XQuery on SQL Hosts

Torsten Grust

U Konstanz, Database Research Group

Torsten.Grust@uni-konstanz.de

S. Margherita di Pula—Sardinia, September 2004

7th EDBT Summer School on "XML and Databases"

A Purely Relational XQuery Processing Stack

- A **fully relational** XQuery processor, developed bottom-up:

A Purely Relational XQuery Processing Stack

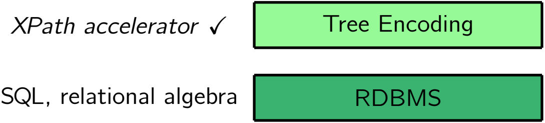
- A **fully relational** XQuery processor, developed bottom-up:

SQL, relational algebra

RDBMS

A Purely Relational XQuery Processing Stack

- A **fully relational** XQuery processor, developed bottom-up:

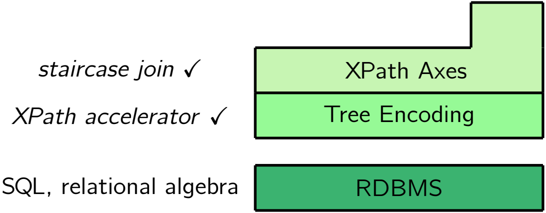


XPath accelerator ✓ | Tree Encoding

SQL, relational algebra | RDBMS

A Purely Relational XQuery Processing Stack

- A **fully relational** XQuery processor, developed bottom-up:



staircase join ✓

XPath Axes

XPath accelerator ✓

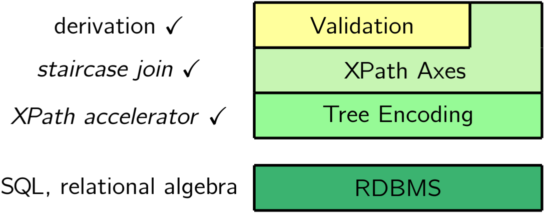
Tree Encoding

SQL, relational algebra

RDBMS

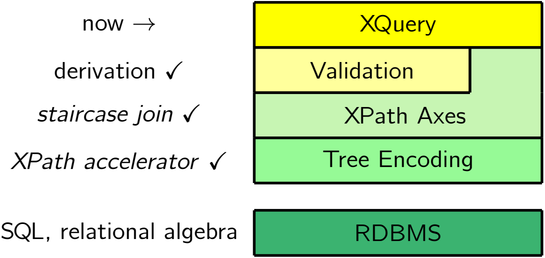
A Purely Relational XQuery Processing Stack

- A **fully relational** XQuery processor, developed bottom-up:



A Purely Relational XQuery Processing Stack

- A **fully relational** XQuery processor, developed bottom-up:



now → XQuery

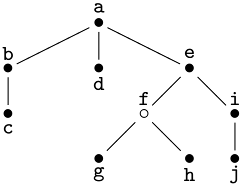
derivation ✓ Validation

staircase join ✓ XPath Axes

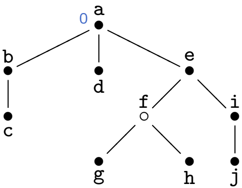
XPath accelerator ✓ Tree Encoding

SQL, relational algebra RDBMS

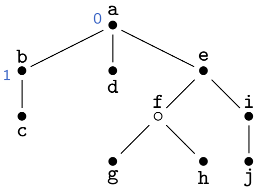
Node-based Relational Encodings of XQuery's Data Model



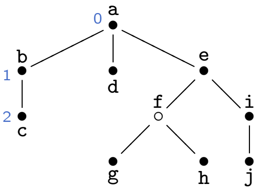
Node-based Relational Encodings of XQuery's Data Model



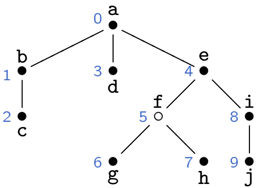
Node-based Relational Encodings of XQuery's Data Model



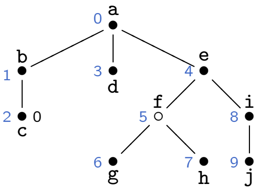
Node-based Relational Encodings of XQuery's Data Model



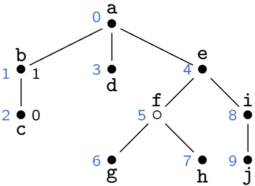
Node-based Relational Encodings of XQuery's Data Model



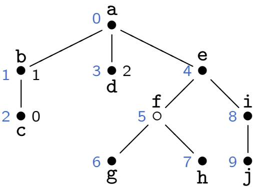
Node-based Relational Encodings of XQuery's Data Model



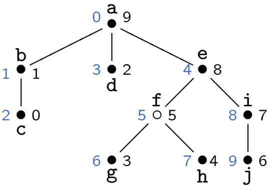
Node-based Relational Encodings of XQuery's Data Model



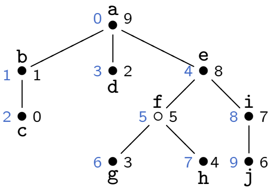
Node-based Relational Encodings of XQuery's Data Model

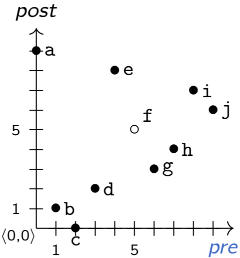


Node-based Relational Encodings of XQuery's Data Model

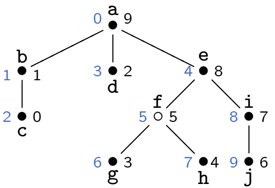


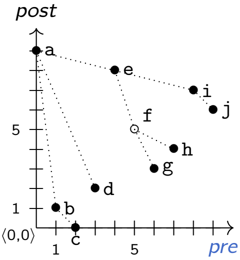
Node-based Relational Encodings of XQuery's Data Model



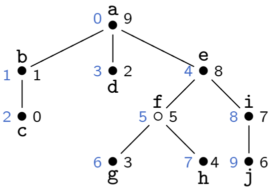


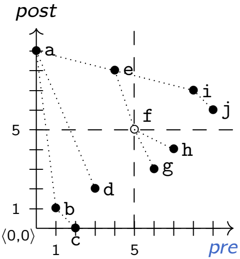
Node-based Relational Encodings of XQuery's Data Model



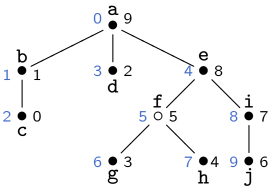


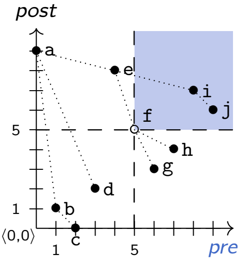
Node-based Relational Encodings of XQuery's Data Model



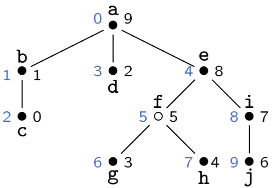


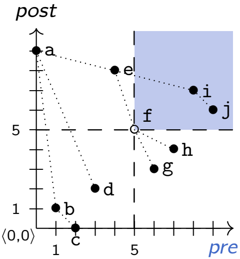
Node-based Relational Encodings of XQuery's Data Model





Node-based Relational Encodings of XQuery's Data Model

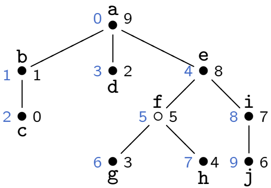


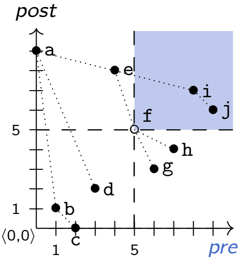


pre	post
0	9
1	1
2	0
3	2
4	8
5	5
6	3
7	4
8	7
9	6

- Staircase join (⊿) evaluates XPath axes on *pre*/*post* plane

Node-based Relational Encodings of XQuery's Data Model

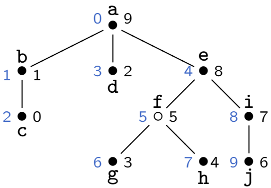


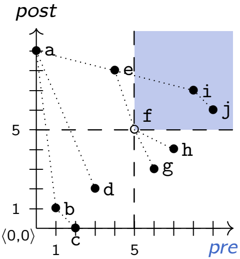


pre	post
0	9
1	1
2	0
3	2
4	8
5	5
6	3
7	4
8	7
9	6

- Staircase join (⌟) evaluates XPath axes on *pre*/*post* plane
- **Tuple ≙ node** (¬ tree-based)

