Query Optimization with RDF using SPARQL

By Lori London, Raul Cardenas and Rezwana Rimpi
**Semantic Web**: push towards the creation of a web of data. It sets the standards for the web.

RDF (Resource Description Framework)

RDF databases contain data in a form known as **triples**, formatted as: (Subject, Predicate, Object) or (S, P, O)

- **Subject** - a [URI](https://www.example.com/Lori) that is used as a [Resource](https://www.example.com/Lori)
- **Object** - a [URI](https://www.example.com/Lori) or [literal](https://www.example.com/Lori) that can be any primitive data type (i.e. float, String, integer)
- **Predicate** - a [URI](https://www.example.com/Lori) used as a [relationship](https://www.example.com/Lori) between the Subject and the Object in the triple

Prefix Lori: www.example.com/Lori

Ex. (Lori, property:age, 23)
In SQL, we are used to having tables and attributes define what data is in that particular table.

In RDF, we have the predicate column define what the triple is representing.

<table>
<thead>
<tr>
<th>SID</th>
<th>Title</th>
<th>Singer</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>“rain”</td>
<td>pid1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PID</th>
<th>Name</th>
<th>POB</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid1</td>
<td>John</td>
<td>Austin</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>hasType</td>
<td>“song”</td>
</tr>
<tr>
<td>id1</td>
<td>hastitle</td>
<td>“rain”</td>
</tr>
<tr>
<td>id1</td>
<td>sungBy</td>
<td>pid1</td>
</tr>
<tr>
<td>pid1</td>
<td>hasName</td>
<td>“John”</td>
</tr>
<tr>
<td>pid1</td>
<td>bornIn</td>
<td>“Austin”</td>
</tr>
</tbody>
</table>
SPARQL is the W3C standard as query language to RDF.

**Example SQL Select Statement**

```sql
SELECT title from Books
WHERE book_id = "book1"
```

**Example SPARQL Select Statement**

```sparql
SELECT ?title
WHERE { book1 hasTitle ?title }
```

A **triple pattern** is a predicate defined in the SPARQL where clause that indicates what kind of triple we are looking for.

A **query variable** is a component with a “?” that will return every value of that component in the database.
Examples Join Statements

```
SELECT  ?title ?price
WHERE   { ?x ns:price ?price .
          ?x dc:title ?title . }
```

```
SELECT  ?title
WHERE   { ?x ns:isType ?book .
```

Joins can happen on query variables only
Motivation

What is the difference from SQL querying on regular relational databases?

- Movement from structured schemas to partially structured schemas
- Queries explore unknown data structures
- Enabling joining and obtaining information from multiple datasets with one simple query
- Processing hundreds and hundreds of datasets is expensive ... so we optimize!
Topics for Today

○ Efficient Indexing Techniques
○ Optimization in Joins
○ Optimization using Selectivity Estimations
Efficient Indexing Techniques
To index an RDF triple, we index through an access pattern. An access pattern is a combination of how each component in the triple is specified, be it a literal or a variable:

1. ?x, ?y, ?z
2. s, ?y, ?z
3. ?x, p, ?z
4. ?x, ?y, o
5. s, p, ?z
6. s, ?y, o
7. ?x, p, o
8. s, p, o
Multiple Access Pattern (MAP)

Example:
select ?x
where{ ?x property:student "UT" }

Triple Pattern - (?x, property:student, “UT”)

POS - (property:student, “UT”, ?x)

OPS - (“UT”, property:student, ?x)
The RDF-3X Engine for Scalable Management of RDF data

Thomas Neumann · Gerhard Weikum
Published 1 September 2009. VLDB Journal
Composed of three different data structures, but today we are only focused on two of the structures:

**Mapping Dictionary** - for each component in an RDF triple, the component is mapped to an object id (OID)

<table>
<thead>
<tr>
<th>OID</th>
<th>Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>9</td>
<td>“CS”</td>
</tr>
<tr>
<td>4</td>
<td>Lori</td>
</tr>
<tr>
<td>15</td>
<td>“major”</td>
</tr>
<tr>
<td>24</td>
<td>“flower”</td>
</tr>
</tbody>
</table>
Triple Store Implementation (cont.)

