Generating Efficient Execution Plans for Vertically Partitioned XML Databases

Patrick Kling, M. Tamer Özsu, and Khuzaima Daudjee

University of Waterloo
David R. Cheriton School of Computer Science

VLDB 2011
The Problem

- Centralized query evaluation techniques for XML well understood
- These techniques do not scale to large collection sizes and heavy workloads
- Goal: use distribution to improve scalability
- Focus on end-to-end cost of query evaluation
Distributed XML Query Evaluation: Two Scenarios

- Integrating multiple data sources
  - Fragmentation is determined by existing data sources
  - Need flexible fragmentation model to express this

- Distribution for performance
  - Choose fragmentation to suit workload
  - Can use more constrained fragmentation model
  - Fragmentation specification allows for distributed query optimization
Distributed XML Query Evaluation: Two Scenarios

- Integrating multiple data sources
  - Fragmentation is determined by existing data sources
  - Need flexible fragmentation model to express this
- Distribution for performance
  - Choose fragmentation to suit workload
  - Can use more constrained fragmentation model
  - Fragmentation specification allows for distributed query optimization
Outline

1. Fragmenting XML Collections

2. Querying Distributed XML Collections
   - Query Model
   - Distributed Query Evaluation
   - Improving Performance

3. Performance Evaluation

4. Conclusion
Fragmenting XML Collections

- Ad-hoc fragmentation
- Structure-based fragmentation
Ad-hoc fragmentation

- Cut arbitrary edges in document tree
- Highly flexible (good for data integration)
- No explicit fragmentation specification
- Limited potential for exploiting fragmentation characteristics for query optimization
- Not a suitable choice for this work
Structure-based Fragmentation

- Fragmentation according to characteristics of data or schema
- Yields a fragmentation specification that can be exploited for query optimization
- Better choice when distributing for performance
Our Fragmentation Model

- Focus on simplicity and precise fragmentation specification
- Focus on partitioning collection (replication is orthogonal)
- Follow semantics of relational fragmentation techniques
  - Horizontal fragmentation (based on predicates/selection)
  - Vertical fragmentation (based on partitioning of schema/projection)
  - Hybrid fragmentation (combination of horizontal and vertical steps)
Our Fragmentation Model

- Focus on simplicity and precise fragmentation specification
- Focus on partitioning collection (replication is orthogonal)
- Follow semantics of relational fragmentation techniques
  - Horizontal fragmentation (based on predicates/selection)
  - Vertical fragmentation (based on partitioning of schema/projection)
  - Hybrid fragmentation (combination of horizontal and vertical steps)
Vertical Fragmentation

\[ \text{author}_2 \]

\[ P^{1 \rightarrow 2}_{13} \quad P^{1 \rightarrow 3}_{14} \]

\[ f_1^V \]

\[ \text{name}_2 \]

\[ \text{first}_2 \quad \text{last}_2 \]

\[ \text{Jane} \quad \text{Dean} \]

\[ f_2^V \]

\[ RP^{1 \rightarrow 2}_{13} \]

\[ f_3^V \]

\[ \text{pubs}_2 \]
Vertical fragmentation is specified by a *fragmentation schema*.
Outline

1. Fragmenting XML Collections

2. Querying Distributed XML Collections
   - Query Model
   - Distributed Query Evaluation
   - Improving Performance

3. Performance Evaluation

4. Conclusion
Query model

XQ, subset of XPath

- Nested paths with child and descendant steps
- Explicit node tests and wild cards
- Value constraints (numeric or textual)
- \( Q := \sigma | * | Q//Q | Q/Q | Q[q] \)
  \( q := Q | . | = | \neq | str | . | = | \neq | \leq | < | \geq | > | num \)
Query Example

“Find all references in publications written by authors whose first name is ‘William’ and whose last name is ‘Shakespeare’.”
“Find all references in publications written by authors whose first name is ‘William’ and whose last name is ‘Shakespeare’ ”

