On the Efficient Distributed Evaluation of SPARQL Queries

Damien Graux

Supervisor: Nabil Layaida
Co-Supervisor: Pierre Geneves
Funded by: Datalyse Project

Universite Grenoble Alpes, INRIA, LIG

Tyrex Team
<tyrex.inria.fr>

December 15th, 2016
A practical usecase:

What did you miss (touristically) last time you travelled (by plane)?
A practical usecase:

What did you miss (touristically) last time you travelled (by plane)?

More specifically: “Is it possible to sightsee at stopovers?”
A practical usecase:

What did you miss (touristically) last time you travelled (by plane)?

More specifically: “Is it possible to sightsee at stopovers?”
A practical usecase:

What did you miss (touristically) last time you travelled (by plane)?

More specifically: “Is it possible to sightsee at stopovers?”

Planes
- Relational

Subways
- GTFS

POIs
- RDF

Reviews
- Various
Context & Objectives driven by an example

A practical usecase:

What did you miss (touristically) last time you travelled (by plane)?

More specifically: “Is it possible to sightsee at stopovers?”

Planes
- Relational
- Thousand

Subways
- GTFS
- Million

POIs
- RDF
- Billion

Reviews
- Various
- Billion
Context & Objectives driven by an example

A practical use case:

What did you miss (touristically) last time you travelled (by plane)?

More specifically: “Is it possible to sightsee at stopovers?”

Planes
- Relational
- Thousand
- Static

Subways
- GTFS
- Million
- Static

POIs
- RDF
- Billion
- Dynamic

Reviews
- Various
- Billion
- Dynamic
A practical use case:

What did you miss (touristically) last time you travelled (by plane)?

More specifically: “Is it possible to sightsee at stopovers?”

Planes
- Relational
- Thousand
- Static

Subways
- GTFS
- Million
- Static

POIs
- RDF
- Billion
- Dynamic

Reviews
- Various
- Billion
- Dynamic

Finally,...

... Linking the blocks!
A practical usecase:

What did you miss (touristically) last time you travelled (by plane)?

More specifically: “Is it possible to sightsee at stopovers?”

Complex Problem

Planes
- Relational
- Thousand
- Static

Subways
- GTFS
- Million
- Static

POIs
- RDF
- Billion
- Dynamic

Reviews
- Various
- Billion
- Dynamic

Finally,…

… Linking the blocks!
Context & Objectives driven by an example

Context:
- Large datasets available
- Heterogeneous data
Context & Objectives driven by an example

Context:
- Large datasets available
- Heterogeneous data

Objectives:
- Efficiently request these datasets
- Aggregate results to build complex applications
Context & Objectives driven by an example

A practical usecase:

What did you miss (touristically) last time you travelled (by plane)?

More specifically: “Is it possible to sightsee at stopovers?”

Complex Problem

Finally, . . .

. . . Linking the blocks!
My PhD topic

Focuses

1. Focusing on evaluating SPARQL queries,
2. On large amounts of RDF data,
3. In a distributed context.
My PhD topic

Focuses

1. Focusing on evaluating SPARQL queries,
2. On large amounts of RDF data,
3. In a distributed context.

Problem

How to design efficient distributed SPARQL evaluators?
Section 1

RDF & SPARQL
Resource Description Framework [HM04]
Resource Description Framework [HM04]
Resource Description Framework [HM04]

RDF essentials

- RDF is a W3C standard
- RDF designed to provide, share and exchange datasets
- An RDF graph is a set of RDF triples
- An RDF triple has three components:
  - a subject (s)
  - a predicate (p)
  - a object (o)
**SPARQL Protocol and RDF Query Language** [G+13]

SELECT ?s ?g WHERE {
  ?s type Museum
  ?g type Painter
  ?s shows ?g
}
SPARQL Protocol and RDF Query Language [G+13]

?-s type Museum
?-g type Painter
?-s shows ?g

?s: Ducth School, Louvre
SPARQL Protocol and RDF Query Language [G+13]

?s type Museum
?g type Painter
?s shows ?g

?s: Dutch School, Louvre
?g: Rembrandt, Hals, Vermeer, Van Dyck
SPARQL Protocol and RDF Query Language [G+13]

