Scalable Multi-Query Optimization for SPARQL

Wangchao Le\textsuperscript{1} Anastasios Kementsietsidis\textsuperscript{2} Songyun Duan\textsuperscript{2} Feifei Li\textsuperscript{1}

\textsuperscript{1}University of Utah \hspace{1cm} \textsuperscript{2}IBM Research

April 2, 2012
Outline

1 Introduction

2 Preliminary

3 Our approach

4 Experiments

5 Conclusions
We are inundated with a large collection of RDF (Resource Description Framework) data.
We are inundated with a large collection of RDF (Resource Description Framework) data.

- DBpedia, Uniprot, Freebase etc.

Internally ...

```
<rdf:RDF
  xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
  xmlns:dcterms="http://purl.org/dc/terms/"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:dbpedia-ontology="http://dbpedia.org/ontology/"
  xmlns:rdfs2="http://www.w3.org/2000/01/rdf-schema2#"
  xmlns:dbpedia-owl="http://dbpedia.org/ontology/">
  <rdf:Description rdf:about="urn:x-states:New York">
    <dcterms:alternative>NY</dcterms:alternative>
  </rdf:Description>
</rdf:RDF>
```
We are inundated with a large collection of RDF (Resource Description Framework) data.
- DBpedia, Uniprot, Freebase etc

Internally ...

```
<rdf:RDF
    xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
    xmlns:dcterms="http://purl.org/dc/terms/"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:bibo="http://purl.org/ontology/bibo/"
    xmlns:dbpedia-ontology="http://dbpedia.org/ontology/"
    xmlns:dbpedia-owl="http://dbpedia.org/ontology/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:owl="http://www.w3.org/2002/07/owl/"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
    <rdf:Description rdf:about="urn:x-states:New York">
        <dcterms:alternative>NY</dcterms:alternative>
    </rdf:Description>
</rdf:RDF>
```

Triple format:
```
```

subject    predicate    object
We are inundated with a large collection of RDF (Resource Description Framework) data.
- DBpedia, Uniprot, Freebase etc
- A large graph and encode rich semantics

Internally ...

```
<rdf:RDF
   xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
   xmlns:dcterms="http://purl.org/dc/terms/">
   <rdf:Description rdf:about="urn:x-states:New York">
      <dcterms:alternative>NY</dcterms:alternative>
   </rdf:Description>
</rdf:RDF>
```

Triple format:
```
```

subject          predicate          object
We are inundated with a large collection of RDF (Resource Description Framework) data.

- DBpedia, Uniprot, Freebase etc
- A large graph and encode rich semantics

Internally ...

```xml
<rdf:RDF
  xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
  xmlns:dcterms="http://purl.org/dc/terms/">
  <rdf:Description rdf:about="urn:x-states:New York">
    <dcterms:alternative>NY</dcterms:alternative>
  </rdf:Description>
</rdf:RDF>
```

Triple format:

```
```

Query language: SPARQL
We are inundated with a large collection of RDF (Resource Description Framework) data.
- DBpedia, Uniprot, Freebase etc
- A large graph and encode rich semantics

Available engines to manage RDF data?
We are inundated with a large collection of RDF (Resource Description Framework) data.

- DBpedia, Uniprot, Freebase etc
- A large graph and encode rich semantics

Available engines to manage RDF data?

- **RDBMS**: Migrate RDF, e.g., Sesame, JenaSDB etc.
- **Generic RDF stores**: e.g., RDF3X, JenaTDB etc.

---


Introduction
• Observation: queries share common parts
• Multi-query optimization

\[ Q_1, Q_2, Q_3, Q_{n-1}, Q_n \]

SPARQL queries

RDF store
Introduction

- A tempting choice: turn to MQO in relational databases [MQO88][MQO90][MQO00]
  - SPARQL ↔ relational algebra [EPS08][FSR07].
  - Exist quite a few relational solutions for RDF store.

Introduction

- A tempting choice: turn to MQO in relational databases [MQO88][MQO90][MQO00]
  - SPARQL ↔ relational algebra [EPS08][FSR07].
  - Exist quite a few relational solutions for RDF store.

- For SPARQL and RDF, new issues arise in practice.
A tempting choice: turn to MQO in relational databases [MQO88][MQO90][MQO00]
- SPARQL↔relational algebra [EPS08][FSR07].
- Exist quite a few relational solutions for RDF store.

For SPARQL and RDF, new issues arise in practice.
- Convert SPARQL to SQL: not all engines use RDBMS
- Conversion to SQL → a large number of joins
Introduction

- **A tempting choice:** turn to MQO in relational databases
  - [MQO88][MQO90][MQO00]
  - SPARQL↔relational algebra [EPS08][FSR07].
  - Exist quite a few relational solutions for RDF store.