/author[./name[./first = “William” and ./last = “Shakespeare”]]/reference
Query Example

“Find all references in publications written by authors whose first name is ‘William’ and whose last name is ‘Shakespeare’ ”

/author[./name[./first = “William” and ./last = “Shakespeare”]]/reference

• Node tests
Query Example

“Find all references in publications written by authors whose first name is ‘William’ and whose last name is ‘Shakespeare’ ”

/author[./name[./first = “William” and ./last = “Shakespeare”]]/reference

- Node tests
- Value constraints
Query Example

“Find all references in publications written by authors whose first name is ‘William’ and whose last name is ‘Shakespeare’”

/author[./name[./first = “William” and ./last = “Shakespeare”]]///reference

- Node tests
- Value constraints
- Structural constraints
Tree Patterns

- **author**
- **name**
- **reference**
  - first = 'William'
  - last = 'Shakespeare'
Tree Patterns

- Pattern nodes with **node tests** and value constraints

```
author
  - name
    - first .='William'
    - last .='Shakespeare'
  - reference
```
Tree Patterns

- Pattern nodes with node tests and value constraints
Tree Patterns

- Pattern nodes with node tests and value constraints
- Edges annotated with XPath axes
Tree Patterns

- Pattern nodes with node tests and value constraints
- Edges annotated with XPath axes
- Extraction point nodes
Evaluating Tree Pattern Queries
Evaluating Tree Pattern Queries

\[
\text{author} \\
\text{name} \\
\quad \text{first} \quad \text{last} \\
\quad .='William' \\
\quad .='Shakespeare' \\
\text{reference} \\
\text{a_1} \\
\text{author_4} \\
\text{name_4} \\
\quad \text{first_4} \\
\quad \text{last_4} \\
\quad \text{William} \\
\quad \text{Shakespeare} \\
\text{pubs_4} \\
\text{book_4} \\
\quad \text{chapter_4} \\
\quad \text{chapter_5} \\
\text{reference_4}
\]
Evaluating Tree Pattern Queries
Evaluating Tree Pattern Queries
Evaluating Tree Pattern Queries

```
_ author_
  /
//
_ name_ a1e _ reference_
  /   /
//   /
first  last
  .= 'William'
  .= 'Shakespeare'

author4
  /
//
name4 pubs4
  /
//
first4  last4  book4
  William Shakespeare

chapter4  chapter5
  reference4
```
Evaluating Tree Pattern Queries
Evaluating Tree Pattern Queries

16
Evaluating Tree Pattern Queries
Evaluating Tree Pattern Queries

[\[a_1^e = \text{reference}_4\]]
Evaluating Tree Pattern Queries

- Various centralized approaches exist
  - Navigating document trees
  - Structural joins
- We leverage these for distributed query evaluation
Querying Vertically Distributed XML Collections

- **Input**
  - Fragmentation-unaware tree pattern query
  - Fragmentation schema

- **Tasks**
  - Annotate tree pattern nodes with corresponding fragments
  - Decompose tree pattern into sub-patterns for individual fragments
  - Convert sub-patterns to local plans using existing techniques (each site is free to choose local strategy)
  - Generate distributed execution plan that specifies how results are combined
Querying Vertically Distributed XML Collections

- Annotate tree pattern nodes
- Decompose tree pattern
- Convert sub-patterns into local plans
- Generate distributed execution plan

```
[9x251]Querying Vertically Distributed XML Collections

• Annotate tree pattern nodes
• Decompose tree pattern
• Convert sub-patterns into local plans
• Generate distributed execution plan

author

name

first .='William'

last .='Shakespeare'

reference
```
Querying Vertically Distributed XML Collections

- Annotate tree pattern nodes
- Decompose tree pattern
- Convert sub-patterns into local plans
- Generate distributed execution plan
Querying Vertically Distributed XML Collections

- Annotate tree pattern nodes
- Decompose tree pattern
- Convert sub-patterns into local plans
- Generate distributed execution plan