Node-based Relational Encodings of XQuery's Data Model





pre	post
0	9
1	1
2	0
3	2
4	8
5	5
6	3
7	4
8	7
9	6

- Staircase join (⏌) evaluates XPath axes on *pre*/*post* plane
- **Tuple ≙ node** (¬ tree-based)
- Any encoding reflecting **node identity**/**document order** suffices

Source Language: XQuery Core

XQuery Core

- literals

Source Language: XQuery Core

XQuery Core

- literals
- sequences $(e_1\,,e_2)$

Source Language: XQuery Core

XQuery Core

- literals
- sequences $(e_1\,,e_2)$
- variables ($\$v$)
- `let` $\cdots$ `return`
- `for` $\cdots$ `where` $\cdots$ `return`

Source Language: XQuery Core

XQuery Core

- literals
- sequences (e_1, e_2)
- variables ($\$v$)
- `let` $\cdots$ `return`
- `for` $\cdots$ `where` $\cdots$ `return`
- `for` $\cdots$ [at $\$v$] $\cdots$ `where` $\cdots$ `return`

Source Language: XQuery Core

XQuery Core

- literals
- sequences (e_1, e_2)
- variables ($\$v$)
- `let` $\cdots$ `return`
- `for` $\cdots$ `where` $\cdots$ `return`
- `for` $\cdots$ [`at` $\$v$] $\cdots$ `where` $\cdots$ `return`
- `if` $\cdots$ `then` $\cdots$ `else`
- `typeswitch` $\cdots$ `case` $\cdots$ `default`

Source Language: XQuery Core

XQuery Core

- literals
- sequences (e_1, e_2)
- variables (\$$v$)
- `let` ··· `return`
- `for` ··· `where` ··· `return`
- `for` ··· [`at` \$$v$] ··· `where` ··· `return`
- `if` ··· `then` ··· `else`
- `typeswitch` ··· `case` ··· `default`
- `element {`···`} {`···`}`
- `text {`···`}`
- XPath (e/α)

Source Language: XQuery Core

XQuery Core

- literals
- sequences (e_1 `,` e_2)
- variables (`$`v)
- `let` ··· `return`
- `for` ··· `where` ··· `return`
- `for` ··· [`at $`v] ··· `where` ··· `return`
- `if` ··· `then` ··· `else`
- `typeswitch` ··· `case` ··· `default`
- `element {`···`} {`···`}`
- `text {`···`}`
- XPath (e/α)
- function application
- document order (e_1 `<<` e_2)
- node identity (e_1 `is` e_2)

Source Language: XQuery Core

XQuery Core

- literals
- sequences $(e_1\,,e_2)$
- variables (`$`v)
- `let` ⋯ `return`
- `for` ⋯ `where` ⋯ `return`
- `for` ⋯ [`at $`v] ⋯ `where` ⋯ `return`
- `if` ⋯ `then` ⋯ `else`
- `typeswitch` ⋯ `case` ⋯ `default`
- `element {`⋯`} {`⋯`}`
- `text {`⋯`}`
- XPath (e/α)
- function application

- document order (e_1 `<<` e_2)
- node identity (e_1 `is` e_2)
- arithmetics (`+`,`-`,`*`,`idiv`)

Source Language: XQuery Core

XQuery Core

- literals
- sequences (e_1 , e_2)
- variables ($\$v$)
- let ··· return
- for ··· where ··· return
- for ··· [at $\$v$] ··· where ··· return
- if ··· then ··· else
- typeswitch ··· case ··· default
- element {···} {···}
- text {···}
- XPath (e/α)
- function application

- document order (e_1 << e_2)
- node identity (e_1 is e_2)
- arithmetics (+,−,∗,idiv)
- fn:doc()
- fn:root()
- fn:data()
- fn:distinct-doc-order()
- fn:count()
- fn:sum()
- fn:empty()
- fn:position()
- fn:last()

Source Language: XQuery Core

XQuery Core

- literals
- sequences (e_1 , e_2)
- variables ($\$v$)
- `let` ··· `return`
- `for` ··· `where` ··· `return`
- `for` ··· [`at` $\$v$] ··· `where` ··· `return`
- `if` ··· `then` ··· `else`
- `typeswitch` ··· `case` ··· `default`
- `element` `{`···`}` `{`···`}`
- `text` `{`···`}`
- XPath (e/α)
- function application
- document order (e_1 `<<` e_2)
- node identity (e_1 `is` e_2)
- arithmetics (`+`,`-`,`*`,`idiv`)
- `fn:doc()`
- `fn:root()`
- `fn:data()`
- `fn:distinct-doc-order()`
- `fn:count()`
- `fn:sum()`
- `fn:empty()`
- `fn:position()`
- `fn:last()`

- Expression may nest as defined by W3C XQuery Working Draft

Here: Focus on non-XPath-related XQuery Fragment

- Item sequences, **sequence order**

Here: Focus on non-XPath-related XQuery Fragment

- Item sequences, **sequence order**

```
(1.0, "x", <a/>)
```

Here: Focus on non-XPath-related XQuery Fragment

- Item sequences, **sequence order**

`(1.0, "x", <a/>)` ↦

item	
node	atom
NULL	"1.0"
NULL	"x"
pre(a)	NULL

Here: Focus on non-XPath-related XQuery Fragment

- Item sequences, **sequence order**

`(1.0, "x", <a/>)` ↦

	item	
pos	node	atom
1	NULL	"1.0"
2	NULL	"x"
3	*pre*(a)	NULL

Here: Focus on non-XPath-related XQuery Fragment

- Item sequences, **sequence order**

`(1.0, "x", <a/>)` ↦

	item	
pos	node	atom
1	`NULL`	`"1.0"`
2	`NULL`	`"x"`
3	*pre*(`a`)	`NULL`

pos	item
1	`1.0`
2	`"x"`
3	*pre*(`a`)

Here: Focus on non-XPath-related XQuery Fragment

- Item sequences, **sequence order**

`(1.0, "x", <a/>)` ↦

	item	
pos	node	atom
1	NULL	"1.0"
2	NULL	"x"
3	*pre*(a)	NULL

pos	item
1	1.0
2	"x"
3	*pre*(a)

- Compiling **nested** `for` ··· `let` ··· `where` ··· `return` **blocks**

Here: Focus on non-XPath-related XQuery Fragment

- Item sequences, **sequence order**

`(1.0, "x", <a/>)` ↦

	item	
pos	node	atom
1	NULL	"1.0"
2	NULL	"x"
3	*pre*(a)	NULL

pos	item
1	1.0
2	"x"
3	*pre*(a)

- Compiling **nested** `for` ··· `let` ··· `where` ··· `return` **blocks**
  - ▷ **Variable representation** and **scopes**

Here: Focus on non-XPath-related XQuery Fragment

- Item sequences, **sequence order**

`(1.0, "x", <a/>)` ↦

	item	
pos	node	atom
1	NULL	"1.0"
2	NULL	"x"
3	*pre*(a)	NULL

pos	item
1	1.0
2	"x"
3	*pre*(a)

- Compiling **nested** `for` ··· `let` ··· `where` ··· `return` **blocks**
  - ▷ **Variable representation** and **scopes**
- Node construction
- XPath evaluation over persistent and transient nodes

Target Language: Flat Relational Algebra

Relational Algebra

π	column projection, renaming
σ	row selection

Target Language: Flat Relational Algebra

Relational Algebra

π	column projection, renaming
σ	row selection
$\dot{\cup}$, $\setminus$	disjoint union, difference
δ	duplicate elimination

Target Language: Flat Relational Algebra

Relational Algebra

π	column projection, renaming
σ	row selection
$\dot{\cup}$, $\setminus$	disjoint union, difference
δ	duplicate elimination
$\bowtie$	equi-join
$\times$	Cartesian product

Target Language: Flat Relational Algebra

Relational Algebra

π	column projection, renaming
σ	row selection
$\dot{\cup}$, $\setminus$	disjoint union, difference
δ	duplicate elimination
$\bowtie$	equi-join
$\times$	Cartesian product
ϱ	row numbering

Target Language: Flat Relational Algebra

Relational Algebra

π	column projection, renaming
σ	row selection
$\dot\cup$, $\setminus$	disjoint union, difference
δ	duplicate elimination
$\bowtie$	equi-join
$\times$	Cartesian product
ϱ	row numbering
⊿	staircase join
ε, τ	element/text node construction

Target Language: Flat Relational Algebra

Relational Algebra

π	column projection, renaming
σ	row selection
$\dot{\cup}$, $\setminus$	disjoint union, difference
δ	duplicate elimination
$\bowtie$	equi-join
$\times$	Cartesian product
ϱ	row numbering
⊿	staircase join
ε, τ	element/text node construction
$\circledast$	arithmetic/comparison/Boolean operator $*$

