RDF data and SPARQL query processing

École d’Hiver é-EGC Semantic Web

Bernd Amann
Grenoble, 22 janvier 2017

UPMC-LIP6
Introduction

Goal of this tutorial

- Give a flavor of some important RDF data processing issues
- Concentrate on data access / query evaluation
- Show the role of database technology for the construction of the semantic web

Prerequisites

- Basic notions in SQL, relational algebra (join), query optimization (index)
Outline

1 RDF and SPARQL

2 RDF storage and SPARQL query evaluation
1 - RDF and SPARQL

- RDF Data Model(s)
- SPARQL by Example
- SPARQL and Entailment
- SPARQL algebra
RDF in a nutshell

Resource Description Framework: RDF & SPARQL

Universal data model for sharing data and knowledge:

- many-sided data model:
  - data: triple sets, labeled graphs, structured object graphs
  - schemas (optional): classes and properties, sub-class/sub-properties, entailment
  - queries: graph pattern matching, set oriented algebra

- facilitates incremental data and knowledge integration: data/resource linking, semantic data annotation, schema-based validation, inference/entailment

- data model for new information and knowledge processing technologies merging different technologies: web, databases, knowledge-based reasoning, NLP

W3C foundations for the Semantic Web

- Feb 1999: RDF Model and Syntax Specification Recommendation
- Feb. 2004: RDF Vocabulary Description Language 1.0: RDF Schema
- Feb. 2004: RDF Semantics
- Jan. 2008: SPARQL Protocol and RDF Query Language 1.0
- Mar. 2013: SPARQL 1.1 Query Language
RDF model 1: graphs

- identified nodes (blue circles): r1, r2, r3, ...
- unidentified (blank) nodes (green circles): existential quantifier for nodes
- typed literal nodes (rectangles): 'François I', 'Mary Stuart',...
RDF model 2: triple sets

Turtle roisblankst

```turtle
```

Set of triples \((s, p, o)\):

- \(o\) is the value/object of property \(p\) for subject \(s\)
- \(s\) and \(p\) are URIs
- \(o\) is an URI or a typed literal
RDF model 3: typed resources and properties

Turtle_roisblankst

@prefix : <http://www.asws.com/rois#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

:r1 rdf:type rdf:Resource ; :fils :r3 ; :nom "Francois I" .
:r5 rdf:type rdf:Resource ; :epoux :r3 ; :nom "Catherine de Medici" ;
:r8 rdf:type rdf:Resource ; :epoux :r2 ; :nom "Elisabeth d'Autriche" .

- :r1, :r2, ... are resources in namespace http://www.asws.com/rois#
- :fils is a property in namespace http://www.asws.com/rois#
- rdf:Resource is a resource in namespace
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
- rdf:type is a property in namespace
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
Turtle roisschema

@prefix : <http://www.asws.com/rois#>  .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>  .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#>  .

:Personne rdf:type rdfs:Class .
:Homme rdf:type rdfs:Class ; rdfs:subClassOf :Personne .
:Femme rdf:type rdfs:Class ; rdfs:subClassOf :Personne .
:Roi rdf:type rdfs:Class ; rdfs:subClassOf :Personne .
:Reine rdf:type rdfs:Class ; rdfs:subClassOf :Personne .
:fils rdf:type rdf:Property ; rdfs:subPropertyOf :enfant ;

■ :Personne, :Homme, :Femme ... are classes;
■ :Homme is a subclass of :Personne ...
■ :epoux is a resource of type :rdf:Property with domain :Reine and range :Roi;
■ :fils is a sub-property of property :enfant.
RDFS schema = RDF graph
RDF/S entailment

The semantics of an RDF/S (RDF+RDFS) graph is defined by a set of logical **entailment rules** generating new triples:

- **triple (a, p, b)** produces triples
  - (:a, rdf:type, rdf:Resource),
  - (:b, rdf:type, rdf:Resource),
  - (:p, rdf:type, rdf:Property)

- **triple (p, rdfs:domain, c)** then produces triples
  - (:p, rdf:type, rdf:Property),
  - (:c, rdf:type, rdfs:Class), (:a, rdf:type, :c)