- **For SPARQL and RDF, new issues arise in practice.**
  - Convert SPARQL to SQL: not all engines use RDBMS
  - Conversion to SQL → a large number of joins
  - Store dependent solution
Outline

1 Introduction
2 Preliminary
3 Our approach
4 Experiments
5 Conclusions
We focus on two types of queries
We focus on two types of queries:

**Type 1:** $Q := SELECT RD WHERE GP$

**Type 2:** $Q_{\text{OPT}} := SELECT RD WHERE GP \ (\text{OPTIONAL } GP_{\text{OPT}})^+$

**Problem statement.**

Input: a set $Q$ of Type 1 queries and a data graph $G$

Output: a set of rewritten queries, $Q_{\text{OPT}}$ of Type 1 and Type 2 queries

**Requirements:**

- **Soundness and completeness:** $Q_{\text{OPT}}(G) \equiv Q(G)$

- **Cost:** $T_r(Q) + T_e(Q_{\text{OPT}}) \leq 19/113$
We focus on two types of queries

**Type 1:** $Q := \text{SELECT RD WHERE GP}$

**Type 2:** $Q_{\text{OPT}} := \text{SELECT RD WHERE GP (OPTIONAL GP}_{\text{OPT}})^+$

<table>
<thead>
<tr>
<th>subj</th>
<th>pred</th>
<th>obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>name</td>
<td>&quot;Alice&quot;</td>
</tr>
<tr>
<td>p1</td>
<td>zip</td>
<td>10001</td>
</tr>
<tr>
<td>p1</td>
<td>mbox</td>
<td>alice@home</td>
</tr>
<tr>
<td>p1</td>
<td>mbox</td>
<td>alice@work</td>
</tr>
<tr>
<td>p1</td>
<td>www</td>
<td><a href="http://home/alice">http://home/alice</a></td>
</tr>
<tr>
<td>p2</td>
<td>name</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>p2</td>
<td>zip</td>
<td>10001</td>
</tr>
<tr>
<td>p3</td>
<td>name</td>
<td>&quot;Ella&quot;</td>
</tr>
<tr>
<td>p3</td>
<td>zip</td>
<td>10001</td>
</tr>
<tr>
<td>p3</td>
<td>www</td>
<td><a href="http://work/ella">http://work/ella</a></td>
</tr>
<tr>
<td>p4</td>
<td>name</td>
<td>&quot;Tim&quot;</td>
</tr>
<tr>
<td>p4</td>
<td>zip</td>
<td>&quot;11234&quot;</td>
</tr>
</tbody>
</table>

(a) triple table $D$

SELECT $?name$
WHERE { ?x name $?name, ?x zip 1001,
}

(b) Example query $Q_{\text{OPT}}$

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>&quot;Ella&quot;</td>
</tr>
</tbody>
</table>
We focus on two types of queries

Type 1: \( Q := \text{SELECT RD WHERE GP} \)
Type 2: \( Q_{\text{OPT}} := \text{SELECT RD WHERE GP (OPTIONAL GP}_{\text{OPT}}) \)

(a) triple table \( D \)

<table>
<thead>
<tr>
<th>subj</th>
<th>pred</th>
<th>obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>name</td>
<td>&quot;Alice&quot;</td>
</tr>
<tr>
<td>p1</td>
<td>zip</td>
<td>10001</td>
</tr>
<tr>
<td>p1</td>
<td>mbox</td>
<td>alice@home</td>
</tr>
<tr>
<td>p1</td>
<td>mbox</td>
<td>alice@work</td>
</tr>
<tr>
<td>p1</td>
<td>www</td>
<td><a href="http://home/alice">http://home/alice</a></td>
</tr>
<tr>
<td>p2</td>
<td>name</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>p2</td>
<td>zip</td>
<td>10001</td>
</tr>
<tr>
<td>p3</td>
<td>name</td>
<td>&quot;Ella&quot;</td>
</tr>
<tr>
<td>p3</td>
<td>zip</td>
<td>10001</td>
</tr>
<tr>
<td>p3</td>
<td>www</td>
<td><a href="http://work/ella">http://work/ella</a></td>
</tr>
<tr>
<td>p4</td>
<td>name</td>
<td>&quot;Tim&quot;</td>
</tr>
<tr>
<td>p4</td>
<td>zip</td>
<td>&quot;11234&quot;</td>
</tr>
</tbody>
</table>

(b) Example query \( Q_{\text{OPT}} \)

SELECT ?name, ?mail, ?hpage
WHERE { ?x name ?name, ?x zip 10001, } OPTIONAL {?x mbox ?mail } OPTIONAL {?x www ?hpage }
We focus on two types of queries

**Type 1:** \( Q := \text{SELECT RD WHERE GP} \)

**Type 2:** \( Q_{\text{OPT}} := \text{SELECT RD WHERE GP (OPTIONAL GP}_{\text{OPT}})^+ \)

(a) triple table \( D \)

