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.

```
typeswitch (x1, x2, ..., xk)
  case t1 return e1
  case t2 return e2
   ⋮
  case tn return en
  default return edef
```

1. Compare **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`::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

$x = v$ `of type` t	**(atomic values)**
$x =$ `element` n `of type` t `{` $\cdots$ `}`	**(element nodes)**
$x =$ `attribute` n `of type` t `{` $\cdots$ `}`	**(attribute nodes)**
$x =$ `text` `{` $\cdots$ `}`	**(text nodes)**[1]
$\ldots$	

- 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.

The XDM Type Hierarchy

```
xs:anyType
├ xs:untyped
├ xs:anySimpleType
│ ├ xs:anyAtomicType
│ │ ├ xs:boolean
│ │ ├ xs:decimal
│ │ │ └ xs:integer
│ │ ├ xs:string
│ │ └ xs:untypedAtomic
│ └ user-defd. list types
└ user-defd. complex types
```

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

The XDM Type Hierarchy

- xs:anyType
  - xs:untyped
  - xs:anySimpleType
    - xs:anyAtomicType
      - xs:boolean
      - xs:decimal
        - *my:shoesize*
        - xs:integer
          - *my:hatsize*
      - xs:string
      - xs:untypedAtomic
    - *my:hatsizelist*
  - *my:stockitem*

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

The XDM Type Hierarchy

```
xs:anyType
├ xs:untyped
├ xs:anySimpleType
│ └ xs:anyAtomicType
│   ├ xs:boolean
│   ├ xs:decimal
│   │ ├ my:shoesize
│   │ └ xs:integer
│   │   └ my:hatsize
│   ├ xs:string
│   └ xs:untypedAtomic
│ └ my:hatsizelist
└ my:stockitem
```

- 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

```
0 xs:anyType 12
├1 xs:untyped 0
├2 xs:anySimpleType 9
│ ├3 xs:anyAtomicType 7
│ │ ├4 xs:boolean 0
│ │ ├5 xs:decimal 3
│ │ │ ├6 my:shoesize 0
│ │ │ └7 xs:integer 1
│ │ │   └8 my:hatsize 0
│ │ ├9 xs:string 0
│ │ └10 xs:untypedAtomic 0
│ └11 my:hatsizelist 0
└12 my:stockitem 0
```

- 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"**

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

Type Ranks

- 0 `xs:anyType` 12
  - 1 `xs:untyped` 0
  - 2 `xs:anySimpleType` 9
    - 3 `xs:anyAtomicType` 7
      - 4 `xs:boolean` 0
      - 5 `xs:decimal` 3
        - 6 *`my:shoesize`* 0
        - 7 `xs:integer` 1
          - 8 *`my:hatsize`* 0
      - 9 `xs:string` 0
      - 10 `xs:untypedAtomic` 0
    - 11 *`my:hatsizelist`* 0
  - 12 *`my:stockitem`* 0

- 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"**

$$t_1 \text{ derives from } t_2$$
$$\Leftrightarrow$$
$$\underbrace{pre(t_2)} \leq pre(t_1) \leq \underbrace{pre(t_2) + size(t_2)}$$

known at compile time!

Type Ranks

```
0 xs:anyType 12
├ 1 xs:untyped 0
├ 2 xs:anySimpleType 9
│ ├ 3 xs:anyAtomicType 7
│ │ ├ 4 xs:boolean 0
│ │ ├ 5 xs:decimal 3
│ │ │ ├ 6 my:shoesize 0
│ │ │ └ 7 xs:integer 1
│ │ │   └ 8 my:hatsize 0
│ │ ├ 9 xs:string 0
│ │ └ 10 xs:untypedAtomic 0
│ └ 11 my:hatsizelist 0
└ 12 my:stockitem 0
```

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

- $x = 56 of type 8 (`my:hatsize`)

  `$x instance of xs:decimal`

  $$\Leftrightarrow$$

  $$\underbrace{5}_{\texttt{xs:decimal}} \leq 8 \leq \underbrace{5 + 3}_{\texttt{xs:decimal}}$$

- Decidable in **constant time**.

Sequences and Occurrence Indicators

The argument to type matching typically is a sequence.

$$(x_1\texttt{,}\ \ldots\texttt{,}\ x_k)\ \texttt{instance of}\ t\,\Box \qquad \Box \in \{\sqcup, \texttt{?}, \texttt{+}, \texttt{*}\}$$

The match succeeds iff

1. x_i matches t for all x_i in $x = (x_1\texttt{,}\ \ldots\texttt{,}\ x_k)$ and
2. the sequence length k is compatible with the occurrence indicator $\Box$.

Sequences and Occurrence Indicators

Expressed in terms of type ranks:

$$\boxed{1}\ x_i \text{ matches } t \text{ for all } x_i \text{ in } x = (x_1\,,\ \ldots\,,\, x_k)$$

$$\Leftrightarrow$$

$$\forall\,(x_i = v_i \texttt{ of type } t_i) \in x:$$

$$\mathit{pre}(t_i) \geq \mathit{pre}(t) \wedge \mathit{pre}(t_i) \leq \mathit{pre}(t) + \mathit{size}(t)$$

Expressed in terms of type ranks:

$$\boxed{1}\ x_i \text{ matches } t \text{ for all } x_i \text{ in } x = (x_1 \texttt{,} \ldots \texttt{,} x_k)$$

$$\Leftrightarrow$$

$$\forall\, (x_i = v_i \ \texttt{of type}\ t_i) \in x :$$
$$pre(t_i) \geq pre(t) \wedge pre(t_i) \leq pre(t) + size(t)$$

Type aggregation:

$$\Leftrightarrow$$

$$\min_{(x_i = v_i\ \texttt{of type}\ t_i) \in x} \big(pre(t_i)\big) \geq pre(t)$$
$$\wedge\ \max_{(x_i = v_i\ \texttt{of type}\ t_i) \in x} \big(pre(t_i)\big) \leq pre(t) + size(t)$$

Find minimum and maximum type ranks first, then compare once.