![Diagram of XML tree structure]

```
  author
   /
  /   /
name  reference
   /
  /   /
first  last
   .= 'William'
   .= 'Shakespeare'
```
Querying Vertically Distributed XML Collections

- Annotate tree pattern nodes
- Decompose tree pattern
- Convert sub-patterns into local plans
- Generate distributed execution plan
Querying Vertically Distributed XML Collections

- Annotate tree pattern nodes
- Decompose tree pattern
- Convert sub-patterns into local plans
- Generate distributed execution plan
Querying Vertically Distributed XML Collections

- Annotate tree pattern nodes
- Decompose tree pattern
- Convert sub-patterns into local plans
- Generate distributed execution plan

\[ \begin{align*}
\land \text{id}(a_3^p) &= \text{id}(a_3^{rp}) \\
\land \text{id}(a_2^p) &= \text{id}(a_2^{rp}) \\
\land \text{id}(a_4^p) &= \text{id}(a_4^{rp}) \\
\end{align*} \]

\[ \begin{align*}
\& \text{id}(a_3^p) = \text{id}(a_3^{rp}) \\
\& \text{id}(a_2^p) = \text{id}(a_2^{rp}) \\
\& \text{id}(a_4^p) = \text{id}(a_4^{rp}) \\
\end{align*} \]
Querying Vertically Distributed XML Collections

- Annotate tree pattern nodes
- Decompose tree pattern
- Convert sub-patterns into local plans
- Generate distributed execution plan

\[
\begin{align*}
\text{id}(a_p^3) &= \text{id}(a_{rp}^3) \\
\text{id}(a_p^2) &= \text{id}(a_{rp}^2) \\
\text{id}(a_p^4) &= \text{id}(a_{rp}^4)
\end{align*}
\]
Querying Vertically Distributed XML Collections

- Annotate tree pattern nodes
- Decompose tree pattern
- Convert sub-patterns into local plans
- Generate distributed execution plan
Querying Vertically Distributed XML Collections

- Annotate tree pattern nodes
- Decompose tree pattern
- Convert sub-patterns into local plans
- Generate distributed execution plan

\[
\begin{align*}
\text{id}(a_{3p}) &= \text{id}(a_{3rp}) \\
\text{id}(a_{2p}) &= \text{id}(a_{2p}) \\
\text{id}(a_{4p}) &= \text{id}(a_{4p}) \\
\text{id}(a_{4p}) &= \text{id}(a_{4p}) \\
\text{id}(a_{3p}) &= \text{id}(a_{3rp}) \\
\end{align*}
\]
Improving Distributed Execution Plans

- Pruning irrelevant fragments
- Join order
- Push cross-fragment joins into local plans
Improving Distributed Execution Plans

- Pruning irrelevant fragments
- Join order
- Push cross-fragment joins into local plans
Pushing Cross-Fragment Joins

Large fraction of local results are discarded by cross-fragment join
Pushing Cross-Fragment Joins

Large fraction of local results are discarded by cross-fragment join
Pushing Cross-Fragment Joins

Large fraction of local results are discarded by cross-fragment join

- Idea: only access relevant sub-trees in fragment
- Avoid computing irrelevant local results
- Use pipelining to push cross-fragment join into local plan
A Local Query Plan

\[ \pi_{a_2^{rp}} \]
\[ \times a_1/a_3 \]
\[ \times a_1/a_2 \]
\[ \times a_2^{rp}/a_1 \]
\[ \sigma_{a_2='William'} \]
\[ \sigma_{a_3='Shakespeare'} \]
\[ \text{scan}_{a_3:\text{last}} \]
\[ \text{scan}_{a_2:\text{first}} \]
\[ \text{scan}_{a_1:\text{name}} \]
\[ \text{scan}_{a_2^{rp}:RP_{1\rightarrow 2}} \]
\[ p_1^2(f_2^V) \]
A Local Query Plan