- **triple (c, rdfs:subClassOf, d)** then produces triple
  - (:a, rdf:type, :d)

The resulting graph encodes all semantic relationships defined by rdfs:subClassof, rdfs:subPropertyOf, ...
Complete RDF graph after entailment
RDF: Summary

- Universal model with multi-sided semantics and formats
- Any RDF/S definition can be materialized into a complete set of triples (entailment)
- The precise semantics with blank nodes is obtained through the reduction into a unique “minimal” graph.
1 - RDF and SPARQL

- RDF Data Model(s)
- SPARQL by Example
- SPARQL and Entailment
- SPARQL algebra
Querying RDF

We search for

- names of kings
- kings without sons
- queens with more than 3 children
- the names of the descendants of François Ier
- the brothers of Henry III

A (RDF) query language must

- be declarative and independent of a particular implementation
- have a precise formal semantics
- take into account the RDF schemas (schema awareness)
- allow to query the RDFS schema and the RDF data
**Graph Patterns**

**Query q1**
```
select ?nf ?ne
from <ex1.ttl>
where {
  ?r :nom "Francois I" ;
  :r5 :nom ?ne }
```

**Query q1ext**
```
select ?nf ?ne
from <ex1.ttl>
where {
  ?r :nom "Francois I" .
  :r5 :nom ?ne }
```

**Basic Graph Pattern (BGP)**
Set of triples (Turtle expression) where node and property identifiers can be replaced by variables.
Graph pattern matching semantics

Mapping query $Q$ in data-set $G$

- compute all mappings $m$ for variables and blank nodes from $Q$ to $G$ such that $m(Q)$ is a subgraph of $G$.
- query answer is the set of mappings (variable bindings) for variables in the query SELECT clause
OPTIONAL (jointure externe)

**Turtle ex1**

```turtle
:r1 :nom "Francois I" ; :fils :r3 .
:r3 :nom "Henri II" ; :epouse :r5 .
:r5 :nom "Catherine de Medicis" ;
```

**Query q2**

```sparql
select ?n ?nf ?ne
from <ex1.ttl>
where {
    OPTIONAL { ?r :epouse ?e . ?e :nom ?ne }
}
```

**Result of q2**

<table>
<thead>
<tr>
<th>n</th>
<th>nf</th>
<th>ne</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Catherine de Medicis&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;Henri II&quot;</td>
<td></td>
<td>&quot;Catherine de Medicis&quot;</td>
</tr>
<tr>
<td>&quot;Francois I&quot;</td>
<td>&quot;Henri II&quot;</td>
<td></td>
</tr>
</tbody>
</table>
FILTER and UNION

Turtle ex1

:r1 :nom "Francois I" ; :fils :r3 .
:r3 :nom "Henri II" ; :epouse :r5 .
:r5 :nom "Catherine de Medicis" ;

Query q3

```
select ?n
from <ex1.ttl>
where {
  FILTER regex (?n,"^Cat")
}
```

Result of q3

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Catherine de Medicis&quot;</td>
</tr>
</tbody>
</table>

Query q4

```
select ?n
from <ex1.ttl>
where {
  { :r1 :nom ?n . }
  UNION
  { :r1 :fils [ :nom ?n ] }
}
```

Result of q4

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Francois I&quot;</td>
</tr>
<tr>
<td>&quot;Henri II&quot;</td>
</tr>
</tbody>
</table>
Example: 3-colorability

**Problem**

Is it possible to use a maximum of 3 distinct colors to color each node in graph $G$ such that two adjacent nodes never have the same color?

$G$:

![Graph G](image)

**Compute answer with SPARQL**

- encode graph $G$ into a SPARQL query $Q$ (undirected edge = 2 directed edges)
- apply $Q$ on a complete RDF graph $RGB$ with three nodes (red, green, blue)
- answer $Q(RGB)$ returns all possible 3-colorings

Known as an NP-complete problem.
Example: 3-colorability

Turtle couleur

:r :e :b ; :e :g .
:b :e :r ; :e :g .
:g :e :r ; :e :b .