**Compressed Index** - uses a MAP index pattern of a compressed RDF triple (a triple formed of its OIDs)

*Ex.*

insert data \{Lori, major, “CS” \} 

(Lori, major, “CS”) ->(4, 15, 9)

1. **SPO**-(4,15,9)  
2. **SOP**-(4,9,15)  
3. **POS**-(15,9,4)  
4. **PSO**-(15,4,9)  
5. **OPS**-(9,15,4)  
6. **OSP**-(9,4,15)
Query Processing and Translation for SPARQL is very similar to SQL with the exception of several nuances:

- Indexing on each Triple Pattern versus selecting one particular index
- Query Graph is based on Triple patterns versus relations
- Favors Bushy Join Trees versus Deep Left/Right Trees of R* Optimizer
**Nuance 1: Index Access Pattern for each Triple Pattern**

**Example:**
```
select ?u where{
?u <crime> .
?u <likes> "A.C. Doyle" .
?f <friend> ?u .
?f <romance> .
?f <likes> "J. Austen" .
}
```

**Indexing Patterns**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS</td>
<td>(crime, ?u)</td>
</tr>
<tr>
<td>OPS</td>
<td>(“A.C. Doyle”, likes, ?u)</td>
</tr>
<tr>
<td>POS</td>
<td>(friend, ?f, ?u)</td>
</tr>
<tr>
<td>PS</td>
<td>(romance, ?f)</td>
</tr>
<tr>
<td>OPS</td>
<td>(“J.Austen”, likes, ?f)</td>
</tr>
</tbody>
</table>
Nuance 2: Triple Pattern Query

Graph

**SQL**

P1 - ?u1 <crime> .

P2 - ?u2 <likes> “A.C. Doyle” .

P3 - ?u3 <friend> ?f1 .

P4 - ?f2 <romance> .


**SPARQL**

?u <crime> .

?u <likes> “A.C. Doyle” .

?u <friend> ?f .

?f <romance> .

Nuance 3: Bushy Join Trees versus Left/Right Deep Trees

- Attempts to use merge joins as much as possible
- Bottom-Top Dynamic Program is implemented to cache joins to increase efficiency

SQL R*

SPARQL RDF-3X
Optimization of Joins

Scalable Join Processing on Very Large RDF Graphs
Thomas Neumann and Gerhard Weikum
SPARQL queries over RDF map to SQL SELECT statements

○ So why not query RDF graph with the performance of SQL
○ Algorithms have been proven to be complete and sound
○ Adaption of relational databases algorithms to RDF and SPARQL
SPARQL Basic Graph Pattern is the conjunction of triple patterns, where each is matches the given attributes.

Assumptions of relational operations:
- Complete
- Sound
- Sequential

Idea run sequential operations on parallel

```sql
SELECT ?title ?price
WHERE { ?x ns:price ?price . T1
?x dc:title ?title . T2}
T1 Joins T2 on ?x
```
SPARQL Joins
SQL Joins => SPARQL Joins

RDBMS
Scan = Simple predicates filter the relations
1. Merge Join
2. Hash Join

SPARQL & MapReduce
Scan = Each triple pattern filters the graph
1. Merge Join
2. Hash Join
Triples of the form: $?x \text{ isType } ?y$
is not really selective and will return a large data set.

$?x ?y ?z$ is a really big problem. Hopefully, there are not many queries using this triple pattern.

This is true because for some queries only the conjunction of triple patterns as whole is selective.
Execution Plan on SPARQL

- A typical set of possible execution plans would include bushy trees!
- Bushy trees give more opportunity for parallelization
- Only 1 option for scan:
  - Index Scan
- Only 2 options for Joins:
  - Hash Join
  - Merge Join
Execution

- The scan operations are launched as the entry point of the pipeline
- Merge Joins are the next step in the pipeline
- A Hash Join would merge the two output streams into a single pipeline
- Bushy trees implies multiple joins running in different jobs
Execution Tree Plan

index scan $PS$ (romance, $f_1$)

index scan $POS$ (friend, $f_2, u_1$)

index scan $OPS$ (J.Austen, likes, $f_3$)