Target Language: Flat Relational Algebra

Relational Algebra

π	column projection, renaming
σ	row selection
$\dot{\cup}$, $\setminus$	disjoint union, difference
δ	duplicate elimination
$\bowtie$	equi-join
$\times$	Cartesian product
ϱ	row numbering
⏌	staircase join
ε, τ	element/text node construction
$\circledast$	arithmetic/comparison/Boolean operator $*$

- **No tree pattern matching** or similar operators involved here

Target Language: Flat Relational Algebra

Relational Algebra

π	column projection, renaming
σ	row selection
$\dot\cup$, $\setminus$	disjoint union, difference
δ	duplicate elimination
$\bowtie$	equi-join
$\times$	Cartesian product
ϱ	row numbering
⎍	staircase join
ε, τ	element/text node construction
$\circledast$	arithmetic/comparison/Boolean operator $*$

- **No tree pattern matching** or similar operators involved here
- This algebra is efficiently implementable on (top of) SQL hosts

Iteration in XQuery Core

- XQuery Core has been designed around the `for` **iteration** primitive:

XQuery iteration

$$\texttt{for } \$v \texttt{ in } (x_1\texttt{,}x_2\texttt{,}\ldots\texttt{,}x_n) \texttt{ return } e$$

Iteration in XQuery Core

- XQuery Core has been designed around the `for` **iteration** primitive:

XQuery iteration

$$\texttt{for } \$v \texttt{ in } (x_1, x_2, \ldots, x_n) \texttt{ return } e$$
$$\equiv$$
$$(e[{}^{x_1}/\$v], e[{}^{x_2}/\$v], \ldots, e[{}^{x_n}/\$v])$$

- Representation of $(x_1, x_2, \ldots, x_n)$:

Iteration in XQuery Core

- XQuery Core has been designed around the `for` **iteration** primitive:

XQuery iteration

$$\texttt{for } \$v \texttt{ in } (x_1,x_2,\ldots,x_n) \texttt{ return } e$$

$$\equiv$$

$$(e[^{x_1}/\$v],e[^{x_2}/\$v],\ldots,e[^{x_n}/\$v])$$

- Representation of $(x_1,x_2,\ldots,x_n)$:

pos	item
1	x_1
2	x_2
$\vdots$	$\vdots$
n	x_n

Iteration in XQuery Core

- XQuery Core has been designed around the `for` **iteration** primitive:

XQuery iteration

$$\texttt{for } \$v \texttt{ in } (x_1, x_2, \ldots, x_n) \texttt{ return } e$$
$$\equiv$$
$$(e[^{x_1}/\$v], e[^{x_2}/\$v], \ldots, e[^{x_n}/\$v])$$

- Representation of $(x_1, x_2, \ldots, x_n)$:

pos	item
1	x_1
2	x_2
$\vdots$	$\vdots$
n	x_n

- Derive $\$v$ as follows:

Iteration in XQuery Core

- XQuery Core has been designed around the `for` **iteration** primitive:

XQuery iteration

$$\texttt{for } \$v \texttt{ in } (x_1, x_2, \ldots, x_n) \texttt{ return } e$$
$$\equiv$$
$$(e[{}^{x_1}/_{\$v}], e[{}^{x_2}/_{\$v}], \ldots, e[{}^{x_n}/_{\$v}])$$

- Representation of $(x_1, x_2, \ldots, x_n)$:

pos	item
1	x_1
2	x_2
$\vdots$	$\vdots$
n	x_n

- Derive $\$v$ as follows:

pos	item
1	x_1
2	x_2
$\vdots$	$\vdots$
n	x_n

Iteration in XQuery Core

- XQuery Core has been designed around the `for` **iteration** primitive:

XQuery iteration

$$\texttt{for}\ \$v\ \texttt{in}\ (x_1\texttt{,}x_2\texttt{,}\ldots\texttt{,}x_n)\ \texttt{return}\ e$$

$$\equiv$$

$$(e[{}^{x_1}\!/\$v]\texttt{,}e[{}^{x_2}\!/\$v]\texttt{,}\ldots\texttt{,}e[{}^{x_n}\!/\$v])$$

- Representation of $(x_1\texttt{,}x_2\texttt{,}\ldots\texttt{,}x_n)$:

pos	item
1	x_1
2	x_2
$\vdots$	$\vdots$
n	x_n

- Derive $\$v$ as follows:

iter	pos	item
1	1	x_1
2	2	x_2
$\vdots$	$\vdots$	$\vdots$
n	n	x_n

Iteration in XQuery Core

- XQuery Core has been designed around the `for` **iteration** primitive:

XQuery iteration

$$\texttt{for \$}v\texttt{ in (}x_1\texttt{,}x_2\texttt{,}\ldots\texttt{,}x_n\texttt{) return } e$$
$$\equiv$$
$$\texttt{(}e[{}^{x_1}/\texttt{\$}v]\texttt{,}e[{}^{x_2}/\texttt{\$}v]\texttt{,}\ldots\texttt{,}e[{}^{x_n}/\texttt{\$}v]\texttt{)}$$

- Representation of $\texttt{(}x_1\texttt{,}x_2\texttt{,}\ldots\texttt{,}x_n\texttt{)}$:

pos	item
1	x_1
2	x_2
$\vdots$	$\vdots$
n	x_n

- Derive $\texttt{\$}v$ as follows:

iter	pos	item
1	1	x_1
2	1	x_2
$\vdots$	$\vdots$	$\vdots$
n	1	x_n

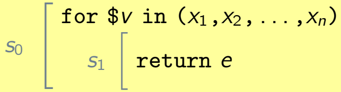
Loop Lifting

- Subexpressions are compiled in dependence of **iteration scope** s in which they appear—represented as unary relation $\mathit{loop}(s)$

Loop Lifting

- Subexpressions are compiled in dependence of **iteration scope** *s* in which they appear—represented as unary relation *loop*(*s*)

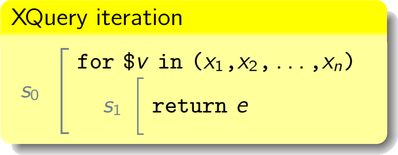
XQuery iteration



```
for $v in ($x_1$,$x_2$, . . . ,$x_n$)
    return $e$
```

Loop Lifting

- Subexpressions are compiled in dependence of **iteration scope** s in which they appear—represented as unary relation $loop(s)$



XQuery iteration

s_0 : for $v in ($x_1$,$x_2$,...,$x_n$)

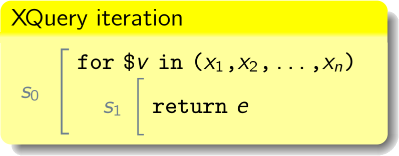
s_1 : return e

$loop(s_0)$

iter
1

Loop Lifting

- Subexpressions are compiled in dependence of **iteration scope** s in which they appear—represented as unary relation $loop(s)$



XQuery iteration

```
s0 [ for $v in (x1,x2,...,xn)
   [   s1 [ return e
```

$loop(s_0)$

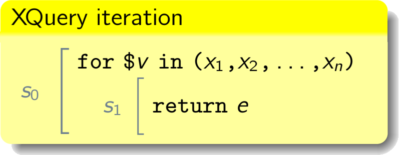
iter
1

$loop(s_1)$

iter
1
⋮
n

Loop Lifting

- Subexpressions are compiled in dependence of **iteration scope** s in which they appear—represented as unary relation $loop(s)$



XQuery iteration

s_0 `for` $\$v$ `in` $(x_1, x_2, \ldots, x_n)$

s_1 `return` e

$loop(s_0)$

iter
1

$loop(s_1)$

iter
1
⋮
n

▷ Single item `"a"` in scope s_1:

Loop Lifting

- Subexpressions are compiled in dependence of **iteration scope** s in which they appear—represented as unary relation $loop(s)$

XQuery iteration

s_0 for $\$v$ in $(x_1, x_2, \ldots, x_n)$

s_1 return e

$loop(s_0)$

iter
1

$loop(s_1)$

iter
1
⋮
n

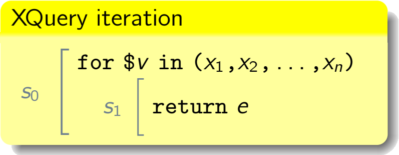
▷ Single item "a" in scope s_1:

$loop(s_1)$ ×

pos	item
1	"a"

Loop Lifting

- Subexpressions are compiled in dependence of **iteration scope** s in which they appear—represented as unary relation $loop(s)$