Query couleur1

SELECT DISTINCT ?a ?b ?c ?d ?f ?g
FROM <couleur.ttl>
WHERE {
}

Result of couleur1

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>:g</td>
<td>:b</td>
<td>:r</td>
<td>:r</td>
<td>:b</td>
<td>:b</td>
</tr>
<tr>
<td>:g</td>
<td>:r</td>
<td>:b</td>
<td>:b</td>
<td>:r</td>
<td>:r</td>
</tr>
<tr>
<td>:b</td>
<td>:g</td>
<td>:r</td>
<td>:r</td>
<td>:g</td>
<td>:g</td>
</tr>
<tr>
<td>:b</td>
<td>:r</td>
<td>:g</td>
<td>:g</td>
<td>:r</td>
<td>:r</td>
</tr>
<tr>
<td>:r</td>
<td>:g</td>
<td>:b</td>
<td>:b</td>
<td>:g</td>
<td>:g</td>
</tr>
<tr>
<td>:r</td>
<td>:b</td>
<td>:g</td>
<td>:g</td>
<td>:b</td>
<td>:b</td>
</tr>
</tbody>
</table>

Diagram representation:
1 - RDF and SPARQL

- RDF Data Model(s)
- SPARQL by Example
- SPARQL and Entailment
- SPARQL algebra
Query processing with entailment

- Queries must take into account the RDF/S semantics (sub-class, sub-property)
- This semantics can be described by a set of logical RDF/S entailment rules.
- Solution 1: query rewriting
  - inject semantics into the query by rewriting
  - rewriting algorithms and generated queries are complex
- Solution 2: entailment rules evaluation
  - dynamically: during query evaluation (backward chaining)
  - statically: before query evaluation (forward chaining)

Assumption: data set = saturated RDF graph encoding all RDF/RDFS information.
1 - RDF and SPARQL

- RDF Data Model(s)
- SPARQL by Example
- SPARQL and Entailment
- SPARQL algebra
Example: ARQ-SPARQL

**SPARQL Algebra**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>triple $t$</td>
<td>all solutions $\nu$ of a triple pattern $t$. The result is a table with an attribute for each variable in $t$.</td>
</tr>
<tr>
<td>bgp ($bgp$)</td>
<td>joins $\Join$ the solutions (tables) of the triple patterns $t \in bgp$ in a basic graph pattern $bgp$</td>
</tr>
<tr>
<td>leftjoin ($e$) ($e'$)</td>
<td>left outer join $\Join$ of tables $e$ and $e'$</td>
</tr>
<tr>
<td>union ($e$) ($e'$)</td>
<td>union $\cup$ of tables $e$ and $e'$</td>
</tr>
<tr>
<td>project ($vars$) ($e$)</td>
<td>projection on the variables (attributes) $vars$ of table $e$</td>
</tr>
</tbody>
</table>

**Query ex1**

```sql
PREFIX : <http://asws.upmc.fr/exemples#>
SELECT ?title
FROM <ex1.ttl>
WHERE { :book1 :title ?title . }
```

**Algebraic expression for ex1**

```sql
(prefix ((: <http://asws.upmc.fr/exemples#>))
  (project (?title)
   (bgp (triple :book1 :title ?title))))
```
Simple pattern

Query couleur1a

PREFIX : <http://colors.org/ns#>
SELECT DISTINCT ?a ?b ?c ?d ?f ?g 
FROM <couleur.ttl>
WHERE {
}

Algebraic expression for couleur1a

(prefix (: <http://colors.org/ns#>))
(distinct
 project (?a ?b ?c ?d ?f ?g)
 (bgp
     (triple ?a :e ?b)
     (triple ?a :e ?c)
     (triple ?b :e ?a)
     (triple ?b :e ?c)
     (triple ?c :e ?a)
     (triple ?c :e ?b)
     (triple ?c :e ?f)
     (triple ?c :e ?g)
 ))) )
Query construct

PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX ns: <http://asws.org/ns#>

CONSTRUCT {
  ?x ns:prix ?price;
  ns:titre ?title .
}

FROM <ex5.ttl>

WHERE {
  { ?x ns:price ?price . } 
  UNION 
  { ?x dc:title ?title . } 
}

Algebraic expression for construct

(prefix ((dc: <http://purl.org/dc/elements/1.1/>)
  (ns: <http://asws.org/ns#>))