$MJ_{f_1=f_2}$

$MJ_{f_2=f_3}$

$MG_{u_1=u_2}$

$MG_{u_2=u_3}$

index scan $PS$ (crime, $u_2$)

index scan $OPS$ (A.C. Doyle, likes, $u_3$)
Sideways-Information-Passing SIP

**SIP**: Pass relevant information between separate joins at query runtime

**Goal**: Highly effective filters on the input stream of joins (Similar to magic sets)

This is a RDF-specific application of SIP. It enhances the filter on subject, predicate and object
“Sideways”: Pass information across operators in a way that cuts through the execution tree

- Restrict scans
- Prune the input stream
- Holistic, there is no data flow
Sideways-Information-Passing SIP

Merge Join

- Ascending order on the index value
- New constraint for each scan
  - $f_1 \geq f_2$
  - $f_2 \geq \max(f_1, f_3)$
  - $f_3 \geq \max(f_1, f_2)$
- The last values are recorded in the shared structure
Hash Join

- There is not direct comparison index value
- Use of domain filter (min, max)
  - 2 domains
    - Observed Domain
    - Potential Domain
  - Intersection of both
Index Scan

- It uses two previous techniques to skip and find “gaps” in the scan
- Index Scan are triple store in a B+tree
Sideways-Information-Passing SIP

- Results:
  - SIP aims to reduce the overhead of intermediate results
  - The higher in the tree the more accurate the domain filters become
  - SIP is still dependent on the execution order
    - Bad join order may to poor performance
    - Can we do better? Use selectivity and cardinality
Query Optimization using Selectivity Estimations

“SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation”
Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, Dave Reynolds
Basic Graph Pattern

○ Basic Graph Pattern or BGP? - set of triple patterns
  
  \[ ?x \text{ type Person} . \]
  \[ ?x \text{ hasSocialSecurityNumber "555−05−7880"} \]

○ Query Optimization Goal
  ■ To find an optimized execution plan
  ■ That means, to find the optimized order of executing the triple patterns
Triple Pattern Selectivity

- Def.: Fraction of *RDF data triples* satisfying the *triple pattern*.

- Selectivity of a triple pattern $t = (s, p, o)$,
  - $\text{sel}(t) = \text{sel}(s) \times \text{sel}(p) \times \text{sel}(o)$
  - Assumption: $\text{sel}(s)$, $\text{sel}(p)$, $\text{sel}(o)$ are statistically independent.
● Selectivity of Predicate
  ○ $\text{sel}(p) = \frac{T_p}{T}$, when $p$ is bound
    here, $T_p$ = number of triples matches $P$
    $T$ = Total number of triples in RDF
  ○ $\text{sel}(p) = 1$, when $p$ is a variable
Joined Triple Pattern

- **Joined Triple pattern**
  - A pair of triple patterns that share a variable
    - Return the name of person who have SocialSecurityNumber = “555-05-7880”.
      ```sql
      select ?x where{
        ?x type Person .
        ?x hasSocialSecurityNumber "555-05-7880"}
      ```
  - Size - the size of the result set satisfying the two patterns
Let $P$ represent a Joined Triple pattern

$$\text{sel}(P) = \frac{S_p}{T^2}, \text{where}$$

$S_p = \text{upper bound size Joined Triple pattern } P$

$T = \text{total number of triples in RDF dataset}$
Basic Graph Pattern Optimization

BGP
1 ?X rdf : type ub : GraduateStudent .
2 ?Y rdf : name ub : University .
3 ?Z rdf : dept ub : Department .
6 ?X ub : undergraduateDegreeFrom ?Y .

node: a triple pattern
edge: joined triple pattern
Basic Graph Pattern Optimization

BGP

1 \(?X\) rdf : type ub : GraduateStudent .
2 \(?Y\) rdf : name ub : University .
3 \(?Z\) rdf : dept ub : Department .
4 \(?X\) ub : memberOf \(?Z\) .
5 \(?Z\) ub : subOrganizationOf \(?Y\) .
6 \(?X\) ub : undergraduateDegreeFrom \(?Y\) .