?s type Museum
?g type Painter
?s shows ?g

?s: Dutch School, Louvre
?g: Rembrandt, Hals, Vermeer, Van Dyck
(?s,?g): (Dutch School, Rembrandt), (Dutch School, Hals), (Dutch School, Vermeer), (Dutch School, Van Dyck), (Collection, Rembrandt)
SPARQL Protocol and RDF Query Language [G+13]

\[ ?s \text{ type } \text{Museum} \\
?g \text{ type } \text{Painter} \\
?s \text{ shows } ?g \]

\[ ?s: \text{Dutch School, Louvre} \]
\[ ?g: \text{Rembrandt, Hals, Vermeer, Van Dyck} \]
\[ (?s,?g): (\text{Dutch School,Rembrandt}), (\text{Dutch School,Hals}), (\text{Dutch School,Vermeer}), \]
\[ (\text{Dutch School,Van Dyck}), (\text{Collection,Rembrandt}) \]

**Solution** \[ (?s,?g): (\text{Dutch School,Rembrandt}), (\text{Dutch School,Hals}), (\text{Dutch School,Vermeer}), (\text{Dutch School,Van Dyck}) \]
Considered SPARQL Fragment

- **Basic Graph Pattern** (BGP) fragment composed of conjunctions of Triple Patterns (TPs).
- **Triple Pattern** (TP)

```sql
SELECT ?s ?g WHERE {
  ?s type Museum
  ?g type Painter
  ?s shows ?g
}
```

- One BGP
- Composed of 3 TPs
Considered SPARQL Fragment

- **Basic Graph Pattern** (BGP) fragment composed of conjunctions of Triple Patterns (TPs).
- **Triple Pattern** (TP)

SELECT ?s ?g WHERE {
    ?s type Museum
    ?g type Painter
    ?s shows ?g
}

- One BGP
- Composed of 3 TPs

Solutions

- A *candidate solution* satisfies a TP when the replacement of the variables of the TP with their value corresponds to a triple that appears in the RDF data.
- A *query solution* is a candidate solution that satisfies all the TPs of the query.
Section 2

Distributed Frameworks
MapReduce Strategy

The paradigm

- Parallel processing of massive datasets [DG08]
- A job has two separate phases:
  1. Map phase which takes k/v pairs, performs computations and returns k/v pairs
  2. Reduce phase where k/v pairs from the Map are ingested to return a single set of results.
- Intermediate results sometimes need to be shuffled – exchanged and/or merge-sorted – across the network to be reduced.

In brief, MapReduce

proposes to not only consider dataset as distributed and fragmented on each machine but also to develop the computation as small blocks (the Map part) which are finally grouped together (the Reduce part).
Distributed Frameworks

Hadoop

- Framework for distributed systems based on MapReduce
- It is twofold:
  - a distributed file system (including replication)
  - a MapReduce library

Cluster Computing Frameworks

- Provide an interface with implicit data parallelism and fault-tolerance
- Offer a set of low-level functions e.g. map, join, collect...
- For instance: PigLatin, Flink, Spark...
Spark in a nutshell

- Master/Worker(s) Architecture
- Various file system sources supported e.g. HDFS
- One of the most active Apache project e.g. 1000+ contributors

2002 MapReduce @ Google

2004 MapReduce Paper

2006 Hadoop @ Yahoo!

2008 Hadoop Summit

2010 Spark Open-Source

July 2016 Apache Spark 2.0

May 2014 Apache Spark 1.0

Apache Spark\textsuperscript{[ZCD+12]}

Spark in a nutshell

- Master/Worker(s) Architecture
- Various file system sources supported \textit{e.g.} HDFS
- One of the most active Apache project \textit{e.g.} 1000+ contributors

Resilient Distributed Datasets

- Distributed object collections
- Split into \textit{partitions} stored in RAM or disks
- Created through deterministic operations
- Fault-tolerant: automatically re-built
Section 3

SPARQL Evaluators
Jumble of Evaluators

4store
CouchBaseRDF
BitMat
YARS
Hexastore
CliqueSquare
RYA
Parliament
Virtuoso
RDF-3X
...
### Jumble of Evaluators

<table>
<thead>
<tr>
<th>When?</th>
<th>Who?</th>
<th>What?</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td>Barstow [Bar01]</td>
<td>Focuses on open-source solutions; and looks at some of their specificities</td>
</tr>
<tr>
<td>2002</td>
<td>Beckett [Bec02]</td>
<td>Updates</td>
</tr>
<tr>
<td>2003</td>
<td>Beckett [BG03]</td>
<td>Focuses on the use of relational database management systems to store RDF datasets</td>
</tr>
<tr>
<td>2004</td>
<td>Lee [Lee04]</td>
<td>Updates</td>
</tr>
<tr>
<td>2012</td>
<td>Faye [FCB12]</td>
<td>Lists the various RDF storage approaches mainly used by single-node systems</td>
</tr>
<tr>
<td>2015</td>
<td>Kaoudi [KM15]</td>
<td>Presents a survey focusing only on RDF in the clouds</td>
</tr>
</tbody>
</table>
RDF Storage Strategies