- Plan scans root proxy nodes in fragment
- Idea: filter these root proxy nodes before evaluating remainder of plan
- Works for navigating plans and plans based on structural joins (shown here)
A Local Query Plan

- Plan scans root proxy nodes in fragment
- Idea: filter these root proxy nodes before evaluating remainder of plan
- Works for navigating plans and plans based on structural joins (shown here)
Pushing Cross-Fragment Joins

\[
p_1^1(f_1^V) \quad \text{scan}_{a_2^{rp}:RP_*^{1\rightarrow 2}}
\]

\[
p_2^2(f_2^V) \quad \text{scan}_{a_3^{rp}:RP_*^{1\rightarrow 3}}
\]

\[
p_3^3(f_3^V) \quad \text{scan}_{a_4^{rp}:RP_*^{3\rightarrow 4}}
\]

\[
p_4^4(f_4^V) \quad \text{id}(a_4^p)=\text{id}(a_4^{rp})
\]
Pushing Cross-Fragment Joins

\[ p_1^3 (f_4^\mathcal{V}) \]
\[ \Join \text{id}(a_4^p) = \text{id}(a_4^{rp}) \]
\[ p_1^2 (f_3^\mathcal{V}) \quad \text{scan}_{a_4^{rp}:RP_3^* \rightarrow 4} \]
\[ \Join \text{id}(a_3^p) = \text{id}(a_3^{rp}) \]
\[ p_1^1 (f_2^\mathcal{V}) \quad \text{scan}_{a_3^{rp}:RP_1^* \rightarrow 3} \]
\[ \Join \text{id}(a_2^p) = \text{id}(a_2^{rp}) \]
\[ p_1^1 (f_1^\mathcal{V}) \quad \text{scan}_{a_2^{rp}:RP_1^* \rightarrow 2} \]
Pushing Cross-Fragment Joins

\[ p_1^A(f^V) \]

\[ \Join \text{id}(a_4^p) = \text{id}(a_4^{rp}) \]

\[ p_1^B(f^V) \]

\[ \Join \text{id}(a_3^p) = \text{id}(a_3^{rp}) \]

\[ p_1^C(f^V) \]

\[ \Join \text{id}(a_2^p) = \text{id}(a_2^{rp}) \]

\[ \text{scan}_{a_4^{rp}: RP_3^* \to 4} \]

\[ \text{scan}_{a_3^{rp}: RP_2^* \to 3} \]

\[ \text{scan}_{a_2^{rp}: RP_1^* \to 2} \]
Pushing Cross-Fragment Joins

\[
p_1^4(f_4^V)
\]

\[
p_1^3(f_3^V) \quad \text{scan}_{a_4^{RP}:RP_3^*\rightarrow 4}
\]

\[
p_1^2(f_2^V) \quad \text{scan}_{a_3^{RP}:RP_1^*\rightarrow 3}
\]

\[
p_1^1(f_1^V) \quad \text{scan}_{a_2^{RP}:RP_1^*\rightarrow 2}
\]
Pushing Cross Fragment Joins: Implementation

- Can use full pipelining if both inputs to join are ordered
- Alternatively, can use index on root proxy nodes
- Full parallelism after first tuple received by local plan
Pushing Cross Fragment Joins

- Avoids accessing large portion of sub-trees within a fragment
- Can only be fully used in left-deep plans
- Decreases flexibility (e.g., where joins are performed)
Label Path Filtering

- Cross-fragment join pushing works well but decreases flexibility
- Goal: find a solution that can obtain partial benefit for scenarios where join pushing cannot be applied
- Idea: use selection instead of join to filter out some root proxy nodes
Label Path Filtering

- Assign to each proxy node the label path from the document root
- Filter for label paths that are compatible with the query

Diagram:

```
author
  ├── name
  │    └── first (William)
  │         └── last (Shakespeare)
  └── pubs
       └── book
            └── chapter
                 └── reference
```
Label Path Filtering