XQuery iteration

s_0

```
for $v in (x1,x2,...,xn)
    return e
```

s_1

$loop(s_0)$

iter
1

$loop(s_1)$

iter
1
⋮
n

▷ Single item "a" in scope s_1:

iter	pos	item
1	1	"a"
⋮	⋮	⋮
n	1	"a"

Loop Lifting

- Subexpressions are compiled in dependence of **iteration scope** s in which they appear—represented as unary relation $loop(s)$

XQuery iteration

s_0 [`for $v in (`x_1`,`x_2`, . . . ,`x_n`)`

s_1 [`return` e

$loop(s_0)$

iter
1

$loop(s_1)$

iter
1
⋮
n

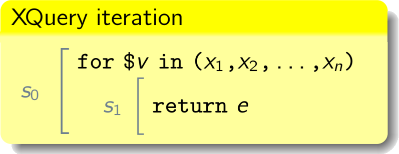
▷ Single item `"a"` in scope s_1:

iter	pos	item
1	1	"a"
⋮	⋮	⋮
n	1	"a"

▷ Sequence `("a","b")` in scope s_1:

Loop Lifting

- Subexpressions are compiled in dependence of **iteration scope** s in which they appear—represented as unary relation $loop(s)$



XQuery iteration

s_0 [`for` $\$v$ `in` $(x_1, x_2, \ldots, x_n)$

s_1 [`return` e

$loop(s_0)$

iter
1

$loop(s_1)$

iter
1
⋮
n

▷ Single item `"a"` in scope s_1:

iter	pos	item
1	1	"a"
⋮	⋮	⋮
n	1	"a"

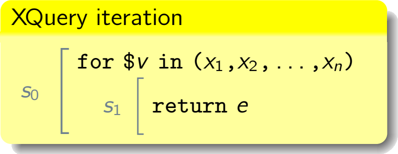
▷ Sequence `("a","b")` in scope s_1:

$loop(s_1)$ ×

pos	item
1	"a"
2	"b"

Loop Lifting

- Subexpressions are compiled in dependence of **iteration scope** s in which they appear—represented as unary relation *loop*(s)



XQuery iteration

s_0 [`for` $\$v$ `in` $(x_1,x_2,\ldots,x_n)$

s_1 [`return` e

loop(s_0)

iter
1

loop(s_1)

iter
1
⋮
n

▷ Single item `"a"` in scope s_1:

iter	pos	item
1	1	"a"
⋮	⋮	⋮
n	1	"a"

▷ Sequence `("a","b")` in scope s_1:

iter	pos	item
1	1	"a"
1	2	"b"
⋮	⋮	⋮
n	1	"a"
n	2	"b"

Nested Scopes

Nested for blocks

```
$s_0$ [ for $$v_0$ in (10,20)
        $s_1$ [ for $$v_1$ in (100,200)
                $s_2$ [ return $$v_0$ + $$v_1$
```

loop(s_0)

iter
1

loop(s_1)

iter
1
2

Nested Scopes

Nested `for` blocks

```
     for $v0 in (10,20)
s0     for $v1 in (100,200)
   s1    s2  return $v0 + $v1
```

loop(s_0)

iter
1

loop(s_1)

iter
1
2

loop(s_2)

iter
1
2
3
4

Nested Scopes

Nested `for` blocks

```
     [ for $v0 in (10,20)
s0   [         [ for $v1 in (100,200)
     [     s1  [      s2 [ return $v0 + $v1
```

loop(s_0)

iter
1

loop(s_1)

iter
1
2

loop(s_2)

iter
1
2
3
4

- Derive $\$v_0, \$v_1$ as before (uses row numbering operator ϱ):

$\$v_0$ in s_1:

iter	pos	item
1	1	10
1	2	20

Nested Scopes

Nested `for` blocks

```
s0 [ for $v0 in (10,20)
s1   [ for $v1 in (100,200)
s2       [ return $v0 + $v1
```

loop(s_0)

iter
1

loop(s_1)

iter
1
2

loop(s_2)

iter
1
2
3
4

- Derive $\$v_0, \$v_1$ as before (uses row numbering operator ϱ):

$\$v_0$ in s_1:

iter	pos	item
1	1	10
2	2	20

Nested Scopes

Nested for blocks

s_0 for $\$v_0$ in (10,20)

s_1 for $\$v_1$ in (100,200)

s_2 return $\$v_0$ + $\$v_1$

loop(s_0)

iter
1

loop(s_1)

iter
1
2

loop(s_2)

iter
1
2
3
4

- Derive $\$v_0, \$v_1$ as before (uses row numbering operator ϱ):

$\$v_0$ in s_1:

iter	pos	item
1	1	10
2	1	20

$\$v_1$ in s_2:

iter	pos	item
1	1	100
1	2	200
2	1	100
2	2	200

Nested Scopes

Nested `for` blocks

```
s0 [ for $v0 in (10,20)
     s1 [ for $v1 in (100,200)
          s2 [ return $v0 + $v1
```

$loop(s_0)$

iter
1

$loop(s_1)$

iter
1
2

$loop(s_2)$

iter
1
2
3
4

- Derive $\$v_0, \$v_1$ as before (uses row numbering operator ϱ):

$\$v_0$ in s_1:

iter	pos	item
1	1	10
2	1	20

$\$v_1$ in s_2:

iter	pos	item
1	1	100
2	2	200
3	1	100
4	2	200

Nested Scopes

Nested `for` blocks

```
s0  for $v0 in (10,20)
       s1  for $v1 in (100,200)
              s2  return $v0 + $v1
```

loop(s_0)

iter
1

loop(s_1)

iter
1
2

loop(s_2)

iter
1
2
3
4

- Derive $\$v_0, \$v_1$ as before (uses row numbering operator ϱ):

$\$v_0$ in s_1:

iter	pos	item
1	1	10
2	1	20

$\$v_1$ in s_2:

iter	pos	item
1	1	100
2	1	200
3	1	100
4	1	200

Nested Scopes

Nested for blocks

```
s0 [ for $v0 in (10,20)
        s1 [ for $v1 in (100,200)
                s2 [ return $v0 + $v1
```

loop(s_0)

iter
1

loop(s_1)

iter
1
2

loop(s_2)

iter
1
2
3
4

- Derive $\$v_0, \$v_1$ as before (uses row numbering operator ϱ):

$\$v_0$ in s_1:

iter	pos	item
1	1	10
2	1	20

$\$v_1$ in s_2:

iter	pos	item
1	1	100
2	1	200
3	1	100
4	1	200

▷ Variable $\$v_0$ in scope s_2?

Relating Scopes

Nested for blocks

```
s0 [ for $v_0 in (10,20)
   [     s1 [ for $v_1 in (100,200)
   [        [     s2 [ return $v_0 + $v_1
```

- Relation *map* captures the semantics of nested iteration:

map:

inner	outer

Relating Scopes

Nested `for` blocks

```
$s_0$  for $\$v_0$ in (10,20)
       $s_1$  for $\$v_1$ in (100,200)
              $s_2$  return $\$v_0$ + $\$v_1$
```

- Relation *map* captures the semantics of nested iteration:

map:

inner	outer
1	1

Relating Scopes

Nested for blocks

```
     [ for $v0 in (10,20)
     |     [ for $v1 in (100,200)
s0   |  s1 |      s2 [ return $v0 + $v1
```

- Relation *map* captures the semantics of nested iteration:

map:

inner	outer
1	1
2	1

Relating Scopes

Nested for blocks

```
s0 [ for $v0 in (10,20)
   [      s1 [ for $v1 in (100,200)
   [         [      s2 [ return $v0 + $v1
```

- Relation *map* captures the semantics of nested iteration:

map:

inner	outer
1	1
2	1
3	2

Relating Scopes

Nested for blocks

```
    [ for $v0 in (10,20)
s0  [           [ for $v1 in (100,200)
    [      s1   [       s2 [ return $v0 + $v1
```

- Relation *map* captures the semantics of nested iteration:

map:

inner	outer
1	1
2	1
3	2
4	2

Relating Scopes

Nested for blocks

```
   for $v0 in (10,20)
s0    for $v1 in (100,200)
   s1    s2 return $v0 + $v1
```

- Relation *map* captures the semantics of nested iteration:

map:

inner	outer
1	1
2	1
3	2
4	2

  - ▷ Representation of $\$v_0$ in s_2:

$$\pi_{iter:inner,pos,item}(\$v_0 \bowtie_{iter=outer} map)$$

Relating Scopes

Nested `for` blocks

s_0 for $\$v_0$ in (10,20)

  s_1 for $\$v_1$ in (100,200)

    s_2 return $\$v_0 + \$v_1$

- Relation *map* captures the semantics of nested iteration:

map:

inner	outer
1	1
2	1
3	2
4	2

  ▷ Representation of $\$v_0$ in s_2:

$$\pi_{iter:inner,pos,item}(\$v_0 \bowtie_{iter=outer} map) =$$

iter	pos	item
1	1	10
2	1	10
3	1	20
4	1	20

Evaluation in scope s_2

```
for $v_0 in (10,20)
   for $v_1 in (100,200)
     s_2 [ return $v_0 + $v_1
```