- **native**
  - In-memory
    - Standalone
    - Embedded
  - On Disks

- **non-native**
  - Web APIs
    - Schema-Carefree
      - Triple Table
  - DBMS-based
    - Schema-Aware
      - Vertical Partitioning
      - Property Table
RDF Storage Strategies

- **native**
  - In-memory
    - BitMat
  - On Disks
    - Standalone
      - Virtuoso
      - RDF-3X
    - Embedded

- **non-native**
  - Web APIs
    - Schema-Carefree
      - Triple Table
        - 3store
    - Schema-Aware
      - Vertical Partitioning
        - swStore
      - Property Table
Distributed Evaluation Methods

Distributed RDF Storage Methods

- Federation
  - Horizontal Fragmentation
  - Graph Partitioning
- Key-Value Stores
  - Triple-based
  - Graph-based
- Independent
- Distributed File System
  - Triple Table
  - Vertical Partitioning
  - Property Table
Distributed Evaluation Methods
Distributed SPARQL Evaluator State-of-the-art Summary

Observations

1. Multiple RDF storage strategies
2. Various methods to distribute data and to compute queries
Observations

1. Multiple RDF storage strategies
2. Various methods to distribute data and to compute queries

How to pick an efficient evaluator?
Distributed SPARQL Evaluator State-of-the-art Summary

**Observations**

1. Multiple RDF storage strategies
2. Various methods to distribute data and to compute queries

**How to pick an efficient evaluator?**

Experimental Evaluation!
Section 4

Multi-Criteria Experimental Ranking
## Experimental Studies

<table>
<thead>
<tr>
<th>When?</th>
<th>Who?</th>
<th>What?</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002</td>
<td>Magkanaraki [MKA⁺02]</td>
<td>Reviews solutions dealing with ontologies</td>
</tr>
<tr>
<td>2009</td>
<td>Stegmaier [SGD⁺09]</td>
<td>Reviews solutions according to several parameters such as their licenses, their architectures and compares them using a scalable test dataset</td>
</tr>
<tr>
<td>2013</td>
<td>Cudré [CMEF⁺13]</td>
<td>Realizes an empirical study of distributed SPARQL evaluators (native RDF stores and several NoSQL solutions they adapted)</td>
</tr>
</tbody>
</table>
### Popular Benchmarks

<table>
<thead>
<tr>
<th>Name</th>
<th>SPARQL Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUBM [GPH05]</td>
<td>BGP</td>
</tr>
<tr>
<td>WatDiv [AHÖD14]</td>
<td>BGP</td>
</tr>
<tr>
<td>SP²Bench [SHLP09]</td>
<td>BGP + FILTER UNION OPTIONAL + Solution Modifiers + ASK</td>
</tr>
<tr>
<td>BolowgnaB [DEW¹¹¹]</td>
<td>BGP + aggregator (e.g. COUNT)</td>
</tr>
<tr>
<td>BSBM [BS09]</td>
<td>BGP + FILTER UNION OPTIONAL + Solution Modifiers + Logical negation + CONSTRUCT</td>
</tr>
<tr>
<td>DBPSB [MLAN11]</td>
<td>Use actually posed queries against dbpedia</td>
</tr>
<tr>
<td>RBench [QÖ15]</td>
<td>Generate queries according to considered datasets</td>
</tr>
</tbody>
</table>
## Popular Benchmarks

<table>
<thead>
<tr>
<th>Name</th>
<th>SPARQL Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LUBM [GPH05]</strong></td>
<td>BGP</td>
</tr>
<tr>
<td><strong>WatDiv [AHÖD14]</strong></td>
<td>BGP</td>
</tr>
<tr>
<td><strong>SP²Bench [SHLP09]</strong></td>
<td>BGP + FILTER UNION OPTIONAL + Solution Modifiers + ASK</td>
</tr>
<tr>
<td><strong>BolowgnaB [DEW+11]</strong></td>
<td>BGP + aggregator (e.g. COUNT)</td>
</tr>
<tr>
<td><strong>BSBM [BS09]</strong></td>
<td>BGP + FILTER UNION OPTIONAL + Solution Modifiers + Logical negation + CONSTRUCT</td>
</tr>
<tr>
<td><strong>DBPSB [MLAN11]</strong></td>
<td>Use actually posed queries against dbpedia</td>
</tr>
<tr>
<td><strong>R Bench [QÖ15]</strong></td>
<td>Generate queries according to considered datasets</td>
</tr>
</tbody>
</table>
## Contrib. 1 – Experimental Comparative Analysis