  (union
    (bgp (triple ?x ns:price ?price))
    (bgp (triple ?x dc:title ?title))))
OPTIONAL = ☰

Query optional10

PREFIX : <http://example.org/ns#>
SELECT ?x ?title ?y ?address
FROM <bibliosem.ttl>
WHERE {
OPTIONAL { ?x :editor :doesnotexist
OPTIONAL { ?y :address ?address } } }

Algebraic expression for optional10

(prefix (: <http://example.org/ns#>))
(project (?x ?title ?y ?address)
(leftjoin
(bgp (triple ?x :title ?title))
(leftjoin
(bgp (triple ?x :editor :doesnotexist))
(bgp (triple ?y :address ?address))))
Disjunctive normal form

Rewriting rules

1. \( \text{et} \cup \) are associative and commutative.
2. \((P1 \text{et} (P2 \cup P3)) \equiv ((P1 \text{et} P2) \cup (P1 \text{et} P3)).\)
3. \((P1 \triangledown (P2 \cup P3)) \equiv ((P1 \triangledown P2) \cup (P1 \triangledown P3)).\)
4. \(((P1 \cup P2) \triangledown P3) \equiv ((P1 \triangledown P3) \cup (P2 \triangledown P3)).\)
5. \(\sigma_R(P1 \cup P2) \equiv \sigma_R(P1) \cup \sigma_R(P2).\)

Disjunctive normal form

All graph patterns can be rewritten into a union of patterns without union:

\[ P \equiv \bigcup_{i} P_i \]

Join (and left outer join) are the main operations to be optimized.
Outline

1. RDF and SPARQL

2. RDF storage and SPARQL query evaluation
2 - RDF storage and SPARQL query evaluation

- RDF triple stores
  - Data organization and query evaluation
  - SPARQL query optimisation
  - Distributed SPARQL query processing
RDF triple stores I

One (simple) goal

Implement an efficient way of **storing and querying very large RDF graphs** (billions of triples)

Fundamental issue

Interaction between

- data organisation,
- data base technologies,
- query workload (query shapes).
## RDF triple stores II

Data organisation aims to facilitate join evaluation by achieving maximal data locality:

<table>
<thead>
<tr>
<th>Organisation</th>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted triple table</td>
<td>merge-join / replication</td>
</tr>
<tr>
<td>group by properties</td>
<td>vertical partitioning / compression</td>
</tr>
<tr>
<td>group by vertices</td>
<td>efficient star join</td>
</tr>
<tr>
<td>group by entities</td>
<td>denormalization / heterogeneity</td>
</tr>
<tr>
<td>group by query</td>
<td>locality for arbitrary query workloads</td>
</tr>
</tbody>
</table>
RDF triple stores III

Database technologies [14]

- **tables + relational algebra**: Jena, Virtuoso, DB2RDF, RDF-3X
- **graphs + matching**: gStore, chameleon-db
- **noSQL + map-reduce**: SHARD, HadoopRDF, JenaHBase, TrinityRDF

Data organisation and technology ⇒ Very large solution space.
RDF triple stores IV

Comparison criteria

- **Processing Cost:**
  - storage / data compression (memory, disc)
  - data loading / preprocessing
  - query execution (joins)

- **Deployment cost:**
  - implementation
  - configuration and tuning

- **Scalability:**
  - data distribution and replication
  - query parallelization

The rest of this talk mainly concentrates on *relational* storage schemes allowing for *algebraic* SPARQL query evaluation
Relational RDF: Choices

Query engine

- SQL queries
- relational select-project-join algebra + index (centralized)
- map-reduce algebra (distributed)

Query optimization

- query rewriting (logical)
- operator implementation (physical)
- parallelization / scalability
- index structures

Additional information

- RDF schema
- data statistics
- query workload
2 - RDF storage and SPARQL query evaluation

- RDF triple stores
- Data organization and query evaluation
- SPARQL query optimisation
- Distributed SPARQL query processing
Solution 1: Single triple table

One “big” table

Triples(subject, property, object)

- One single table storing all triples
- Systems: Jena, Oracle, RDF3X, Hexastore, 4store

