URIs recursively by traversing (specific) data links at query execution runtime [Hartig, 2013; Ladwig and Tran, 2011]
-Implements reachability-based query semantics

Advantages

Easy to implement.
No data structure to maintain.

- Important issue is selection of seed URIs
- Retrieved data serves to discover new URIs and to construct result
Traversal Approaches

- Discover relevant URIs recursively by traversing (specific) data links at query execution runtime [Hartig, 2013; Ladwig and Tran, 2011]
- Implements reachability-based query semantics

Advantages

- Easy to implement.
- No data structure to maintain.
- Important issue is selection of seed

Disadvantages

- Possibilities for parallelized data retrieval are limited
- Repeated data retrieval introduces significant query latency.
Outline

1. RDF Introduction
2. Data Warehousing Approach
   Relational Approaches
   Graph-Based Approaches
3. Distributed RDF Processing
   Cloud-based Solutions
   Partition-based Approaches
   Partial Execution Approach
   SPARQL Endpoint Federation
4. Live Querying Approach
   Traversal-based approach
   Index-based approach
   Hybrid approach
5. Conclusions
Conclusions

- RDF and Linked Object Data seem to have considerable promise and have found use
- There are prototype systems that provide alternative solutions
- There are commercial systems too: e.g., MarkLogic, Virtuoso
Conclusions

- RDF and Linked Object Data seem to have considerable promise and have found use
- There are prototype systems that provide alternative solutions
- There are commercial systems too: e.g., MarkLogic, Virtuoso
- More work needs to be done (and I did not talk about these)
  - Query semantics
  - Adaptive system design
  - Optimizations – both in data warehousing and distributed environments
  - Live querying requires significant thought to reduce latency
Conclusions

- RDF and Linked Object Data seem to have considerable promise and have found use
- There are prototype systems that provide alternative solutions
- There are commercial systems too: e.g., MarkLogic, Virtuoso
- More work needs to be done (and I did not talk about these)
  - Query semantics
  - Adaptive system design
  - Optimizations – both in data warehousing and distributed environments
  - Live querying requires significant thought to reduce latency
- What I did not talk about:
  - Not much on general distributed/parallel processing
  - Not much on SPARQL semantics
  - Nothing about RDFS – no schema stuff
  - Nothing about entailment regimes $> 0 \Rightarrow$ no reasoning
Thank you!

Research supported by

[Logos of various sponsors]


References III


References V


References VI