<table>
<thead>
<tr>
<th>subj</th>
<th>pred</th>
<th>obj</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>name</td>
<td>&quot;Alice&quot;</td>
</tr>
<tr>
<td>p1</td>
<td>zip</td>
<td>10001</td>
</tr>
<tr>
<td>p1</td>
<td>mbox</td>
<td>alice@home</td>
</tr>
<tr>
<td>p1</td>
<td>mbox</td>
<td>alice@work</td>
</tr>
<tr>
<td>p1</td>
<td>www</td>
<td><a href="http://home/alice">http://home/alice</a></td>
</tr>
<tr>
<td>p2</td>
<td>name</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>p2</td>
<td>zip</td>
<td>10001</td>
</tr>
<tr>
<td>p3</td>
<td>name</td>
<td>&quot;Ella&quot;</td>
</tr>
<tr>
<td>p3</td>
<td>zip</td>
<td>10001</td>
</tr>
<tr>
<td>p3</td>
<td>www</td>
<td><a href="http://work/ella">http://work/ella</a></td>
</tr>
<tr>
<td>p4</td>
<td>name</td>
<td>&quot;Tim&quot;</td>
</tr>
<tr>
<td>p4</td>
<td>zip</td>
<td>&quot;11234&quot;</td>
</tr>
</tbody>
</table>

(b) Example query \( Q_{\text{OPT}} \)

```
SELECT ?name, ?mail, ?hpage
WHERE { ?x name ?name, ?x zip 10001,
OPTIONAL {?x mbox ?mail }
OPTIONAL {?x www ?hpage }}
```

(c) Output \( Q_{\text{OPT}}(D) \)

<table>
<thead>
<tr>
<th>name</th>
<th>mail</th>
<th>hpage</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>alice@home</td>
<td><a href="http://home/alice">http://home/alice</a></td>
</tr>
<tr>
<td>&quot;Alice&quot;</td>
<td>alice@work</td>
<td><a href="http://home/alice">http://home/alice</a></td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
<td></td>
<td><a href="http://home/alice">http://home/alice</a></td>
</tr>
<tr>
<td>&quot;Ella&quot;</td>
<td></td>
<td><a href="http://work/ella">http://work/ella</a></td>
</tr>
<tr>
<td>&quot;Tim&quot;</td>
<td></td>
<td><a href="http://work/ella">http://work/ella</a></td>
</tr>
</tbody>
</table>
We focus on two types of queries

**Type 1:** \( Q := \text{SELECT RD WHERE GP} \)

**Type 2:** \( Q_{\text{OPT}} := \text{SELECT RD WHERE GP (OPTIONAL GP}_{\text{OPT}})^+ \)

- Problem statement.
We focus on two types of queries

**Type 1**: $Q := \text{SELECT RD WHERE GP}$

**Type 2**: $Q_{\text{OPT}} := \text{SELECT RD WHERE GP (OPTIONAL GP}_{\text{OPT}})^+$

Problem statement.

- Input: a set $Q$ of **Type 1** queries and a data graph $G$
Preliminary

- We focus on two types of queries
  
  **Type 1:** $Q := \text{SELECT RD WHERE GP}$
  
  **Type 2:** $Q_{\text{OPT}} := \text{SELECT RD WHERE GP (OPTIONAL GP}_{\text{OPT}})^+$

- Problem statement.
  
  - Input: a set $Q$ of **Type 1** queries and a data graph $G$
  - Output: a set of rewritten queries, $Q_{\text{OPT}}$ of **Type 1** and **Type 2** queries
We focus on two types of queries

**Type 1:** \( Q \) := SELECT RD WHERE GP

**Type 2:** \( Q_{OPT} \) := SELECT RD WHERE GP (OPTIONAL GP\(_{OPT}\))

Problem statement.

- Input: a set \( Q \) of **Type 1** queries and a data graph G
- Output: a set of rewritten queries, \( Q_{OPT} \) of **Type 1** and **Type 2** queries
- Requirements:
  - soundness and completeness: \( Q_{OPT}(G) \equiv Q(G) \)
  - cost: \( \frac{T_r(Q) + T_e(Q_{opt})}{T_e(Q)} \leq 1 \)
Our approach

1. Introduction
2. Preliminary
3. Our approach
4. Experiments
5. Conclusions
Motivating example

(a) Query Q₁

(b) Query Q₂
Motivating example

(a) Query $Q_1$

(b) Query $Q_2$

\[\begin{align*}
\text{Q1: } & \text{SELECT * WHERE } \{ ?x_1 P_1 ?z_1 . ?y_1 P_2 ?z_1 . ?y_1 P_3 ?w_1 . ?w_1 P_4 v_1 . \} \\
\end{align*}\]

\[\begin{align*}
\text{Q2: } & \text{SELECT * WHERE } \{ ?x_2 P_1 ?z_2 . ?t_2 P_3 ?z_2 . ?y_2 P_2 . \} \\
\end{align*}\]

\[\begin{align*}
\text{v1 : constant} & \quad \text{v1 : variable} \\
\end{align*}\]
Motivating example

(a) Query Q₁

(b) Query Q₂
Motivating example

SELECT *
WHERE { ?x P1 ?z, ?y P2 ?z, }

(a) Query Q1
(b) Query Q2

(I) Structure only

QOPT
SELECT *
WHERE { ?x P1 ?z, ?y P2 ?z,
   OPTIONAL { ?y P3 ?w, ?w P4 v1 }
}

(a) Query Q1
(b) Query Q2

(I) Structure only Q_{OPT}
Motivating example

(a) Query $Q_1$

(b) Query $Q_2$

```
SELECT *
WHERE {
  ?x P1 ?z, ?y P2 ?z,
  OPTIONAL {?y P3 ?w, ?w P4 v1 }
  OPTIONAL {?t P3 ?x, ?t P5 v1, ?w P4 v1 }
}
```