### Considered Benchmarks
- LUBM: generated datasets and 14 queries (Q1-Q14)
- WatDiv: generated datasets and 20 queries

### Competitors
- Selection criteria: OpenSource, Popular or Recent
- Two types of evaluators:
  - Conventional (with preprocessing): 4store, CumulusRDF, CouchBaseRDF, RYA, CliqueSquare and S2RDF
  - Direct: PigSPARQL
Contrib. 1 – Obtained Results

We learned:

1. Considering the same dataset, loading times are spread over several magnitude orders.
### Contrib. 1 – Obtained Results

With the following RDF datasets:

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of Triples</th>
<th>Original File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>WatDiv1k</td>
<td>109 million</td>
<td>15 GB</td>
</tr>
<tr>
<td>Lubm1k</td>
<td>134 million</td>
<td>23 GB</td>
</tr>
<tr>
<td>Lubm10k</td>
<td>1.38 billion</td>
<td>232 GB</td>
</tr>
</tbody>
</table>

**Figure**: Preprocessing Time.
Contrib. 1 – Obtained Results

We learned:

1. Considering the same dataset, loading times are spread over several magnitude orders.
2. For the same query on the same dataset, elapsed times can differ very significantly.
Contrib. 1 – Obtained Results

Q1

```sparql
SELECT ?X WHERE {
  ?X rdf:type ub:GraduateStudent .
  ?X ub:takesCourse GraduateCourse0
}
```

Q2

```sparql
SELECT ?X ?Y ?Z WHERE {
  ?X rdf:type ub:GraduateStudent .
  ?Y rdf:type ub:University .
  ?Z rdf:type ub:Department .
  ?X ub:undergraduateDegreeFrom ?Y
}
```

Q3

```sparql
SELECT ?X WHERE {
  ?X ub:publicationAuthor AssistantProfessor0
}
```

Figure: Query Response Time with Lubm1k (134 million triples).
Contrib. 1 – Obtained Results

We learned:

1. Considering the same dataset, loading times are spread over several magnitude orders.
2. For the same query on the same dataset, elapsed times can differ very significantly.
3. Even with large datasets, most queries are not harmful *per se*, *i.e.* queries that incur long running times with some implementations still remain in the “comfort zone” for other implementations.
Contribution 1 – Obtained Results

(a) 4store

(b) S2RDF

(c) RYA

(d) PigSPARQL

Figure: Obtained results with WatDiv1k.
**Contrib. 1 – Obtained Results**

We learned:

1. Considering the same dataset, loading times are spread over several magnitude orders

2. For the same query on the same dataset, elapsed times can differ very significantly

3. Even with large datasets, most queries are not harmful *per se*, *i.e.* queries that incur long running times with some implementations still remain in the “comfort zone” for other implementations

**Ok, but...**

... how to rank evaluators? 😊
An extended set of metrics

**Usual metrics:**
- Time
- Disk Footprint

**Our additions:**
- Disk Activity
- Network Traffic
- Resources: CPU, RAM, SWAP
An extended set of metrics

Usual metrics:
- Time, always
- Disk Footprint, only sometimes

Our additions:
- Disk Activity, new
An extended set of metrics

Usual metrics:
- Time: always
- Disk Footprint: only sometimes

Our additions:
- Disk Activity: new
- Network Traffic: new
An extended set of metrics

Usual metrics:
- Time 
  always
- Disk Footprint 
  only sometimes

Our additions:
- Disk Activity 
  new
- Network Traffic 
  new
- Resources: CPU, RAM, SWAP 
  new
Contrib. 2 – Multi-Criteria Reading Grid

Criteria List

- **Velocity**: the fastest possible answers

**Query Time**
## Contrib. 2 – Multi-Criteria Reading Grid

### Criteria List

- **Velocity**: the fastest possible answers

- **Resiliency**: trying to avoid as much as possible to recompute everything when a machine fails

*Query Time*

*Footprint*
## Criteria List

- **Velocity**: the fastest possible answers
  
  *Query Time*

- **Resiliency**: trying to avoid as much as possible to recompute everything when a machine fails
  