Query join1

```sparql
SELECT ?title ?price
FROM <bibliosem.ttl>
```

```sql
SELECT t1.object as title, t2.object as price
FROM Triples t1, Triples t2
WHERE t1.property = ':title' and t2.property = ':price'
  and t1.subject = t2.subject;
```

+ simplicity (one single table), genericity (schema independent), compacity.
- Many auto-joins on a single big table → exhaustive indexing.
Solution 2: Property tables (Jena)

Tables regrouping properties

Entity \(_i\)\((\text{Subject, Property}_1, \ldots, \text{Property}_n)\)

- Regroup (clustering) de properties which are often shared by a same subject \((\text{group by entity})\).
- Systems: Jena, DB2RDF

Query simple1

```
SELECT ?title ?price ?editor
FROM <bibliosem.ttl>
```

SQL

```
select t1.title, t1.editor, t2.editor
from title_editor t1, price t2
where t1.subject = t2.subject;
```

+ less joins
- strong typing (schema) → DB2RDF [3]
Solution 3: Binary tables (group by property)

One table per property type

Property$_i$(Subject, Object)

- Vertical partitioning (group by property)
- Systems: SW-Store [1], Virtuoso, MonetDB

Query simple1

```
SELECT ?title ?price ?editor
FROM <bibliosem.ttl>
```

SQL

```
select t1.title, t2.editor, t3.editor
from title t1, editor t2, price t3
where t1.subject = t2.subject
and t2.subject = t3.subject
```

+ data partitioning, efficient property scan
- many (star) joins $\rightarrow$ sorting and efficient merge-join join implementation
2 - RDF storage and SPARQL query evaluation

- RDF triple stores
- Data organization and query evaluation
- SPARQL query optimisation
- Distributed SPARQL query processing
Query Optimization (SPARQL)

How can we improve performance?

- query simplification (remove redundant computation)
- algebraic rewriting (operation reordering)
- algebra operator implementation:
  - data organization: sorting and indexing
  - algorithms
- choice of underlying technologies:
  - relational DBMS
  - noSQL + Map/Reduce (Hadoop, SPARK)
  - native

Again, many options and solutions.
SPARQL query optimization

Query simplification

- Remove redundant triple patterns (“inverse” of saturation)
- Example:
  - Since `?x` binds to a property by definition, the second triple pattern can be removed.

Algebraic rewriting

- Push selections (selective triple patterns first)
- Join is **associative** and **commutative** → there exist $O(n!)$ equivalent join orderings.
- Join **evaluation cost** is sensitive to the order and the size of its arguments.
- Equivalent join plans are **more or less easy to parallelize**.

Optimal join plan = optimal join ordering.
Acyclic join-graphs

\[ t1: (?a :p :x) \quad t2: (?b :p :y) \]
\[ t3: (?c :p :x) \quad t4: (?c :q ?a) \]
\[ t5: (?d :r ?b) \quad t6: (?d :s ?a) \]

Join graph

Acylic ordered join graph

Linear plans
- (((t1, t4), t6), t3), t5), t2
- (((t1, t4), t3), t6), t5), t2
- (((t1, t4), t6), t5), t2), t3

Bushy plans
- ((t1, t6), (t3, t4)), (t2, t5)
- ((t1, t6), (t2, t5)), (t3, t4)
- ((t3, t4), (t1, t6)), (t2, t5)
Multiple access paths / exhaustive indexing

RDF-3X [13], Hexastore

- RDF triple store based on *exhaustive indexing* of RDF triple set:
  - input table: $T(S, P, O)$
  - one B+-tree index for each permutation of $S, P, O$ : $SPO$, $SOP$, $PSO$, $POS$, $OSP$, $OPS$ (and subsets [13])
  - each index defines an order $\rightarrow$ merge-join on B+-tree index

- Join ordering and index choice depends on the positions of constants (filtering) and join variables (join).
Why merge-join plans?

SPARQL Pattern

123 45 ?x . ?x 345 ?y . ?z 678 ?y

Join expression

\[
\left[ \sigma_{S=123, P=45} SPO \land \left( \sigma_{P=345} PSO \right) \right] \land \left( \sigma_{P=678} POS \right)
\]

- Tables POS, SPO and POS are lexicographically sorted.
- Scan tables in sort order: on each iteration, check join condition and generate result.