(I) Structure only $Q_{OPT}$
Motivating example

SELECT *
WHERE { ?x P1 ?z, ?y P2 ?z,
    OPTIONAL { ?y P3 ?w, ?w P4 v1 }
    OPTIONAL { ?t P3 ?x, ?t P5 v1, ?w P4 v1 }
}

OPTIONALs are evaluated on top of the common substructures
(intermediate results cached by engine).
Motivating example

(a) Query $Q_1$  (b) Query $Q_2$

<table>
<thead>
<tr>
<th>pattern p</th>
<th>$\alpha(p)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$?x : P_1 : ?z$</td>
<td>30%</td>
</tr>
<tr>
<td>$?y : P_2 : ?z$</td>
<td>20%</td>
</tr>
<tr>
<td>$?y : P_3 : ?w$</td>
<td>18%</td>
</tr>
<tr>
<td>$?w : P_4 : v_1$</td>
<td>1%</td>
</tr>
<tr>
<td>$?t : P_5 : v_1$</td>
<td>2%</td>
</tr>
</tbody>
</table>

*Max common subquery is not selective

(II) Using cost in optimization
Motivating example

(a) Query $Q_1$

(b) Query $Q_2$

<table>
<thead>
<tr>
<th>Pattern $p$</th>
<th>$\alpha (p)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$?x \ P_1 \ ?z$</td>
<td>30%</td>
</tr>
<tr>
<td>$?y \ P_2 \ ?z$</td>
<td>20%</td>
</tr>
<tr>
<td>$?y \ P_3 \ ?w$</td>
<td>18%</td>
</tr>
<tr>
<td>$?w \ P_4 \ v_1$</td>
<td>1%</td>
</tr>
<tr>
<td>$?t \ P_5 \ v_1$</td>
<td>2%</td>
</tr>
</tbody>
</table>

*Max common subquery is not selective

(II) Using cost in optimization
Motivating example

\[\begin{align*}
\text{(a) Query } Q_1 & \\
\text{(b) Query } Q_2 &
\end{align*}\]

SELECT *
WHERE {
?w P_4 v_1,
OPTIONAL { ?x_1 P_1 ?z_1, ?y_1 P_2 ?z_1, ?y_1 P_3 ?w }
OPTIONAL { ?x_2 P_1 ?z_2, ?y_2 P_2 ?z_2, ?t_2 P_3 ?x_2, ?t_2 P_5 v_1 }
}

(II) Using cost in optimization
Our approach

\[ Q = \{ q_1, q_2, \ldots, q_n \} \]
Our approach

$Q = \{ q_1, q_2, \ldots, q_n \}$  They often do not share one common subquery
Our approach

\[ Q = \{ q_1, q_2, \ldots, q_n \} \]

- Similar queries can be optimized together
Our approach

\[ Q = \{ q_1, q_2, \ldots, q_n \} \]

- Similar queries can be optimized together
  - finding structure similarity is expensive
  - group by predicates
  - distance: Jaccard similarity of predicate sets
Our approach

\[ Q = \{ q_1, q_2, \ldots, q_n \} \]

Partition input queries

Group 1  \rightarrow  Rewriting

Group 2  \rightarrow  Rewriting

\cdots

Group k  \rightarrow  Rewriting

• Similar queries can be optimized together
• Finding structure similarity is expensive
• Group by predicates
• Distance: Jaccard similarity of predicate sets
Our approach

\[ Q = \{ q_1, q_2, \ldots, q_n \} \]

Paritition input queries

- Group 1
  - Rewriting
- Group 2
  - Rewriting
- \ldots
- Group \( k \)
  - Rewriting

\[ \text{• Recursively rewrite a subset of type 1 queries} \]
\[ \text{(hierarchically)} \rightarrow \text{a set of type 2 queries} \]
Our approach

\[ Q = \{ q_1, q_2, \ldots, q_n \} \]

Paritition input queries

Group 1  Group 2  \cdots  Group k

Rewriting  Rewriting  \cdots  Rewriting

- Recursively rewrite a subset of type 1 queries (hierarchically) → a set of type 2 queries
  - finding common edge subgraphs
  - optimizations to avoid bad efficiency
  - **cost**: guard against bad rewritings
  - approx. by the min selectivity in common subquery
Our approach

\[ Q = \{ q_1, q_2, \ldots, q_n \} \]

- Similar queries can be optimized together
- Finding structure similarity is expensive
- Group by predicates
- Distance: Jaccard similarity of predicate sets

Rewriting

- Recursively rewrite a subset of type 1 queries (hierarchically) → a set of type 2 queries
  - Finding common edge subgraphs
  - Optimizations to avoid bad efficiency
  - Cost: guard against bad rewritings
  - Approx. by the min selectivity in common subquery