Type Aggregation

- Aggregation (once more) beneficial for efficient XML processing.
- Implementations highly tuned in today's DBMSs.

Likewise:

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

2 the sequence length k is compatible with $\square$

$$\Leftrightarrow$$

Count sequence items, then compare according to $\square$.

Type Aggregation in Relational XQuery

Example: XQuery on purely relational database back-ends.[2]

iter	*pos*	*item*	*type*
1	1	43	6
1	2	56	8
2	1	"XL"	9

- 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
```

[2] http://www.pathfinder-xquery.org/

Type Aggregation in Relational XQuery

Example:

`e instance of xs:decimal*`

1. Add type information to loop-lifted sequence encoding.
2. Aggregate, then compare.

$min \geq 5 \ \wedge \ max \leq 5 + 3$ **?**

3. Projection re-establishes loop-lifted encoding.

→ Standard DBMS operators suffice.

iter	*pos*	*item*	*type*
1	1	43	6
1	2	56	8
2	1	`"XL"`	9

`my:shoesize` `my:hatsize` `xs:string`

aggregate ↓

iter	*min*	*max*
1	6	8
2	9	9

compare ↓

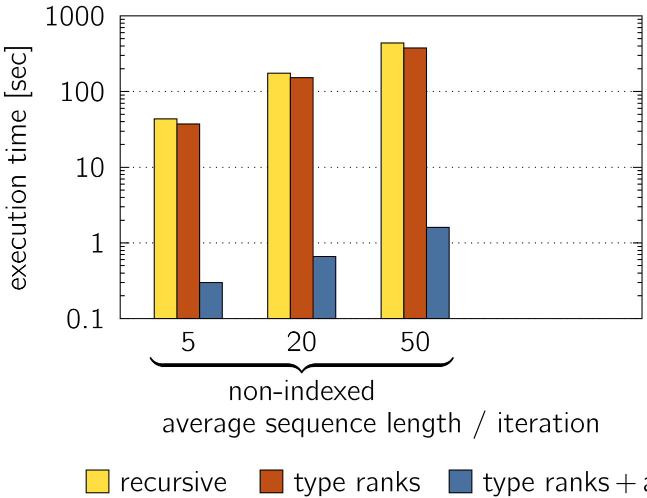
iter	*min*	*max*	*res*
1	6	8	`true`
2	9	9	`false`

project ↓

iter	*pos*	*item*	*type*
1	1	`true`	4
2	1	`false`	4

`xs:boolean`

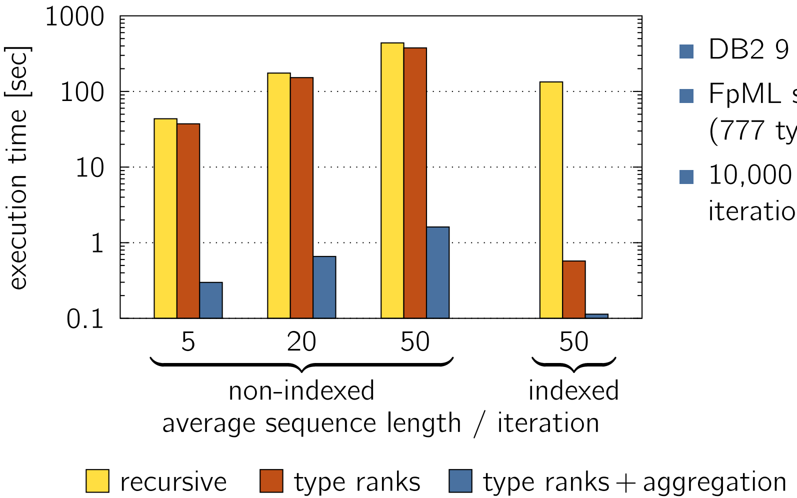
Proof-of-concept implementation using SQL.



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

Type Aggregation in an RDBMS

Proof-of-concept implementation using SQL.



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

Type aggregation yields new runtime guarantees.

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

Traditional:

match	$O(k)$
match	$O(k)$
$\vdots$	$\vdots$
match	$O(k)$
$\sum$	$O(n \cdot k)$

```
typeswitch (x1, x2, ..., xk)
  case t1 return e1
  case t2 return e2
  ⋮
  case tn return en
  default return edef
```

Type aggregation:

aggregate	$O(k)$
compare	$O(1)$
compare	$O(1)$
$\vdots$	$\vdots$
compare	$O(1)$
$\sum$	$O(n + k)$

- 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.
  $\rightarrow$ Inspired by XPath Accelerator tree encoding.
- **Type aggregation:** use aggregation to handle sequences.
  $\rightarrow$ Exploit efficient implementations in modern DBMSs.
- **New runtime guarantees:** $O(n \cdot k) \rightarrow O(n + k)$ for `typeswitch`es
- Faithful to **XQuery semantics**.
  $\rightarrow$ Paper also covers XML node matching, incl. substitution groups