Evaluation in scope s_2

```
for $\$v_0$ in (10,20)
   for $\$v_1$ in (100,200)
      $s_2$ return $\$v_0$ + $\$v_1$
```

$\$v_0$

iter_0	pos_0	item_0
1	1	10
2	1	10
3	1	20
4	1	20

Evaluation in scope s_2

```
for $v_0 in (10,20)
   for $v_1 in (100,200)
  s_2 [ return $v_0 + $v_1
```

$\$v_0$

iter_0	pos_0	item_0
1	1	10
2	1	10
3	1	20
4	1	20

$\$v_1$

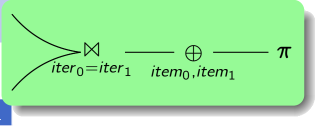
iter_1	pos_1	item_1
1	1	100
2	1	200
3	1	100
4	1	200

Evaluation in scope s_2

```
for $v0 in (10,20)
   for $v1 in (100,200)
      s2 [ return $v0 + $v1
```

$\$v_0$

$iter_0$	pos_0	$item_0$
1	1	10
2	1	10
3	1	20
4	1	20



$\$v_1$

$iter_1$	pos_1	$item_1$
1	1	100
2	1	200
3	1	100
4	1	200

Evaluation in scope s_2

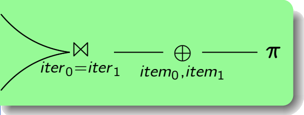
```
for $v0 in (10,20)
   for $v1 in (100,200)
      s2 [ return $v0 + $v1
```

$\$v_0$

iter_0	pos_0	item_0
1	1	10
2	1	10
3	1	20
4	1	20

$\$v_1$

iter_1	pos_1	item_1
1	1	100
2	1	200
3	1	100
4	1	200



=

iter	pos	item
1	1	110
2	1	210
3	1	120
4	1	220

Empty Sequences

- Encode () by **absence** of *iter* value in loop-lifted sequence *e*:

loop:

iter
1
2
3

Empty Sequences

- Encode `()` by **absence** of *iter* value in loop-lifted sequence *e*:

loop:

iter
1
2
3

e:

iter	pos	item
1	1	`"a"`
3	1	`"x"`
3	2	`"y"`

Empty Sequences

- Encode `()` by **absence** of *iter* value in loop-lifted sequence *e*:

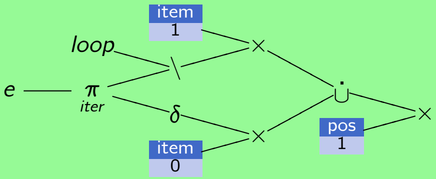
loop:

iter
1
2
3

e:

iter	pos	item
1	1	`"a"`
3	1	`"x"`
3	2	`"y"`

Compile `fn:empty(`*e*`)`



Empty Sequences

- Encode `()` by **absence** of *iter* value in loop-lifted sequence *e*:

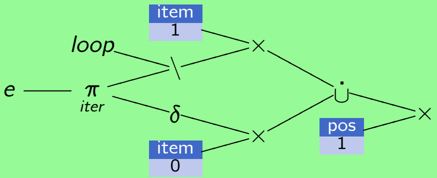
loop:

iter
1
2
3

e:

iter	pos	item
1	1	`"a"`
3	1	`"x"`
3	2	`"y"`

Compile `fn:empty(`*e*`)`



=

iter	pos	item
1	1	0
2	1	1
3	1	0

Example: Compiling Conditional Expressions

XQuery conditional expression

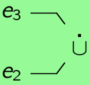
`if` (e_1) **`then`** e_2 **`else`** e_3

Example: Compiling Conditional Expressions

XQuery conditional expression

`if` (e_1) `then` e_2 `else` e_3

Equivalent algebraic code

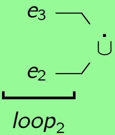


Example: Compiling Conditional Expressions

XQuery conditional expression

`if` (e_1) `then` e_2 `else` e_3

Equivalent algebraic code

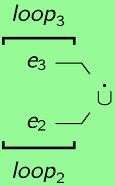


Example: Compiling Conditional Expressions

XQuery conditional expression

if (e_1) **then** e_2 **else** e_3

Equivalent algebraic code

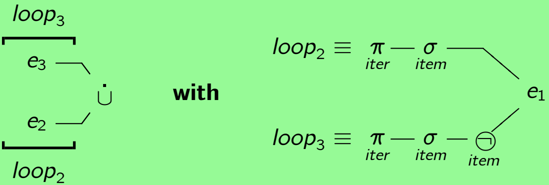


Example: Compiling Conditional Expressions

XQuery conditional expression

if (e_1) **then** e_2 **else** e_3

Equivalent algebraic code

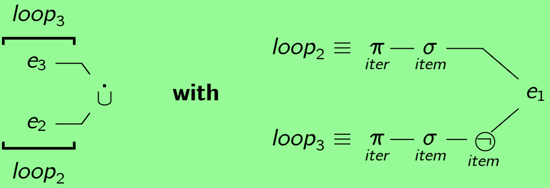


Example: Compiling Conditional Expressions

XQuery conditional expression

`if` (e_1) `then` e_2 `else` e_3

Equivalent algebraic code



$loop_3$
e_3
$\dot{\cup}$
e_2
$loop_2$
with
$loop_2 \equiv \pi_{iter} — \sigma_{item}$
e_1
$loop_3 \equiv \pi_{iter} — \sigma_{item} — \textcircled{\neg}_{item}$

- $\textcircled{\neg}_c$ denotes the algebra's Boolean negation operator (column c)
- σ_c selects all tuples with column $c \neq 0$

Inference Rules

- The compiler is specified in terms of **inference rules**, collectively defining the $\Mapsto$ (*compiles to*) function

Inference Rules

- The compiler is specified in terms of **inference rules**, collectively defining the ⤇ (*compiles to*) function
- Here is the (somewhat simplified) inference rule for the compilation of `if···then···else`:

Inference rule `if···then···else`

$$\frac{\begin{array}{c}\Gamma; loop \vdash e_1 \Mapsto q_1 \\ loop_2 \equiv \pi_{iter}(\sigma_{item}(q_1)) \qquad loop_3 \equiv \pi_{iter}(\sigma_{item}(\bigcirc\kern-0.75em\neg_{\,item}(q_1))) \\ \Gamma; loop_2 \vdash e_2 \Mapsto q_2 \qquad \Gamma; loop_3 \vdash e_3 \Mapsto q_3\end{array}}{\Gamma; loop \vdash \texttt{if (}e_1\texttt{) then } e_2 \texttt{ else } e_3 \Mapsto q_2 \mathbin{\dot\cup} q_3}$$

  - ▷ Γ denotes an environment mapping variables to their compiled equivalent

Example: Evaluation of a Conditional Expression

XQuery expression

$$\texttt{for \$x in 1 to 4}$$

$$s_1 \left[\ \texttt{return if (}\underbrace{\texttt{\$}x\ \texttt{mod 2 = 0}}_{e_1}\texttt{) then }\underbrace{\texttt{"even"}}_{e_2}\texttt{ else }\underbrace{\texttt{"odd"}}_{e_3}\right.$$

Example: Evaluation of a Conditional Expression

XQuery expression

```
for $x in 1 to 4
```

s_1 return if ($\underbrace{\texttt{\$}x\ \texttt{mod 2 = 0}}_{e_1}$) then $\underbrace{\texttt{"even"}}_{e_2}$ else $\underbrace{\texttt{"odd"}}_{e_3}$

Evaluation

iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

Example: Evaluation of a Conditional Expression

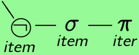
XQuery expression

```
for $x in 1 to 4
```

$$s_1 \quad \texttt{return if (}\underbrace{\texttt{\$x mod 2 = 0}}_{e_1}\texttt{) then }\underbrace{\texttt{"even"}}_{e_2}\texttt{ else }\underbrace{\texttt{"odd"}}_{e_3}$$

Evaluation

iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1



Example: Evaluation of a Conditional Expression

XQuery expression

```
for $x in 1 to 4
```

$$s_1 \left[\ \texttt{return if (} \underbrace{\texttt{\$}x\texttt{ mod 2 = 0}}_{e_1} \texttt{) then } \underbrace{\texttt{"even"}}_{e_2} \texttt{ else } \underbrace{\texttt{"odd"}}_{e_3}\right.$$