<table>
<thead>
<tr>
<th>Pattern p</th>
<th>( \alpha(p) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(?x P_1 ?z)</td>
<td>30%</td>
</tr>
<tr>
<td>(?y P_2 ?z)</td>
<td>20%</td>
</tr>
<tr>
<td>(?y P_3 ?w)</td>
<td>18%</td>
</tr>
<tr>
<td>(?w P_4 v_1)</td>
<td>1%</td>
</tr>
<tr>
<td>(?t P_5 v_1)</td>
<td>2%</td>
</tr>
</tbody>
</table>
Our approach

\[ Q = \{ q_1, q_2, \ldots, q_n \} \]

Paritition input queries

- Group 1
- Group 2
- \ldots
- Group \( k \)

- Rewriting
- Rewriting
- \ldots
- Rewriting

Execution (hierarchically) \rightarrow a set of type 2 queries

- Similar queries can be optimized together
- finding structure similarity is expensive
- group by predicates
- distance: Jaccard similarity of predicate sets

- Rewriting
- finding common edge subgraphs
- optimizations to avoid bad efficiency
- cost: guard against bad rewritings
- approx. by the min selectivity in common subquery
Our approach

\[ Q = \{ q_1, q_2, \ldots, q_n \} \]

Paritition input queries

Group 1 \quad \text{Group 2} \quad \cdots \quad \text{Group } k

Rewriting \quad \text{Rewriting} \quad \cdots \quad \text{Rewriting}

Execution

Result distribution

\[ r(q_1) \quad r(q_2) \quad r(q_n) \]
Our approach

- Related issues
Our approach

- Related issues
  - Distributing results, i.e., Type 2 query $\rightarrow$ Type 1 queries

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>c</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>g</td>
<td></td>
</tr>
</tbody>
</table>

RD of a Type 1 query: e.g., X and Z

↑↓
columns from results of the Type 2 rewriting
Our approach

- Related issues
  - Distributing results, *i.e.*, **Type 2 query** → **Type 1 queries**

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td>g</td>
</tr>
</tbody>
</table>

RD of a **Type 1** query: *e.g.*, X and Z

↑↓ columns from results of the **Type 2** rewriting

- Soundness and completeness
Our approach

- Related issues
  - Distributing results, \textit{i.e.}, \textbf{Type 2} query $\rightarrow$ \textbf{Type 1} queries

\begin{tabular}{ccc}
X & Y & Z \\
\hline
a & & \\
b & c & \\
d & e & \\
f & & g
\end{tabular}

RD of a \textbf{Type 1} query: \textit{e.g.}, $X$ and $Z$

\begin{tabular}{cccc}
& & & \\
\hline
& & & \\
& & & \\
& & & \\
\end{tabular}

- Soundness and completeness
- Extensibility of the solution: more general queries
  - handle variable predicates
  - OPTIONAL queries
Experiments

- Implementation highlights
  - C++
  - 64-bit Linux, 2GHz Xeon(R) CPU, 4GB memory
Experiments

- **Implementation highlights**
  - C++
  - 64-bit Linux, 2GHz Xeon(R) CPU, 4GB memory

- **Dataset**
  - Extend LUBM benchmark generator: randomness in structure, variances of sel.
Experiments

- Implementation highlights
  - C++
  - 64-bit Linux, 2GHz Xeon(R) CPU, 4GB memory

- Dataset
  - Extend LUBM benchmark generator:
    randomness in structure, variances of sel.

- RDF stores: Jena TDB 0.85 etc
Experiments

- Implementation highlights
  - C++
  - 64-bit Linux, 2GHz Xeon(R) CPU, 4GB memory
- Dataset
  - Extend LUBM benchmark generator: randomness in structure, variances of sel.
- RDF stores: Jena TDB 0.85 etc
- Queries
  - Ensure randomness in structure, e.g., star, chain and circle
Experiments

- Implementation highlights
  - C++
  - 64-bit Linux, 2GHz Xeon(R) CPU, 4GB memory

- Dataset
  - Extend LUBM benchmark generator: randomness in structure, variances of sel.
  - RDF stores: Jena TDB 0.85 etc

- Queries

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Symbol</th>
<th>Default</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset size</td>
<td>D</td>
<td>4M</td>
<td>3M to 9M</td>
</tr>
<tr>
<td>Number of queries</td>
<td></td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>Size of query (num of triple patterns)</td>
<td></td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>Number of seed queries</td>
<td>κ</td>
<td>6</td>
<td>5 to 10</td>
</tr>
<tr>
<td>Size of seed queries</td>
<td>(q_{cmn})</td>
<td>(\sim)</td>
<td>(</td>
</tr>
<tr>
<td>Max selectivity of patterns in Q</td>
<td>(\alpha_{max}(Q))</td>
<td>random</td>
<td>0.1% to 4%</td>
</tr>
<tr>
<td>Min selectivity of patterns in Q</td>
<td>(\alpha_{min}(Q))</td>
<td>1%</td>
<td>0.1% to 4%</td>
</tr>
</tbody>
</table>
Experiments

- Implementation highlights
  - C++
  - 64-bit Linux, 2GHz Xeon(R) CPU, 4GB memory
- Dataset
  - Extend LUBM benchmark generator:
    - randomness in structure, variances of sel.
- RDF stores: Jena TDB 0.85 etc
- Queries