  *Footprint*

- **Immediacy**: evaluating some SPARQL queries only once
  
  *Preprocessing Time*
Contrib. 2 – Multi-Criteria Reading Grid

Criteria List

- **Velocity**: the fastest possible answers
  
  *Query Time*

- **Resiliency**: trying to avoid as much as possible to recompute everything when a machine fails
  
  *Footprint*

- **Immediacy**: evaluating some SPARQL queries only once
  
  *Preprocessing Time*

- **Dynamicity**: dealing with dynamic data
  
  *Preprocessing Time & Disk Activity*
Contrib. 2 – Multi-Criteria Reading Grid

Criteria List

- **Velocity**: the fastest possible answers

- **Resiliency**: trying to avoid as much as possible to recompute everything when a machine fails

- **Immediacy**: evaluating some SPARQL queries only once

- **Dynamicity**: dealing with dynamic data

- **Parsimony**: minimizing some of the resources
Contrib. 2 – Ranking

How to design efficient SPARQL evaluators?

We first need to specify the criteria...

- Immediacy
- Velocity
- WatDiv1k
- Parsimony
- Lubm1k
- Dynamicity
- Resiliency
Contrib. 2 – Ranking

- Immediacy
- Parsimony
- Dynamicity
- Resiliency
- Velocity
  - WatDiv1k
  - Lubm1k

How to design efficient SPARQL evaluators?
We first need to specify the criteria.
Contrib. 2 – Ranking

How to design efficient SPARQL evaluators?

We first need to specify the criteria...
Contrib. 2 – Ranking

How to design efficient SPARQL evaluators?

We first need to specify the criteria...
Section 5

Efficient Distributed SPARQL Evaluation
Contrib. 3 – Efficient Distributed SPARQL evaluation

We designed:

- SPARQLGX
- SDE
- RDFHive

Available from: <https://github.com/tyrex-team>
Contrib. 3 – Efficient Distributed SPARQL evaluation

These evaluators in nutshells:

- **SPARQLGX**  a distributed SPARQL evaluator with Apache Spark
- **SDE**  a direct SPARQL evaluator with Apache Spark
- **RDFHive**  a direct evaluation of SPARQL with Apache Hive

Available from: <https://github.com/tyrex-team>
Contrib. 3 – Efficient Distributed SPARQL evaluation

Considering the reading grid, we have:

- **SPARQLGX**  velocity, resiliency
- **SDE**  immediacy, dynamicity, resiliency
- **RDFHive**  immediacy, dynamicity, resiliency, parsimony

Available from: <https://github.com/tyrex-team>
Details of SPARQLGX

1. Selected storage model
2. SPARQL translation process
3. Optimization strategies
Vertical Partitioning [Abadi et al. 2007]
SPARQLGX Storage Model

RDF *predicates* carry the semantic information, thereby:

- Limited number of distinct predicates *e.g.* few hundreds [Gallego et al. 2011]
- Predicates rarely variable in queries [Gallego et al. 2011]
RDF *predicates* carry the semantic information, thereby:

- Limited number of distinct predicates *e.g.* few hundreds [Gallego *et al.* 2011]
- Predicates rarely variable in queries [Gallego *et al.* 2011]

**Vertical Partitioning**
Splitting by predicate and saving two-column files
Vertical Partitioning [Abadi et al. 2007]  
SPARQLGX Storage Model

<table>
<thead>
<tr>
<th>RDF predicates carry the semantic information, thereby:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Limited number of distinct predicates e.g. few hundreds [Gallego et al. 2011]</td>
</tr>
<tr>
<td>- Predicates rarely variable in queries [Gallego et al. 2011]</td>
</tr>
</tbody>
</table>

**Vertical Partitioning**
Splitting by predicate and saving two-column files

**Advantages**
- Natural compression and indexing
- Straightforward implementation
**Vertical Partitioning** [Abadi et al. 2007]