Evaluation

iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

$$\underset{item}{\textcircled{$\neg$}} - \underset{item}{\sigma} - \underset{iter}{\pi} \quad \equiv \quad loop_3$$

iter
1
3

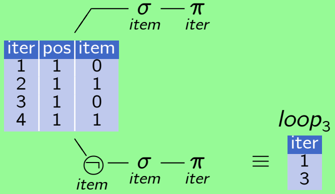
Example: Evaluation of a Conditional Expression

XQuery expression

```
for $x in 1 to 4
```

$$s_1 \quad \texttt{return if (}\underbrace{\texttt{\$}x\texttt{ mod 2 = 0}}_{e_1}\texttt{) then }\underbrace{\texttt{"even"}}_{e_2}\texttt{ else }\underbrace{\texttt{"odd"}}_{e_3}$$

Evaluation



σ_{item} — π_{iter}
$\neg_{item}$ — σ_{item} — π_{iter} $\equiv$
loop$_3$

iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

loop$_3$

iter
1
3

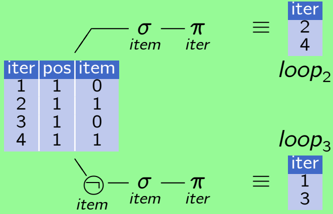
Example: Evaluation of a Conditional Expression

XQuery expression

```
for $x in 1 to 4
```

$$s_1 \; \texttt{return if (}\underbrace{\texttt{\$x mod 2 = 0}}_{e_1}\texttt{) then }\underbrace{\texttt{"even"}}_{e_2}\texttt{ else }\underbrace{\texttt{"odd"}}_{e_3}$$

Evaluation



σ item — π iter ≡
loop$_2$
loop$_3$
$\neg$ item — σ item — π iter ≡

iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

iter
2
4

loop$_2$

loop$_3$

iter
1
3

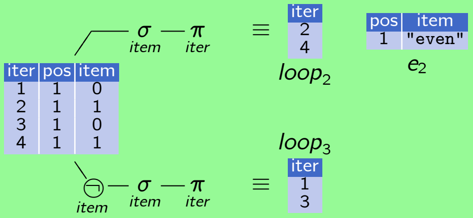
Example: Evaluation of a Conditional Expression

XQuery expression

```
for $x in 1 to 4
```

$$s_1\ \Big[\ \texttt{return if (}\underbrace{\texttt{\$}x\ \texttt{mod 2 = 0}}_{e_1}\texttt{) then }\underbrace{\texttt{"even"}}_{e_2}\texttt{ else }\underbrace{\texttt{"odd"}}_{e_3}$$

Evaluation



iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

$loop_2$

iter
2
4

e_2

pos	item
1	"even"

$loop_3$

iter
1
3

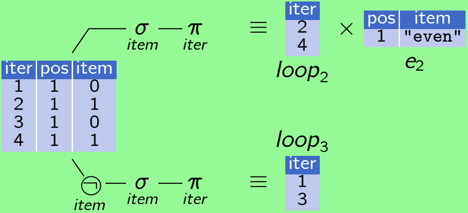
Example: Evaluation of a Conditional Expression

XQuery expression

```
for $x in 1 to 4
```

$$s_1 \;\Big[\; \texttt{return if (} \underbrace{\texttt{\$}x\texttt{ mod 2 = 0}}_{e_1} \texttt{) then } \underbrace{\texttt{"even"}}_{e_2} \texttt{ else } \underbrace{\texttt{"odd"}}_{e_3}$$

Evaluation



iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

$\sigma_{item} — \pi_{iter} \equiv$

iter
2
4

$loop_2$ $\times$

pos	item
1	"even"

e_2

$\neg_{item} — \sigma_{item} — \pi_{iter} \equiv$

$loop_3$

iter
1
3

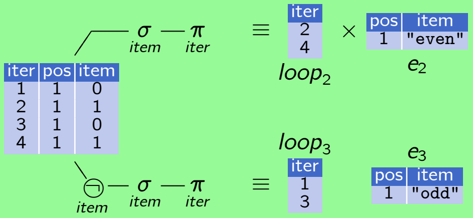
Example: Evaluation of a Conditional Expression

XQuery expression

```
for $x in 1 to 4
```

$$s_1 \quad \texttt{return if (}\underbrace{\texttt{\$}x\texttt{ mod 2 = 0}}_{e_1}\texttt{) then }\underbrace{\texttt{"even"}}_{e_2}\texttt{ else }\underbrace{\texttt{"odd"}}_{e_3}$$

Evaluation



iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

$loop_2$

iter
2
4

e_2

pos	item
1	"even"

$loop_3$

iter
1
3

e_3

pos	item
1	"odd"

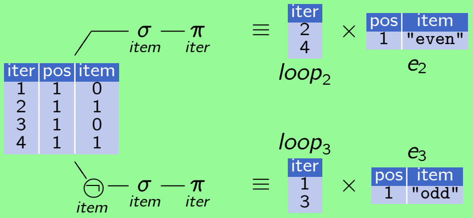
Example: Evaluation of a Conditional Expression

XQuery expression

```
for $x in 1 to 4
```

$$s_1 \; \texttt{return if (}\underbrace{\texttt{\$x mod 2 = 0}}_{e_1}\texttt{) then }\underbrace{\texttt{"even"}}_{e_2}\texttt{ else }\underbrace{\texttt{"odd"}}_{e_3}$$

Evaluation



iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

$$\pi_{iter}\,\sigma_{item} \equiv loop_2 \times e_2$$

$loop_2$:

iter
2
4

e_2:

pos	item
1	"even"

$$\pi_{iter}\,\sigma_{item}\,\neg_{item} \equiv loop_3 \times e_3$$

$loop_3$:

iter
1
3

e_3:

pos	item
1	"odd"

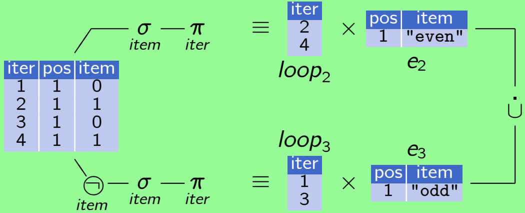
Example: Evaluation of a Conditional Expression

XQuery expression

```
for $x in 1 to 4
```

$$s_1 \left[\; \texttt{return if (}\underbrace{\texttt{\$x mod 2 = 0}}_{e_1}\texttt{) then }\underbrace{\texttt{"even"}}_{e_2}\texttt{ else }\underbrace{\texttt{"odd"}}_{e_3}\right.$$

Evaluation



iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

iter
2
4

$loop_2$

pos	item
1	"even"

e_2

$loop_3$

iter
1
3

e_3

pos	item
1	"odd"

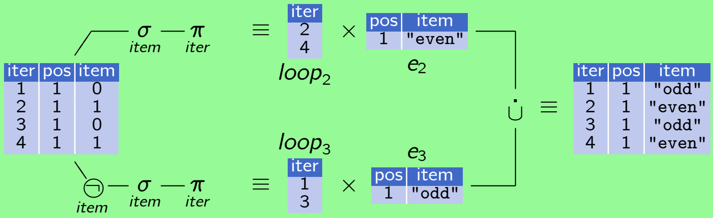
Example: Evaluation of a Conditional Expression

XQuery expression

```
for $x in 1 to 4
```

$$s_1 \; \Big[\; \texttt{return if (}\underbrace{\texttt{\$x mod 2 = 0}}_{e_1}\texttt{) then }\underbrace{\texttt{"even"}}_{e_2}\texttt{ else }\underbrace{\texttt{"odd"}}_{e_3}$$

Evaluation



iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

iter
2
4

pos	item
1	"even"

iter
1
3

pos	item
1	"odd"

iter	pos	item
1	1	"odd"
2	1	"even"
3	1	"odd"
4	1	"even"

Compiling the `where` Clause

XQuery expression

$$\left.\begin{array}{l}\texttt{for \$x in 1 to 4}\\ \texttt{where \$}x\texttt{ mod 2 = 0}\\ \texttt{return "even"}\end{array}\right] \equiv \left[\begin{array}{l}\texttt{for \$x in 1 to 4}\\ \texttt{return if \$}x\texttt{ mod 2 = 0}\\ \qquad\quad\texttt{then "even" else ()}\end{array}\right.$$