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Symbol</th>
<th>Default</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset size</td>
<td>$D$</td>
<td>4M</td>
<td>3M to 9M</td>
</tr>
<tr>
<td>Number of queries</td>
<td>$</td>
<td>Q</td>
<td>$</td>
</tr>
<tr>
<td>Size of query (num of triple patterns)</td>
<td>$</td>
<td>Q</td>
<td>$</td>
</tr>
<tr>
<td>Number of seed queries</td>
<td>$\kappa$</td>
<td>6</td>
<td>5 to 10</td>
</tr>
<tr>
<td>Size of seed queries</td>
<td>$</td>
<td>q_{cmn}</td>
<td>$</td>
</tr>
<tr>
<td>Max selectivity of patterns in Q</td>
<td>$\alpha_{max}(Q)$</td>
<td>random</td>
<td>0.1% to 4%</td>
</tr>
<tr>
<td>Min selectivity of patterns in Q</td>
<td>$\alpha_{min}(Q)$</td>
<td>1%</td>
<td>0.1% to 4%</td>
</tr>
</tbody>
</table>

- Rewriting w/ structure: MQO-S; rewriting w/ structure and cost: MQO
Experiments

- **Time on rewriting**
  - **MQO-S-C**: structure based rewriting
  - **MQO-C**: rewriting integrating with cost
Experiments

- **Time on rewriting**
  - **MQO-S-C**: structure based rewriting
  - **MQO-C**: rewriting integrating with cost

*Costly/bad rewritings are rejected → more rounds of comparisons.*
Experiments

- Time on distributing results
  MQO-S-P: parsing results from MQO-S
  MQO-P: parsing results with MQO
Experiments

- Time on distributing results
  - MQO-S-P: parsing results from MQO-S
  - MQO-P: parsing results with MQO

*Non-selective common subqueries increase the set of results.*
Experiments

- Time on distributing results
  - MQO-S-P: parsing results from MQO-S
  - MQO-P: parsing results with MQO

*Non-selective common subqueries increase the set of results.

*Both rewriting and parsing are efficiently doable.
Experiments

- Varying num of queries in a batch
  - No-MQO: no optimization
  - MQO-S: optimization based on structural rewriting
  - MQO: integrating cost
Experiments

- Varying num of queries in a batch
  - No-MQO: no optimization
  - MQO-S: optimization based on structural rewriting
  - MQO: integrating cost

*Both reduce the num of queries to be executed*
Experiments

- Varying num of queries in a batch
  - No-MQO: no optimization
  - MQO-S: optimization based on structural rewriting
  - MQO: integrating cost
Experiments

- Varying min. selectivity in seed queries
  - No-MQO: no optimization
  - MQO-S: optimization based on structural rewriting
  - MQO: integrating cost
### Experiments

- Varying min. selectivity in seed queries
  - No-MQO: no optimization
  - MQO-S: optimization based on structural rewriting
  - MQO: integrating cost

<table>
<thead>
<tr>
<th>$\alpha_{min}(q_{cmn})$ (%)</th>
<th>No-MQO</th>
<th>MQO-S</th>
<th>MQO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>0.5</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

*MQO: reject more bad rewritings; MQO-S: not sensitive*
Experiments

- Varying min. selectivity in seed queries
  - No-MQO: no optimization
  - MQO-S: optimization based on structural rewriting
  - MQO: integrating cost

![Graph showing time (seconds) vs. α_{min}(q_{cmn}) (%) for No-MQO, MQO-S, and MQO]
Experiments

- Varying seed size
  - **MQO-S**: optimization based on structural rewriting
  - **MQO**: integrating cost
  - percentage\(= \frac{T_e(\text{common subquery})}{T_e(Q_{opt})} \times 100\%\)
Experiments

- Varying seed size
  - **MQO-S**: optimization based on structural rewriting
  - **MQO**: integrating cost
  
  \[
  \text{percentage} = \frac{T_e(\text{common subquery})}{T_e(Q_{opt})} \times 100\%
  \]

*MQO-S: up to 25% time on optional*
Conclusions

- In dealing RDF data on the Web, store independency is important.
- Combining SPARQL language and graph algorithms can achieve MQO, i.e., by rewriting queries.
- Cost must be taken in consideration during rewriting.
Thank You

Q and A
Our approach

- Partition input queries
Our approach

- Partition input queries
  - Object: similar queries can be optimized together in rewriting
Our approach

- Partition input queries
  - Object: similar queries can be optimized together in rewriting
Our approach

- **Partition input queries**
  - **Object:** similar queries can be optimized together in rewriting
Our approach

- Partition input queries
  - Object: similar queries can be optimized together in rewriting

Distance: Jaccard similarity on predicates
Our approach

- Partition input queries
  - Object: similar queries can be optimized together in rewriting

Distance: Jaccard similarity on predicates
- Represent each query as a set of predicates.
- Measure the similarity of a pair of queries by set similarity
Our approach

- Partition input queries
  - Object: similar queries can be optimized together in rewriting

Distance: Jaccard similarity on predicates
- Represent each query as a set of predicates.
- Measure the similarity of a pair of queries by set similarity
- Grouping: k-means
Our approach

