Scalable XQuery Type Matching

Jens Teubner · IBM T. J. Watson Research Center
teubner@us.ibm.com
Scalable XQuery Type Matching

Type matching: Inspection of dynamic type information at runtime.

\[
\text{typeswitch}(x_1, x_2, \ldots, x_k) \\
\quad \text{case } t_1 \text{ return } e_1 \\
\quad \text{case } t_2 \text{ return } e_2 \\
\quad \vdots \\
\quad \text{case } t_n \text{ return } e_n \\
\quad \text{default return } e_{\text{def}}
\]

1. Compare \textbf{runtime types} of \((x_1, \ldots, x_k)\) against \(t_i\) in turn.
2. First matching branch determines expression result.

- Likewise:
  - \(e\) instance of \(t\)
  - \(e/ax::\text{element}(n,t)\)

This talk describes a scalable and efficient implementation for 1.

→ Leverage existing DBMS capabilities (aggregation).
→ Faithful to XQuery semantics.
**The XQuery Data Model**

**XQuery: item = value + type annotation**

\[
\begin{align*}
    x &= v \text{ of type } t \quad \text{(atomic values)} \\
    x &= \text{element } n \text{ of type } t \{ \cdots \} \quad \text{(element nodes)} \\
    x &= \text{attribute } n \text{ of type } t \{ \cdots \} \quad \text{(attribute nodes)} \\
    x &= \text{text } \{ \cdots \} \quad \text{(text nodes)} \\
    \ldots
\end{align*}
\]

- A **type annotation** \( t \) references a (named) XML Schema type.
- Type information may come, e.g., from a validated XML instance.
- **Type matching** is XQuery’s means to access type annotations.

---

1 Text, comment, and processing instruction nodes do not carry type information.
Types arrange into a **hierarchy**.

Derived types are added according to their **base type**.
Types arrange into a hierarchy.

Derived types are added according to their base type.
Types arrange into a hierarchy.

Derived types are added according to their base type.

```
let $x := my:hatsize(56)
return $x instance of xs:decimal
```

Existing implementations take the semantics of type matching quite literally.

→ Expensive recursion.
Type Ranks

Use **tree encoding** to encode type hierarchy.

- `pre`: preorder rank (of types!)
- `size`: number of derived types

> cf. XPath Accelerator

Use `pre` values to implement type annotations.

> “type ranks”

\[ t_1 \text{ derives from } t_2 \iff pre(t_2) \leq pre(t_1) \leq pre(t_2) + size(t_2) \]
Use **tree encoding** to encode type hierarchy.

- *pre*: preorder rank (of types!)
- *size*: number of derived types
- cf. XPath Accelerator

Use *pre* values to implement **type annotations**.

- “*type ranks*”

\[ t_1 \text{ derives from } t_2 \Leftrightarrow \]
\[ pre(t_2) \leq pre(t_1) \leq pre(t_2) + \text{size}(t_2) \]

known at compile time!
Type Ranks

let $x := \text{my:hatsize}(56)$
return

$x$ instance of \text{xs:decimal}

- $x = 56$ of type 8

$x$ instance of \text{xs:decimal}

$5 \leq 8 \leq 5 + 3$

Decidable in \text{constant time}. 
The argument to type matching typically is a sequence.

\[(x_1, \ldots, x_k) \text{ instance of } t \square \quad \square \in \{\sqcup, ?, +, *\}\]

The match succeeds iff

1. \(x_i\) matches \(t\) for all \(x_i\) in \(x = (x_1, \ldots, x_k)\) and
2. the sequence length \(k\) is compatible with the occurrence indicator \(\square\).
Sequences and Occurrence Indicators

Expressed in terms of type ranks:

1. $x_i$ matches $t$ for all $x_i$ in $x = (x_1, \ldots, x_k)$

\[ \iff \forall (x_i = v_i \text{ of type } t_i) \in x : \]

\[ \pre(t_i) \geq \pre(t) \land \pre(t_i) \leq \pre(t) + \size(t) \]
Sequences and Occurrence Indicators