Execution plan: an order of nodes.
An order to join the triple patterns
Ex. 1, 2, 4, 3, 5, 6
Deterministic Execution Plan Generation

Node selectivity is Triple Pattern Selectivity
Edge selectivity is Joined Triple Pattern Selectivity

Input

Output Execution plan
Execution Plan Generation (contd.)

Step 1

Edges is ascending order of selectivity
(6,5)
(6,2)
(3,4)
(1,6)
(3,5)
(4,5)
(4,6)
(5,2)
(1,4)

Sink: 5
Execution Plan Generation (contd.)

Step 2

Sink: 5-6

Edges is ascending order of selectivity:
- (6,5)
- (6,2)
- (3,4)
- (1,6)
- (3,5)
- (4,5)
- (4,6)
- (5,2)
- (1,4)
Execution Plan Generation (contd.)

Step 3

Sink: 5-6-2

Edges in ascending order of selectivity:
- (6,5)
- (6,2)
- (3,4)
- (1,6)
- (3,5)
- (4,5)
- (4,6)
- (5,2)
- (1,4)
Step 4

Sink: 5-6-2

Edges is ascending order of selectivity

(6,5)
(6,2)
(3,4)
(1,6)
(3,5)
(4,5)
(4,6)
(5,2)
(1,4)
Execution Plan Generation (contd.)

Step 4 (contd.)

Sink: 5-6-2-1

Edges is ascending order of selectivity:
- (6,5)
- (6,2)
- (3,4)
- (1,6)
- (3,5)
- (4,5)
- (4,6)
- (5,2)
- (1,4)
Execution Plan Generation (contd.)

Step 5

Edges is ascending order of selectivity

(6,5)
(6,2)
(3,4)
(1,6)
(3,5)
(4,5)
(4,6)
(5,2)
(1,4)

Sink: 5-6-2-1-3
Execution Plan Generation (contd.)

Step 6

Sink: 5-6-2-1-3-4
Execution Plan: 5, 6, 2, 1, 3, 4

Edges is ascending order of selectivity
(6,5)
(6,2)
(3,4)
(1,6)
(3,5)
(4,5)
(4,6)
(5,2)
(1,4)
Deterministic Algorithm

**Algorithm 1** Find optimized execution plan $EP$ for $g \in G$

\[\begin{align*}
N & \leftarrow \text{Nodes}(g) \\
E & \leftarrow \text{Edges}(g) \\
EP & \leftarrow \text{select minimum selectivity edge} \ xy \\
\text{if sel}(x) \leq \text{sel}(y) \text{ then} \\
\text{sink} & = x \\
\text{else } \text{sink} & = y \\
& \text{while size}(EP) \leq \text{size}(N) \text{ do} \\
& \quad e \leftarrow \text{SelectEdgeMinSel}(E) \\
& \quad EP \leftarrow \text{OrderNodesBySel}(e) \\
& \quad EP \leftarrow \text{SelectNotVisitedNode}(EP, e) \\
& \text{end while} \\
& \text{return } EP
\end{align*}\]

**Select Sink** (*Deterministically*):
- select the **minimum selectivity edge** $xy$
- if $\text{sel}(x) \leq \text{sel}(y)$ then
  - $\text{sink} = x$
- else $\text{sink} = y$

**Main Loop**: While there is a *non-visited* node $xy$ <- **Next minimum selectivity edge**
- if *one of its endpoint is visited* (say $x$ is visited), then
  - add $y$ to the execution plan
  - make $y$ visited
What about disconnected graph?

- Graph G may have more than one component
- Like System-R algorithm, take cross product of result sets of components.
Properties

- *Deterministic execution plan* based on selectivity estimations.
- Size of intermediate result set is reduced.
- Cartesian product of intermediate results is avoided within a component.
You now know about basic Query Optimization in RDF with SPARQL! SPARQL optimizer will have all of three fundamentals that we spoke about today:

- Due to the simplicity of the RDF model, we are allowed to index on every component in an RDF triple
- SPARQL involves many joins in their queries, and thus we must be aware of only executing the most optimal of query plans
- With SPARQL having deterministic solutions, we do not have to exhaust the entire search space
Thank You

Questions?