- Partition input queries
  - Object: similar queries can be optimized together in rewriting

Distance: Jaccard similarity on predicates
- Represent each query as a set of predicates.
- Measure the similarity of a pair of queries by set similarity
- Grouping: k-means
Hierarchical rewriting and clustering (inside a group)
Our approach

- Hierarchical rewriting and clustering (inside a group)
  Rewrite pairs of queries bottom up

\[ q_{ab} q_{cd} \]

\[ q_{abcd} \]

\[ qa \quad qb \quad qc \quadqd \quad \ldots \quad \ldots \quad \ldots \quad qr \quad qs \]
Our approach

- Hierarchical rewriting and clustering (inside a group)

Rewrite pairs of queries bottom up

Rewriting $\rightarrow$ finding maximal common triple patterns

In the language of graph

Hierarchical rewriting and clustering (inside a group)

Rewrite pairs of queries bottom up
Pair up queries with max Jaccard similarity
Hierarchical rewriting and clustering (inside a group)

Rewrite pairs of queries bottom up
Pair up queries with max Jaccard similarity
Hierarchical rewriting and clustering (inside a group)

Rewrite pairs of queries bottom up
Pair up queries with max Jaccard similarity

Rewriting $\rightarrow$ finding maximal common triple patterns
In the language of graph ...
Our approach

- Hierarchical rewriting and clustering (inside a group)
  - Rewrite pairs of queries bottom up
  - Pair up queries with max Jaccard similarity

- Rewriting → finding maximal common triple patterns
- In the language of graph . . .

Our approach

- Hierarchical rewriting and clustering (inside a group)
  - Rewrite pairs of queries bottom up
  - Pair up queries with max Jaccard similarity

Rewriting → finding maximal common triple patterns

In the language of graph . . .

  - maximal common connected edge subgraphs
Our approach

- Hierarchical rewriting and clustering (inside a group)
  
  Rewrite pairs of queries bottom up
  Pair up queries with max Jaccard similarity

- Rewriting → finding maximal common triple patterns

- In the language of graph . . . 


  - maximal common connected edge subgraphs
    → maximal common connected *induced* sugraphs in linegraphs
Our approach

- Hierarchical rewriting and clustering (inside a group)
  Rewrite pairs of queries bottom up
  Pair up queries with max Jaccard similarity

- Rewriting → finding maximal common triple patterns

- In the language of graph . . .


  - maximal common connected edge subgraphs
    → maximal common connected *induced* subgraphs in linegraphs
    → maximal cliques in the product graph
Our approach

(a) Query $Q_1$  
(b) Query $Q_2$  
(c) Query $Q_3$  
(d) Query $Q_4$
Our approach

- Linegraph: invert vertices and edges
Our approach

- **Linegraph**: invert vertices and edges
- sub—sub: $\ell_0$, sub—obj: $\ell_1$, obj—sub: $\ell_2$, obj—obj: $\ell_3$
Our approach

• Linegraph: invert vertices and edges
  • sub–sub: $\ell_0$, sub–obj: $\ell_1$, obj–sub: $\ell_2$, obj–obj: $\ell_3$

• product graph: simultaneous walk
Our approach

- Linegraph: invert vertices and edges
- sub–sub: $\ell_0$, sub–obj: $\ell_1$, obj–sub: $\ell_2$, obj–obj: $\ell_3$

- product graph: simultaneous walk
Our approach

- Linegraph: invert vertices and edges
  - sub–sub: $\ell_0$, sub–obj: $\ell_1$, obj–sub: $\ell_2$, obj–obj: $\ell_3$

- product graph: simultaneous walk

The triangle (clique) highlights the common subgraph composed by $\blacksquare \times \blacklozenge$

---

The triangle (clique) highlights the common subgraph composed by $\blacksquare \times \blacklozenge$

---

The triangle (clique) highlights the common subgraph composed by $\blacksquare \times \blacklozenge$

---

The triangle (clique) highlights the common subgraph composed by $\blacksquare \times \blacklozenge$
Our approach

- Linegraph: invert vertices and edges
- sub–sub: $\ell_0$, sub–obj: $\ell_1$, obj–sub: $\ell_2$, obj–obj: $\ell_3$

- product graph: simultaneous walk
- blowup in size, esp. $> 2$ queries affect clique detection
Our approach

- Linegraph: invert vertices and edges
- sub–sub: $\ell_0$, sub–obj: $\ell_1$, obj–sub: $\ell_2$, obj–obj: $\ell_3$

- product graph: simultaneous walk
- blowup in size, esp. > 2 queries affect clique detection
- optimize the product graph

The triangle (clique) highlights the common subgraph composed by graph query pattern 1 and graph query pattern 2.
Our approach

- Linegraph: invert vertices and edges
  - sub—sub: $\ell_0$, sub—obj: $\ell_1$, obj—sub: $\ell_2$, obj—obj: $\ell_3$

- Product graph: simultaneous walk
  - Blowup in size, esp. > 2 queries affect clique detection
  - Optimize the product graph