<table>
<thead>
<tr>
<th>S=123</th>
<th>P=45</th>
<th>O</th>
<th>P=345</th>
<th>S=?a</th>
<th>O</th>
<th>P=678</th>
<th>O=?b</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>3</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>123</td>
<td>45</td>
<td>23</td>
<td>123</td>
<td>23</td>
<td>45</td>
<td>123</td>
<td>45</td>
<td>23</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>123</td>
<td>45</td>
<td>56</td>
<td>123</td>
<td>56</td>
<td>78</td>
<td>123</td>
<td>56</td>
<td>78</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>123</td>
<td>45</td>
<td>78</td>
<td>345</td>
<td>23</td>
<td>67</td>
<td>345</td>
<td>23</td>
<td>67</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>345</td>
<td>67</td>
<td>23</td>
<td>345</td>
<td>34</td>
<td>188</td>
<td>345</td>
<td>34</td>
<td>188</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>678</td>
<td>67</td>
<td>99</td>
<td>678</td>
<td>99</td>
<td>67</td>
<td>678</td>
<td>99</td>
<td>67</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>678</td>
<td>103</td>
<td>456</td>
<td>678</td>
<td>456</td>
<td>103</td>
<td>678</td>
<td>103</td>
<td>456</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Linear cost: \( O(|SPO| + |PSO| + |POS|) \)
Example: Binary merge-join plans

Graph pattern

\[ \begin{align*}
  t1 & : (?a p x) & t2 & : (?b p y) \\
  t3 & : (?c p x) & t4 & : (?c q ?a) \\
  t5 & : (?d r ?b) & t6 & : (?d s ?a)
\end{align*} \]

Acyclic-directed join graph

Join Strategy

- Heuristics: Join selective patterns first.
- For example: \((t1 \bowtie_a t6) < (t4 \bowtie_a t6), (t3 \bowtie_a t6) < (t4 \bowtie_a t6)\) etc...

Resulting join plan

\(((t1 \bowtie_a t6) \bowtie_a (t3 \bowtie_c t4)) \bowtie_d (t2 \bowtie_b t5)\)
Join plan

**Logical join plan**

\[(t1 \bowtie_a t6) \bowtie_a (t3 \bowtie_c t4)) \bowtie_d (t2 \bowtie_b t5)\]

**Graph pattern**

- t1: (?a p x)
- t2: (?b p y)
- t3: (?c p x)
- t4: (?c q ?a)
- t5: (?d r ?b)
- t6: (?d s ?a)

**Index selection**

<table>
<thead>
<tr>
<th>patterns</th>
<th>constants</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1, t2, t3, t5*, t6*</td>
<td>P, O</td>
<td>POS</td>
</tr>
<tr>
<td>t4*</td>
<td>P, S</td>
<td>PSO</td>
</tr>
</tbody>
</table>

*: constants forwarded by join

**Physical join plan**

\[
\begin{align*}
  &\sigma_{p,x}(PO) \bowtie_a \sigma_{s}(POS) \\
  &\sigma_{p,x}(POS) \bowtie_a \sigma_{q}(PSO) \\
  &\sigma_{p,y}(POS) \bowtie_b \sigma_{r}(POS)
\end{align*}
\]
Variable graphs: n-ary join plans

**Goal**
Generate bushy n-ary join plan favorizing merge-joins

**Graph pattern**
- \( t_1: (?a : p : x) \)
- \( t_2: (?b : p : y) \)
- \( t_3: (?c : p : x) \)
- \( t_4: (?c : q ?a) \)
- \( t_5: (?d : r ?b) \)
- \( t_6: (?d : s ?a) \)

**Variable graph**
- \(?a,3\)
- \(?b,2\)
- \(?c,2\)
- \(?d,2\)

**Algorithm [18]**
- Compute **maximal independent node sets** of the variable graph
- Independent node sets (maximal): \{?c, ?d\}, \{?c, ?b\}, \{?a, ?b\}
- Each such set represents a set of n-ary joins that can be executed in parallel (independence).
- Apply heuristics (triple pattern selectivity, data organization, existing indexes) for choosing the optimal set and reordering joins.