- Assign to each proxy node the label path from the document root
- Filter for label paths that are compatible with the query

\[
/\text{author}/\text{pubs}/\text{book}
\]
Label Path Filtering

- Assume there are two types of publications: book and article
- Can use selection to filter chapters based on publication type
Label Path Filtering

- Can be used in more cases
- Retains higher degree of flexibility
- Benefit is more limited (does not filter all irrelevant root proxy nodes)
Determining the Best Distributed Execution Plan

- Join pushing and label path filtering are not always advantageous
- Determine best execution plan using cost model
Outline

1. Fragmenting XML Collections

2. Querying Distributed XML Collections
   - Query Model
   - Distributed Query Evaluation
   - Improving Performance

3. Performance Evaluation

4. Conclusion
Performance Evaluation

- Implemented techniques within Natix
- 12 GB XMark collection (auction data)
- 1 Amazon EC2 instance for each of each of 10 vertical fragments
Performance Evaluation

- XPathMark queries (with few filtering value constraints)
- Modified, more selective XPathMark queries

<table>
<thead>
<tr>
<th></th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>/site/closed_auctions/closed_auction/annotation/description/text/keyword</td>
</tr>
<tr>
<td>A2</td>
<td>//closed_auction//keyword</td>
</tr>
<tr>
<td>A3</td>
<td>/site/closed_auctions/closed_auction//keyword</td>
</tr>
<tr>
<td>A4</td>
<td>/site/closed_auctions/closed_auction[annotation/description/text/keyword]/date</td>
</tr>
<tr>
<td>A5</td>
<td>/site/closed_auctions/closed_auction[descendant::keyword]/date</td>
</tr>
<tr>
<td>A6</td>
<td>/site/people/person[profile/gender and profile/age]/name</td>
</tr>
<tr>
<td>B7</td>
<td>//person[profile/@income]/name</td>
</tr>
<tr>
<td>A1S</td>
<td>/site/closed_auctions/closed_auction[price &gt; 600]/annotation/description/text/keyword</td>
</tr>
<tr>
<td>A2S</td>
<td>//closed_auction[price &gt; 600]//keyword</td>
</tr>
<tr>
<td>A3S</td>
<td>/site/closed_auctions/closed_auction[price &gt; 600]//keyword</td>
</tr>
<tr>
<td>A4S</td>
<td>/site/closed_auctions/closed_auction[price &gt; 600][annotation/description/text/keyword]/date</td>
</tr>
<tr>
<td>A5S</td>
<td>/site/closed_auctions/closed_auction[price &gt; 600][descendant::keyword]/date</td>
</tr>
<tr>
<td>A6S</td>
<td>/site/people/person[starts-with(name, 'Ry')][profile/gender and profile/age]/name</td>
</tr>
<tr>
<td>B7S</td>
<td>//person[starts-with(name, 'Ry')][profile/@income]/name</td>
</tr>
</tbody>
</table>
Performance Evaluation: XPathMark

![Bar Chart]

- **Y-axis**: Response time (seconds)
- **X-axis**: Plan (A1, A2, A3, A4, A5, A6, B7)

Legend:
- cent
- dist
- push

- **Plans**:
  - A1: Response time range
  - A2: Response time range
  - A3: Response time range
  - A4: Response time range
  - A5: Response time range
  - A6: Response time range
  - B7: Response time range

- **Cent** and **Dist** show different performance metrics.
- **Push** indicates a specific performance scenario.
Performance Evaluation: Selective XPathMark

![Graph showing response time for different plans (A1 to B7) and execution strategies (cent, dist, push). The x-axis represents the plan, and the y-axis represents the response time in seconds. The graph shows the performance comparison between different plans and execution strategies.]
Conclusions

- Distribution can make XML query evaluation more scalable
- Join pushing can significantly improve query performance
- A cost model is essential for finding the optimal technique for a given query
References