Expressed in terms of type ranks:

$x_i$ matches $t$ for all $x_i$ in $x = (x_1, \ldots, x_k)$

\[ \iff \forall (x_i = v_i \text{ of type } t_i) \in x : \]

\[ pre(t_i) \geq pre(t) \land pre(t_i) \leq pre(t) + size(t) \]

Type aggregation:

\[ \iff \]

\[ \min_{(x_i = v_i \text{ of type } t_i) \in x} (pre(t_i)) \geq pre(t) \]

\[ \land \max_{(x_i = v_i \text{ of type } t_i) \in x} (pre(t_i)) \leq pre(t) + size(t) \]

Find minimum and maximum type ranks first, then compare once.
Aggregation (once more) beneficial for efficient XML processing.
Implementations highly tuned in today’s DBMSs.

Likewise:

Use aggregation to test compatibility with occurrence indicator $\square$:

1. the sequence length $k$ is compatible with $\square$

$\iff$

**Count** sequence items, then compare according to $\square$. 
Example: XQuery on purely relational database back-ends.²

<table>
<thead>
<tr>
<th>iter</th>
<th>pos</th>
<th>item</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>43</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>&quot;XL&quot;</td>
<td>9</td>
</tr>
</tbody>
</table>

- All loops unrolled, *iter*: logical iteration.
- *pos*: sequence order, *item* holds payload.
- new column *type*: preorder type ranks.

Type aggregation:

```
SELECT iter, MIN(type), MAX(type), COUNT(*)
FROM q
GROUP BY iter
```

²http://www.pathfinder-xquery.org/
Example:

1. Add type information to loop-lifted sequence encoding.
2. Aggregate, then compare.
   \[ \text{min} \geq 5 \land \text{max} \leq 5 + 3 \, ? \]
3. Projection re-establishes loop-lifted encoding.

→ Standard DBMS operators suffice.
Type Aggregation in an RDBMS

Proof-of-concept implementation using SQL.

- DB2 9 SQL
- FpML schema (777 types)
- 10,000 for iterations

Scalable XQuery Type Matching

Jens Teubner 12 / 14
Type Aggregation in an RDBMS

Proof-of-concept implementation using SQL.

![Graph showing execution time for different scenarios]

- **DB2 9 SQL**
- **FpML schema (777 types)**
- **10,000 for iterations**

**Average sequence length / iteration**

- **recursive**
- **type ranks**
- **type ranks + aggregation**

---

Scalable XQuery Type Matching 12 / 14
Type aggregation yields new runtime guarantees.

- **typeswitch**: Match a sequence against a number of types in turn.

**Traditional:**

\[
\text{match } O(k) \\
\vdots \\
\text{match } O(k) \\
\sum O(n \cdot k)
\]

**Type aggregation:**

\[
\text{typeswitch}(x_1, x_2, \ldots, x_k) \\
\begin{align*}
\text{case } t_1 & \text{ return } e_1 \\
\text{case } t_2 & \text{ return } e_2 \\
\vdots & \\
\text{case } t_n & \text{ return } e_n \\
\text{default return } e_{\text{def}}
\end{align*}
\]

\[
\begin{align*}
\text{aggregate } & O(k) \\
\text{compare } & O(1) \\
\text{compare } & O(1) \\
\vdots & \\
\text{compare } & O(1) \\
\sum & O(n + k)
\end{align*}
\]

- Recursion may further increase left-hand-side complexity.
Summary

A scalable implementation for XQuery’s dynamic type semantics.

- **Type ranks**: constant time for singleton type matching.
  - Inspired by XPath Accelerator tree encoding.

- **Type aggregation**: use aggregation to handle sequences.
  - Exploit efficient implementations in modern DBMSs.

- **New runtime guarantees**: $O(n \cdot k) \rightarrow O(n + k)$ for typeswitches

- Faithful to XQuery semantics.
  - Paper also covers XML node matching, incl. substitution groups