**Physical join plan**
- \( \Join_{?c,?d} (\Join_{?a} (t_1, t_4, t_6), \Join_{?b} (t_2, t_5), t_3) \)
Variable graphs: n-ary join plans

Graph pattern

\[ t_1 : (\text{?a : p : x}) \quad t_2 : (\text{?b : p : y}) \]
\[ t_3 : (\text{?c : p : x}) \quad t_4 : (\text{?c : q : ?a}) \]
\[ t_5 : (\text{?d : r : ?b}) \quad t_6 : (\text{?d : s : ?a}) \]

Physical join plan

\[ \land_{\text{?c,?d}} (\land_{\text{?a}} (t_1, t_4, t_6), \land_{\text{?b}} (t_2, t_5), t_3) \]
2 - RDF storage and SPARQL query evaluation

- RDF triple stores
- Data organization and query evaluation
- SPARQL query optimisation
- Distributed SPARQL query processing
Distributed SPARQL query processing I

Goal

Implement scalable RDF triple stores capable to efficiently process SPARQL queries over very large triple sets.
Challenges

- **Re-use distributed data processing technology:**
  - distributed file system: partitioned triple files + table scan
  - key-value stores: enables usage of indexes (SPO,...)
  - graph-based: graph partitioning
  - Issue: **maximize data “locality”** (use replication if necessary)

- **Produce efficient parallelizable logical join plans**
  - binary versus n-ary join plans
  - left-deep versus flat bushy join-trees (CliqueSquare [6])

- **Implement efficient distributed join processing algorithms**
  - minimize data exchange between data processing nodes
  - hybrid join plans

*Again, many options ⇒ many solutions (see [10]*)
Distributed RDF-3X [9]

- partitioning RDF triple set graph on a cluster of nodes (METIS graph partitioner)
- each node stores its subgraph in an RDF-3X instance
- partial replication to avoid inter-machine communication
- parallel query execution with Hadoop:
  - check if query is parallelizable without communication (PWOC)
  - if yes: result = union of locally evaluated sub-queries
  - if not: decompose query pattern into subqueries (minimal edge partitioning of query graph) evaluated locally
SPARQL with Map-Reduce

Existing solutions

- S2RDF [16]
  - Join index
  - Single join algorithm

- CliqueSquare [6]
  - Flat bushy join-plans
  - Data replication (by subject, property and object)

- “SPARQL on SPARK” [12]
  - Goal: avoid replication, indexing, complex partitioning
  - Challenge: efficient join plans, saving data transfer
  - Approach: cost-based join optimization, combine different join algorithms
MapReduce Parallelism

**MapReduce Program**

**Data parallelism:** synchronous parallel tasks on independent data sets.

Three consecutive steps executed by \( m \) mappers \( M_i \) and \( r \) reducers \( R_j \):

- **initialization:** data set \( D \) is distributed to the \( m \) mappers \( M_i \).
- **map:** each mapper \( M_i \) **partitions** and **sorts** its local data set (tuples) \( D_i \) according to a **key-value** \( k \) to generate partitions \( P_i(k) \).
- **shuffle:** all partitions \( P_i(k) \) **with the same key-value** \( k \) **are sent to the same reducer** \( R_j \).
- **reduce:** some reducer \( R_j \) aggregates for a given \( k \) the partitions \( P_i(k) \) of all mappers.

**Example**

Use hash function \( h(x) = x \mod 3 \) and send partition \( P_i(k) \) for key value \( k \) where \( h(k) = 0 \) to reducer \( R1 \), \( h(k) = 1 \) to reducer \( R2 \) and \( h(k) = 2 \) to reducer \( R3 \).
MapReduce and Joins

Observations

- Data set can be partitioned and sorted at initialisation over a given set of computation nodes (for example on triple subject).
- Each node can play the role of one or several mappers and one or several reducers.
- The final result is distributed over all computation nodes (filter has no cost).
- It is possible to generate simple statistics at the initialization and during join computation.

Consequences

- Some joins can be executed completely locally depending on the initial partitioning scheme (for example star queries joining triples on their subject).
- Simple statistics can be used to estimate the cost (result size) of a particular join operation.