SPARQLGX Storage Model

<table>
<thead>
<tr>
<th>dataset</th>
<th>type.txt</th>
<th>type.txt</th>
<th>type.txt</th>
<th>type.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dutch School</td>
<td>type</td>
<td>Museum</td>
<td>Museum</td>
<td>Museum</td>
</tr>
<tr>
<td>Dutch School</td>
<td>creationDate</td>
<td>2016</td>
<td>Louvre</td>
<td>Painter</td>
</tr>
<tr>
<td>Dutch School</td>
<td>use</td>
<td>Museum</td>
<td>Hals</td>
<td>Painter</td>
</tr>
<tr>
<td>Dutch School</td>
<td>Rembrandt</td>
<td>Painter</td>
<td>Vermeer</td>
<td>Painter</td>
</tr>
<tr>
<td>Louvre</td>
<td>type</td>
<td>Painter</td>
<td>Van Dyck</td>
<td>Painter</td>
</tr>
<tr>
<td>Rembrandt</td>
<td>type</td>
<td>Painter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hals</td>
<td>type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vermeer</td>
<td>type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Van Dyck</td>
<td>type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Collection</td>
<td>shows</td>
<td>Rembrandt</td>
<td></td>
<td>Rembrandt</td>
</tr>
<tr>
<td>Dutch School</td>
<td>mainTopic</td>
<td>Rembrandt</td>
<td>Rembrandt</td>
<td>Rembrandt</td>
</tr>
<tr>
<td>Dutch School</td>
<td>shows</td>
<td>Rembrandt</td>
<td>Hals</td>
<td>Hals</td>
</tr>
<tr>
<td>Dutch School</td>
<td>shows</td>
<td>Rembrandt</td>
<td>Vermeer</td>
<td>Vermeer</td>
</tr>
<tr>
<td>Dutch School</td>
<td>shows</td>
<td>Van Dyck</td>
<td></td>
<td>Van Dyck</td>
</tr>
<tr>
<td>Dutch School</td>
<td>shows</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**type.txt**

| Dutch School     | Museum                 | Museum                 | Museum                 | Museum                 |
| Louvre           | type                   | Painter                | Painter                | Painter                |
| Rembrandt        | type                   | Painter                | Painter                | Painter                |
| Hals             | type                   | Painter                | Van Dyck               | Painter                |
| Vermeer          | type                   | Van Dyck               |                        |                        |
| Van Dyck         | type                   |                        |                        |                        |

**creationDate.txt**

| Dutch School     | 2016                   |                        |                        |                        |

**use.txt**

| Dutch School     | Louvre                 |                        |                        |                        |

**shows.txt**

| Collection       | Rembrandt              | Rembrandt              | Rembrandt              | Rembrandt              |
| Dutch School     | Rembrandt              | Hals                   | Vermeer                | Van Dyck               |
| Dutch School     | Vermeer                | Van Dyck               |                        |                        |
| Dutch School     | Van Dyck               |                        |                        |                        |

**mainTopic.txt**

| Dutch School     | Rembrandt              | Hals                   | Vermeer                | Van Dyck               |
| Dutch School     | Rembrandt              | Hals                   | Vermeer                | Van Dyck               |
| Dutch School     | Vermeer                | Van Dyck               |                        |                        |
| Dutch School     | Van Dyck               |                        |                        |                        |
Dealing with one TP …

- `textFile` to access relevant files
- `filter` to keep matching triples
Dealing with one TP ...

- `textFile` to access relevant files
- `filter` to keep matching triples

\[
\text{?s type Museum .}
\text{\hspace{100pt}}\text{textFile("type.txt")}
\text{\hspace{100pt}}.filter\{\text{case(s,o)} \Rightarrow \text{o.equals("Museum")}\}
\text{\hspace{100pt}}.map\{\text{case(s,o)} \Rightarrow s\}
\]
Dealing with one TP ...

- `textFile` to access relevant files
- `filter` to keep matching triples

```scala
?s type Museum .
```

```
textFile("type.txt")
.filter{case(s,o)=&gt;o.equals("Museum")}
.map{case(s,o)=&gt;s}
```

... with a conjunction of TPs

- Translate each TP
- Join them one by one
SPARQL Translation Process
SPARQL → Scala

?s type Museum .
?g type Painter .
?s shows ?g
SPARQL Translation Process
SPARQL→Scala

?s type Museum .
?g type Painter .
?s shows ?g

tp1=sc.textFile(‘‘type.txt’’)
  .filter{case(s,o)=>o.equals(‘‘Museum’’)}
  .map{case(s,o)=>s}
  .keyBy{case(s)=>s}
SPARQL Translation Process
SPARQL→Scala

```scala
tp1 = sc.textFile('type.txt')
  .filter{case (s, o) => o.equals('Museum')}
  .map{case (s, o) => s}
  .keyBy{case (s) => s}

tp2 = sc.textFile('type.txt')
  .filter{case (g, o) => o.equals('Painter')}
  .map{(g, o) => g}
  .keyBy{case (g) => g}

tp3 = sc.textFile('shows.txt')
  .keyBy{case (s, g) => (s, g)}

bgp = tp1.cartesian(tp2).values
  .keyBy{case (s, g) => (s, g)}
  .join(tp3).value
```
SPARQL Translation Process
SPARQL→Scala