- Prune non-common predicates
- Check the constants
Our approach

- Linegraph: invert vertices and edges
  - sub—sub: $\ell_0$, sub—obj: $\ell_1$, obj—sub: $\ell_2$, obj—obj: $\ell_3$

- Product graph: simultaneous walk
  - Blowup in size, esp. $>2$ queries
    - Affect clique detection
  - Optimize the product graph

- Prune non-common predicates
- Check the constants
- Prune vertices with non-common neighborhoods
Our approach

- **Linegraph**: invert vertices and edges
- **sub–sub:** $\ell_0$, **sub–obj:** $\ell_1$, **obj–sub:** $\ell_2$, **obj–obj:** $\ell_3$

- **product graph**: simultaneous walk
- **blowup in size**, esp. $>2$ queries affect clique detection
- **optimize the product graph**

- prune non-common predicates
- check the constants
- prune vertices with non-common neighborhoods
Our approach

- Find maximal cliques in the product graph [CLQ02][CLQ03]

Our approach

- Find maximal cliques in the product graph [CLQ02][CLQ03]

  
Our approach

Find maximal cliques in the product graph [CLQ02][CLQ03]


Integrate cost into rewriting
Our approach

- Find maximal cliques in the product graph [CLQ02][CLQ03]
  
  

- Integrate cost into rewriting
  
  - Structure: maximize size of the common subquery in a rewriting
  - Evaluation on cost: guard against bad rewritings
Our approach

- Find maximal cliques in the product graph [CLQ02][CLQ03]
  
  

- Integrate cost into rewriting
  
  - **Structure**: maximize size of the common subquery in a rewriting
  - Evaluation on cost: guard against bad rewritings
  - Measure: min selectivity in the common subquery for approximation
Our approach

- Find maximal cliques in the product graph [CLQ02][CLQ03]
  
  

- Integrate cost into rewriting
  
  - **Structure**: maximize size of the common subquery in a rewriting
  - Evaluation on cost: guard against bad rewritings
  - Measure: min selectivity in the common subquery for approximation
  - **Cost**: discard bad rewritings, keep good ones in hierarchical rewriting
Our approach

- Find maximal cliques in the product graph [CLQ02][CLQ03]
  
  

- Integrate cost into rewriting
  
  - **Structure**: maximize size of the common subquery in a rewriting
  - **Evaluation on cost**: guard against bad rewritings
  - **Measure**: min selectivity in the common subquery for approximation
  - **Cost**: discard bad rewritings, keep good ones in hierarchical rewriting
Our approach

- Find maximal cliques in the product graph [CLQ02][CLQ03]
  
  

- Integrate cost into rewriting
  
  - **Structure**: maximize size of the common subquery in a rewriting
  - **Evaluation on cost**: guard against bad rewritings
  - **Measure**: min selectivity in the common subquery for approximation
  - **Cost**: discard bad rewritings, keep good ones in hierarchical rewriting

![Diagram showing the structure of rewritings and the condition to stop when selectivity drops.](image-url)
Our approach

- Related issues
Our approach

- Related issues
  - Distributing results, i.e., Type 2 query $\rightarrow$ Type 1 queries

<table>
<thead>
<tr>
<th>name</th>
<th>mail</th>
<th>hpage</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>alice@home</td>
<td><a href="http://home/alice">http://home/alice</a></td>
</tr>
<tr>
<td>&quot;Alice&quot;</td>
<td>alice@work</td>
<td><a href="http://home/alice">http://home/alice</a></td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
<td></td>
<td><a href="http://home/alice">http://home/alice</a></td>
</tr>
<tr>
<td>&quot;Ella&quot;</td>
<td></td>
<td><a href="http://work/ella">http://work/ella</a></td>
</tr>
</tbody>
</table>

RD of a Type 1 query

↑↓
columns from results of the Type 2 rewriting
Our approach

- Related issues
  - Distributing results, i.e., Type 2 query $\rightarrow$ Type 1 queries
    
    | name   | mail       | hpage       |
    |--------|------------|-------------|
    | "Alice" | alice@home | http://home/alice |
    | "Alice" | alice@work | http://home/alice |
    | "Bob"   |            |             |
    | "Ella"  |            | http://work/ella |

    RD of a Type 1 query
    $\uparrow \downarrow$
    columns from results of the Type 2 rewriting

- Soundness and completeness
Our approach

- Related issues
  - Distributing results, *i.e.*, **Type 2 query** $\rightarrow$ **Type 1 queries**

<table>
<thead>
<tr>
<th>name</th>
<th>mail</th>
<th>hpage</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>alice@home</td>
<td><a href="http://home/alice">http://home/alice</a></td>
</tr>
<tr>
<td>&quot;Alice&quot;</td>
<td>alice@work</td>
<td><a href="http://home/alice">http://home/alice</a></td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;Ella&quot;</td>
<td></td>
<td><a href="http://work/ella">http://work/ella</a></td>
</tr>
</tbody>
</table>

RD of a **Type 1 query**

↑↓

columns from results of the **Type 2 rewriting**

- Soundness and completeness
- Extensibility of the solution: more general queries
  - handle variable predicates
  - nested OPTIONALs