Join-processing with MapReduce

- Well-known problem studied by the database community.
- Two main join processing strategies: partitioned join and broadcast (replicated) join.
Partitioned join

Join \( t \bowtie_{?x} t' \)

- **map**: each mapper \( M_i \) generates, partitions and sorts the results of patterns \( t \) and \( t' \) according to the bindings \( k \) of join variable \(?x\) to generate partitions \( P_i(t, k) \) and \( P_i(t', k) \).
- **shuffle**: all partitions \( P_i(t, k) \) and \( P_i(t', k) \) with the same key-value \( k \) are sent to the same reducer \( R_j \).
- **reduce**: reducer \( R_j \) joins the partitions \( P_i(t, k) \) and \( P_i(t', k) \) received from all mappers \( M_i \) for a given join value \( k \) of set \( D \).

Example

- \( t: (?x :p :r1) \)
- \( t': (?a :q ?x) \)
- for a given resources \( :r \), all bindings for triple patterns \( (:r :p :r1 \text{ and } (?a :q :r) \) are sent to the same reducer \( R \) which computes the join.
- data transfer cost: \( O(b + b') \) where \( b \) and \( b' \) is the size of all bindings for both patterns \( t \) and \( t' \).
- the result is partitioned on \(?x\)
Broadcast join

Join $t \bowtie_{?x} t'$

- **map**: each mapper $M_i$ generates, partitions and sorts the results of patterns $t$ according to the **bindings** $k$ of join variable $?x$ to generate partitions $P_i(t, k)$ and $P_i(t', k)$.
- **shuffle**: each mapper broadcasts the partition corresponding to the smaller data set, say $B(t, k)$, to all computation nodes (reducers).
- **reduce**: each reducer $R_j$ **joins** all received partitions $P_i(t, k)$ bindings with its local partition $B_j(t', k)$.

**Example**

- $t$: (?x :p :r1) (generates less bindings)
- $t'$: (?a :q ?x)

- for a given node (mapper) $N$, all bindings produced by $N$ for triple pattern $t$ are sent to all nodes (reducers) which compute the join.
- data transfer cost: $O(b \times n)$ where $b$ is the size of bindings for $t$ and $n$ is the number of computation nodes.
- the result preserves the initial (target) partitioning.
Hybrid Join Plans [12]

Execution of a given plan $P$

Dynamic plan execution:

1. Evaluate all star subqueries $S = \{S_1, ..., S_n\}$ locally (n-ary join exploiting initial partitioning on triple subject)
2. Let $Temp = S_i \Join S_k$ the join generating the lowest cost (by choosing among broadcast join and partitioned join)
3. $S = S - \{S_i, S_k\}$
4. loop until $S$ is empty
   - Let $Temp = Temp \Join S_j$ the join generating the lowest cost
   - $S = S - \{S_j\}$
5. end loop

Observations

- The cost of each join is estimated using statistics obtained during initialization and query evaluation.
Experimental Results

Setup

- Cluster: 17 nodes (12 cores, 64GB memory), 1GB/s network
- SPARK: 16 worker nodes, 300 cores, 800 GB memory

Scalability

- Snowflake query Q8 from LUBM Benchmark
- 100M and 1B triples

Comparison with S2RDF [16]

- Query patterns: star (S1), snowflake (F5), complex (C3)
Conclusions

- My goal for this talk was to present some representative issues and solutions in efficient SPARQL/RDF processing.

- There are many open issues I did not develop here: statistical query optimisation, querying and reasoning, data compression, etc.

- For more details see for example [10,14].
Conclusions

My goal for this talk was to present some representative issues and solutions in efficient SPARQL/RDF processing.

There are many open issues I did not develop here: statistical query optimisation, querying and reasoning, data compression, etc.

For more details see for example [10,14].

Thank you for your attention!
Bibliographie I


C. Gutierrez, C. Hurtado, A. Mendelzon, Foundations of Semantic Web Databases, PODS 2004


Huang et al: Scalable SPARQL Querying of Large RDF Graphs. VLDB 2011


Bibliographie III


Bibliographie IV