Compiling the `where` Clause

XQuery expression

```
for $x in 1 to 4
where $x mod 2 = 0
return "even"
```

≡

```
for $x in 1 to 4
return if $x mod 2 = 0
       then "even" else ()
```

Evaluation

iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

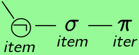
Compiling the `where` Clause

XQuery expression

```
⎡ for $x in 1 to 4      ⎤     ⎡ for $x in 1 to 4
⎢ where $x mod 2 = 0    ⎥  ≡  ⎢ return if $x mod 2 = 0
⎣ return "even"         ⎦     ⎣         then "even" else ()
```

Evaluation

iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1



Compiling the `where` Clause

XQuery expression

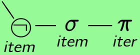
```
for $x in 1 to 4
where $x mod 2 = 0
return "even"
```

≡

```
for $x in 1 to 4
return if $x mod 2 = 0
          then "even" else ()
```

Evaluation

iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1



≡

$loop_3$

iter
1
3

Compiling the `where` Clause

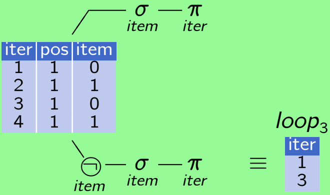
XQuery expression

```
for $x in 1 to 4
where $x mod 2 = 0
return "even"
```

≡

```
for $x in 1 to 4
return if $x mod 2 = 0
          then "even" else ()
```

Evaluation



iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

σ_{item} — π_{iter}
$\neg_{item}$ — σ_{item} — π_{iter}
≡

$loop_3$

iter
1
3

Compiling the `where` Clause

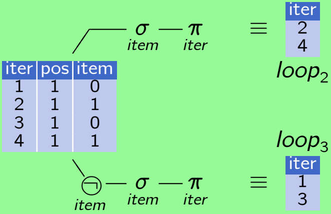
XQuery expression

```
for $x in 1 to 4
where $x mod 2 = 0
return "even"
```

≡

```
for $x in 1 to 4
return if $x mod 2 = 0
          then "even" else ()
```

Evaluation



iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

$\sigma_{item} — \pi_{iter} \equiv$

iter
2
4

loop$_2$

$\neg_{item} — \sigma_{item} — \pi_{iter} \equiv$

loop$_3$

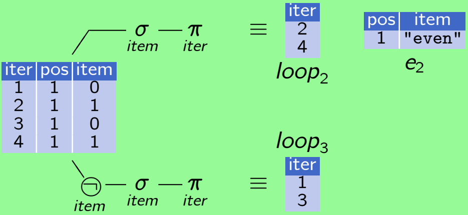
iter
1
3

Compiling the `where` Clause

XQuery expression

```
⟦ for $x in 1 to 4         ⟧      ⟦ for $x in 1 to 4              ⟧
⟦ where $x mod 2 = 0       ⟧  ≡   ⟦ return if $x mod 2 = 0        ⟧
⟦ return "even"            ⟧      ⟦        then "even" else ()    ⟧
```

Evaluation



iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

$loop_2$

iter
2
4

e_2

pos	item
1	"even"

$loop_3$

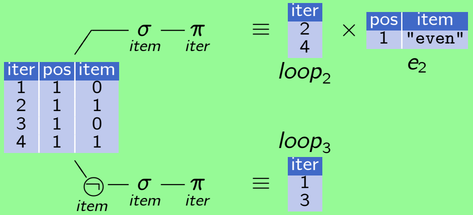
iter
1
3

Compiling the `where` Clause

XQuery expression

```
for $x in 1 to 4              ⎤     ⎡ for $x in 1 to 4
where $x mod 2 = 0            ⎥  ≡  ⎢ return if $x mod 2 = 0
return "even"                 ⎦     ⎣         then "even" else ()
```

Evaluation



iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

iter
2
4

loop$_2$

pos	item
1	"even"

e_2

loop$_3$

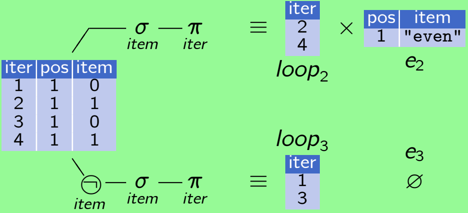
iter
1
3

Compiling the `where` Clause

XQuery expression

```
for $x in 1 to 4     ]     [ for $x in 1 to 4
where $x mod 2 = 0   ]  ≡  [ return if $x mod 2 = 0
return "even"        ]     [         then "even" else ()
```

Evaluation



iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

$\sigma_{item} — \pi_{iter} \equiv$

loop$_2$

iter
2
4

×

e_2

pos	item
1	"even"

$\neg_{item} — \sigma_{item} — \pi_{iter} \equiv$

loop$_3$

iter
1
3

e_3

$\varnothing$

Compiling the `where` Clause

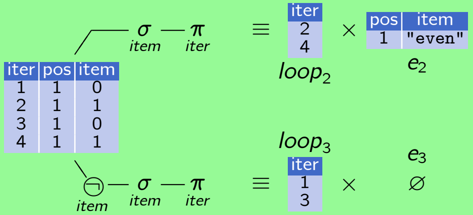
XQuery expression

```
for $x in 1 to 4
where $x mod 2 = 0
return "even"
```

≡

```
for $x in 1 to 4
return if $x mod 2 = 0
           then "even" else ()
```

Evaluation



iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

σ_{item} — π_{iter} ≡ $loop_2$ × e_2

iter
2
4

pos	item
1	"even"

$\neg_{item}$ — σ_{item} — π_{iter} ≡ $loop_3$ × e_3

iter
1
3

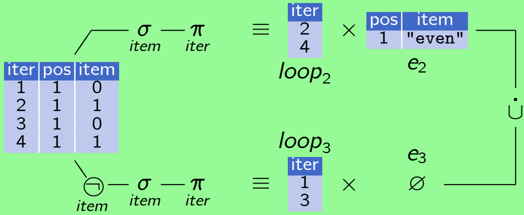
e_3: $\varnothing$

Compiling the `where` Clause

XQuery expression

```
for $x in 1 to 4     ]       [ for $x in 1 to 4
where $x mod 2 = 0   ]   ≡   [ return if $x mod 2 = 0
return "even"        ]       [          then "even" else ()
```

Evaluation



iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

$\pi_{iter}\,\sigma_{item} \equiv$ $loop_2$ × e_2

$loop_2$:

iter
2
4

e_2:

pos	item
1	"even"

$\pi_{iter}\,\sigma_{item}\,\neg_{item} \equiv$ $loop_3$ × e_3

$loop_3$:

iter
1
3

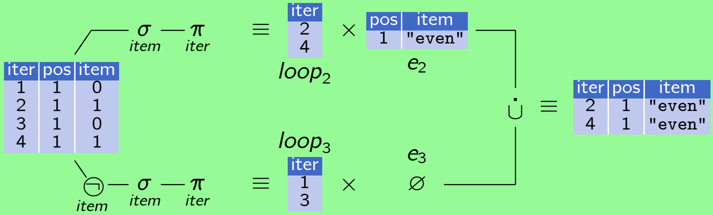
e_3: ∅

Compiling the `where` Clause

XQuery expression

```
⟦ for $x in 1 to 4          ⟧     ⟦ for $x in 1 to 4                  ⟧
⟦ where $x mod 2 = 0        ⟧  ≡  ⟦ return if $x mod 2 = 0            ⟧
⟦ return "even"             ⟧     ⟦         then "even" else ()       ⟧
```

Evaluation



σ item — π iter ≡ loop$_2$ × e$_2$
¬ item — σ item — π iter ≡ loop$_3$ × e$_3$
e$_3$: ∅
$\dot\cup$ ≡

iter	pos	item
1	1	0
2	1	1
3	1	0
4	1	1

loop$_2$

iter
2
4

e_2

pos	item
1	"even"

loop$_3$

iter
1
3

iter	pos	item
2	1	"even"
4	1	"even"

XQuery Core Optimization

- Compiler generates code which expands the FLWOR **tuple space**

⇒ **loop-invariant code motion** becomes relevant

XQuery Core Optimization

- Compiler generates code which expands the FLWOR **tuple space**

⇒ **loop-invariant code motion** becomes relevant

Q_1: Original Query

```
for $x in e1 , $y in e2
    where p($x) return e3($x, $y)
```

XQuery Core Optimization

- Compiler generates code which expands the FLWOR **tuple space**

⇒ **loop-invariant code motion** becomes relevant

Q_1: Original Query

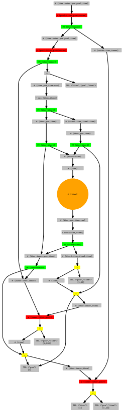
```
for $x in e1 , $y in e2
    where p($x) return e3($x, $y)
```

Q_2: Loop-invariant predicate moved

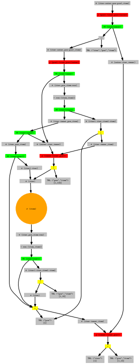
```
for $x in e1
    where p($x) return for $y in e2
                       return e3($x, $y)
```