?s type Museum .
?g type Painter .
?s shows ?g

tp1=sc.textFile('type.txt').filter{case(s,o)=>o.equals('Museum')}.map{case(s,o)=>s}.keyBy{case(s)=>s}
tp2=sc.textFile('type.txt').filter{case(g,o)=>o.equals('Painter')}.map{(g,o)=>g}.keyBy{case(g)=>g}
tp3=sc.textFile('shows.txt').keyBy{case(s,g)=>(s,g)}
bgp=tp1.cartesian(tp2).values.keyBy{case(s,g)=>(s,g)}.join(tp3).value
SPARQL Translation Process

SPARQL→Scala

?s type Museum .
?g type Painter .
?s shows ?g

```
val tp1 = sc.textFile("type.txt")
  .filter{case(s,o)=&gt;o.equals("Museum"))
  .map{case(s,o)=&gt;s}
  .keyBy{case(s)=&gt;s}

val tp2 = sc.textFile("type.txt")
  .filter{case(g,o)=&gt;o.equals("Painter"))
  .map{(g,o)=&gt;g}
  .keyBy{case(g)=&gt;g}

val tp3 = sc.textFile("shows.txt")
  .keyBy{case(s,g)=&gt;(s,g)}

val bgp = tp1.cartesian(tp2).values
  .keyBy{case(s,g)=&gt;(s,g)}
  .join(tp3).value
```
To minimize size of intermediate results, we try:

1. Avoiding cartesian product
2. Exploiting statistics on data
Join Order
SPARQL→Scala

To minimize size of intermediate results, we try:

1. Avoiding cartesian product
2. Exploiting statistics on data

Selectivity

- Selectivity of an element located at pos is: either its occurrence number at pos if it is a constant or the total number of triples if it is a variable.
- Selectivity of a TP is the min of its element selectivities.

We just sort the TPs of a BGP in ascending order of their selectivities.
Join Order

SPARQL → Scala

Initial BGP:

?s type Museum .
?g type Painter .
?s shows ?g
Join Order
SPARQL → Scala

**Initial BGP:**
- `?s type Museum .`
- `?g type Painter .`
- `?s shows ?g`

**New BGP:**
- `?s shows ?g`
- `?s type Museum .`
- `?g type Painter`
Join Order

SPARQL→Scala

**Initial BGP:**
- ?s type Museum .
- ?g type Painter .
- ?s shows ?g

**New BGP:**
- ?s shows ?g
- ?s type Museum .
- ?g type Painter

**Associated Scala code:**
```
Initial BGP:
?s type Museum .
?g type Painter .
?s shows ?g

New BGP:
?g type Painter
?

tp1=sc.textFile('"shows.txt"').keyBy{
    case(s,g)=>s
}

tp2=sc.textFile('"type.txt"').filter{
    case(s,o)=>
        o.equals("Museum")
}.map{
    case(s,o)=>
        s
}.keyBy{
    case(s)=>s
}

tp3=sc.textFile('"type.txt"').filter{
    case(s,o)=>
        o.equals("Painter")
}.map{
    case(g,o)=>
        g
}.keyBy{
    case(g)=>g
}

bgp=tp1.join(tp2).values
    .keyBy{case(s,g)=>(g)}
    .join(tp3).value
```
Direct SPARQL Evaluation
Direct SPARQL Evaluation

SDE (SPARQLGX as Direct Evaluator)

- Directly considering the initial RDF dataset
- Designed to evaluate on single query
Direct SPARQL Evaluation

SDE (SPARQLGX as Direct Evaluator)
- Directly considering the initial RDF dataset
- Designed to evaluate on single query

RDFHive
- Based on Apache Hive (relational solution on the HDFS)
- Translation of queries into Hive-QL
- Offers the possibility of merging relational and RDF datasets
Direct SPARQL Evaluation

Figure: Tradeoff between preprocessing and query evaluation times (seconds) linear WatDiv.
Section 6

Conclusion & Perspectives
Conclusion

Summary of Contributions

1. Update comparative Cudré et al. survey

Submitted
Conclusion

Summary of Contributions

1. Update comparative Cudrè et al. survey  
   Submitted

2. Provide a new reading grid (new set of metrics)  
   Submitted
## Conclusion

### Summary of Contributions

1. Update comparative Cudré *et al.* survey
2. Provide a new reading grid (new set of metrics)
3. Develop several distributed SPARQL evaluators:

### Reusability

Openly available under the CeCILL license from:

<https://github.com/tyrex-team>
Conclusion

Summary of Contributions

1. Update comparative Cudré et al. survey
   - Submitted

2. Provide a new reading grid (new set of metrics)
   - Submitted

3. Develop several distributed SPARQL evaluators:
   - SPARQLGX
     - ISWC 2016
   - SDE
     - ISWC 2016
   - RDFHive

Reusability

Openly available under the CeCILL license from:
<https://github.com/tyrex-team>
Conclusion
Conclusion

- Velocity
- Immediacy
- Parsimony
- Dynamicity
- Resiliency

Graph showing performance metrics for different frameworks:
- SPARQLGX
- SDE
- RDFHive
- 4store
- S2RDF
- RYA
- PigSPARQL
- CumulusRDF
- CouchBaseRDF
- CliqueSquare

Frameworks:
- 4store
- CliqueSquare
- CumulusRDF
- CouchBaseRDF
- PigSPARQL
- RYA
- S2RDF
- SDE
- SPARQLGX

Experiments:
- WatDiv1k
- Lúbm1k
I – Perspectives: SPARQL Benchmarking

Uniform test-suite for dynamicity  
Designing a benchmark for the SPARQL UPDATE fragment  

Staying up to date  
- Adding new evaluators  
- Considering other test suites  
- Benchmarking on other clusters  

Varying the number of nodes  
- Validating our results on larger clusters  
- New kind of limitation?
II – Perspectives: SPARQL Evaluators

- **Improving our evaluators**
  - Extending the supported SPARQL fragment
  - Improving the storage models

- **Designing criteria-specific evaluators**
  - Implementing a parsimonious and resilient evaluator
  - Developing evaluators in highly dynamic context

- **Storage-adaptative distributed evaluators**
  - Adapting the idea of Aluç et al. [AÖD14] in a distributed context
  - Considering SPARQL query shapes
  - Choosing its storage model dynamically!
III – Perspectives: Integration in ETL systems

Designing SPARQL pipeline
- Using CONSTRUCT to refine existing RDF datasets
- Aggregating several sources into a single one

Creating heterogeneous data pipeline
- We provide a prototype for trip planning
- Development of a dedicated language

Mid-Term

Mid/Long-Term

ISWC 2016
Thanks for your attention! 😊
Appendices
Appendices

- Appendices
  - Hadoop
  - Spark
  - Cluster
Concept
Map Reduce

HDFS

Données

Map

Paire <K,V>

Reduce

Résultats
1. Resource allocation via cluster manager through *master*
2. *Executors* acquisition on the cluster nodes
3. Code transfer from the application to the *executors*
4. Task transfer to the *executors*
Technical Details

Cluster of 10 nodes with 17GB of RAM each

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of Triples</th>
<th>Original File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>WatDiv1k</td>
<td>109 million</td>
<td>15 GB</td>
</tr>
<tr>
<td>Lubm1k</td>
<td>134 million</td>
<td>23 GB</td>
</tr>
<tr>
<td>Lubm10k</td>
<td>1.38 billion</td>
<td>232 GB</td>
</tr>
</tbody>
</table>


W3C SPARQL Working Group et al.
SPARQL 1.1 overview, 2013.
http://www.w3.org/TR/sparql11-overview/.

Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin.
LUBM: A benchmark for OWL knowledge base systems.

Patrick Hayes and Brian McBride.
RDF semantics.
www.w3.org/TR/rdf-concepts/.

Zoi Kaoudi and Ioana Manolescu.
RDF in the clouds: a survey.

Ryan Lee.
Scalability report on triple store applications.

Aimilia Magkanaraki, Grigoris Karvounarakis, Ta Tuan Anh, Vassilis Christophides, and Dimitris Plexousakis.
Ontology storage and querying.

Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo.
DBpedia SPARQL Benchmark – Performance assessment with real queries on real data.

Shi Qiao and Z Meral Özsoyoğlu.
Rbench: Application-specific RDF benchmarking.

Florian Stegmaier, Udo Gröbner, Mario Döller, Harald Kosch, and Gero Baese.
Evaluation of current rdf database solutions.
Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel.
SP²Bench: a SPARQL performance benchmark.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica.
NSDI, 2012.