Effect of Code Motion

Q_1:



Q_2:



Properties of Compiled Query Plans

- The target algebra as well as the compiled plans exhibit a number of nice properties:

Properties of Compiled Query Plans

- The target algebra as well as the compiled plans exhibit a number of nice properties:

Algebra/plan properties

π **no need to eliminate duplicate tuples**

Properties of Compiled Query Plans

- The target algebra as well as the compiled plans exhibit a number of nice properties:

Algebra/plan properties

π **no need to eliminate duplicate tuples**

$\dot{\cup}$ all **unions are disjoint**

Properties of Compiled Query Plans

- The target algebra as well as the compiled plans exhibit a number of nice properties:

Algebra/plan properties

π **no need to eliminate duplicate tuples**

$\dot{\cup}$ all **unions are disjoint**

$\bowtie$ all joins are **equi-joins**

Properties of Compiled Query Plans

- The target algebra as well as the compiled plans exhibit a number of nice properties:

Algebra/plan properties

π	**no need to eliminate duplicate tuples**
$\dot\cup$	all **unions are disjoint**
$\bowtie$	all joins are **equi-joins**
$\times$	one input is **singleton** (column attachment)

Properties of Compiled Query Plans

- The target algebra as well as the compiled plans exhibit a number of nice properties:

Algebra/plan properties

π	**no need to eliminate duplicate tuples**
$\dot{\cup}$	all **unions are disjoint**
$\bowtie$	all joins are **equi-joins**
$\times$	one input is **singleton** (column attachment)
	plans are **DAGs** with significant sharing

Properties of Compiled Query Plans

- The target algebra as well as the compiled plans exhibit a number of nice properties:

Algebra/plan properties

π	**no need to eliminate duplicate tuples**
$\dot{\cup}$	all **unions are disjoint**
$\bowtie$	all joins are **equi-joins**
$\times$	one input is **singleton** (column attachment)
	plans are **DAGs** with significant sharing

- Simple, "*assembly style*" operators with simple semantics

XQuery on SQL Hosts (DB2 UDB V8.1)

- Plans translate into SQL query (nesting, but **no correlation**)

XQuery on SQL Hosts (DB2 UDB V8.1)

- Plans translate into SQL query (nesting, but **no correlation**)
  - ▷ π translates into plain `SELECT` (no `DISTINCT`)
  - ▷ $\dot{\cup}$ translates into `UNION ALL`
  - ▷ ϱ (row numbering) exactly mirrors SQL:1999 **OLAP** functionality:

XQuery on SQL Hosts (DB2 UDB V8.1)

- Plans translate into SQL query (nesting, but **no correlation**)
  - ▷ π translates into plain `SELECT` (no `DISTINCT`)
  - ▷ $\dot{\cup}$ translates into `UNION ALL`
  - ▷ ϱ (row numbering) exactly mirrors SQL:1999 **OLAP** functionality:

$$\varrho_{pos:\langle item\rangle/iter}\left(\begin{array}{c|c} \text{iter} & \text{item} \\ \hline 1 & 2 \\ 1 & 6 \\ 1 & 7 \\ 2 & 3 \\ 2 & 5 \end{array}\right)$$

XQuery on SQL Hosts (DB2 UDB V8.1)

- Plans translate into SQL query (nesting, but **no correlation**)
  - ▷ π translates into plain `SELECT` (no `DISTINCT`)
  - ▷ $\dot\cup$ translates into `UNION ALL`
  - ▷ ϱ (row numbering) exactly mirrors SQL:1999 **OLAP** functionality:

$$\varrho_{pos:\langle item\rangle/iter}\left(\begin{array}{c|c} iter & item \\ \hline 1 & 2 \\ 1 & 6 \\ 1 & 7 \\ 2 & 3 \\ 2 & 5 \end{array}\right) \equiv$$

```
SELECT iter , item
       DENSE_RANK() OVER
       (PARTITION BY iter
        ORDER BY item) pos
  FROM (···)
```

XQuery on SQL Hosts (DB2 UDB V8.1)

- Plans translate into SQL query (nesting, but **no correlation**)
  - ▷ π translates into plain `SELECT` (no `DISTINCT`)
  - ▷ $\dot{\cup}$ translates into `UNION ALL`
  - ▷ ϱ (row numbering) exactly mirrors SQL:1999 **OLAP** functionality:

pos	iter	item
	1	2
	1	6
	1	7
	2	3
	2	5

≡

```
SELECT iter , item
       DENSE_RANK() OVER
       (PARTITION BY iter
        ORDER BY item) pos
  FROM (···)
```

XQuery on SQL Hosts (DB2 UDB V8.1)

- Plans translate into SQL query (nesting, but **no correlation**)
  - ▷ π translates into plain `SELECT` (no `DISTINCT`)
  - ▷ $\dot\cup$ translates into `UNION ALL`
  - ▷ ϱ (row numbering) exactly mirrors SQL:1999 **OLAP** functionality:

pos	iter	item
	1	2
	1	6
	1	7
	2	3
	2	5

$\equiv$

```
SELECT iter , item
       DENSE_RANK() OVER
       (PARTITION BY iter
        ORDER BY item) pos
  FROM (···)
```

XQuery on SQL Hosts (DB2 UDB V8.1)

- Plans translate into SQL query (nesting, but **no correlation**)
  - ▷ π translates into plain `SELECT` (no `DISTINCT`)
  - ▷ $\dot{\cup}$ translates into `UNION ALL`
  - ▷ ϱ (row numbering) exactly mirrors SQL:1999 **OLAP** functionality:

pos	iter	item
1	1	2
2	1	6
3	1	7
1	2	3
2	2	5

$\equiv$

```
SELECT iter ,item
       DENSE_RANK() OVER
       (PARTITION BY iter
        ORDER BY item) pos
  FROM (···)
```

XQuery on SQL Hosts (DB2 UDB V8.1)

- Plans translate into SQL query (nesting, but **no correlation**)
  - ▷ π translates into plain `SELECT` (no `DISTINCT`)
  - ▷ $\dot{\cup}$ translates into `UNION ALL`
  - ▷ ϱ (row numbering) exactly mirrors SQL:1999 **OLAP** functionality:

pos	iter	item
1	1	2
2	1	6
3	1	7
1	2	3
2	2	5

$\equiv$

```
SELECT iter,item
       DENSE_RANK() OVER
       (PARTITION BY iter
        ORDER BY item) pos
  FROM (···)
```

execution time [s]	1.1 MB	110 MB	1.1 GB
XMark Q1	< 0.01	< 0.01	< 0.01
XMark Q6	0.01	0.18	1.7
XMark Q7	0.01	0.52	5.3

XQuery on SQL Hosts

XMark Query Q8

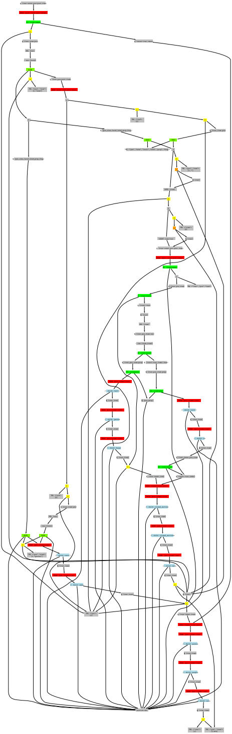
```
for $p in fn:doc("auction.xml")/site/people/person
return let $a := for $t in fn:doc("auction.xml")/site/
                                 closed_auctions/closed_auction
                 return if fn:data($t/buyer/person/text()) =
                           fn:data($p/id/text())
                        then $t else ()
       return <item> { <person> { $p/name/text() } </person>,
                       text { fn:count($a) } } </item>
```

XQuery on SQL Hosts

XMark Query Q8

```
for $p in fn:doc("auction.xml")/site/people/person
return let $a := for $t in fn:doc("auction.xml")/site/
                                 closed_auctions/closed_auction
                 return if fn:data($t/buyer/person/text()) =
                           fn:data($p/id/text())
                        then $t else ()
       return <item> { <person> { $p/name/text() } </person>,
                       text { fn:count($a) } } </item>
```

- Compiles into DAG of 120 algebraic operators, significant sharing



XQuery on SQL Hosts

XMark Query Q8

```
for $p in fn:doc("auction.xml")/site/people/person
return let $a := for $t in fn:doc("auction.xml")/site/
                                 closed_auctions/closed_auction
                 return if fn:data($t/buyer/person/text()) =
                           fn:data($p/id/text())
                        then $t else ()
       return <item> { <person> { $p/name/text() } </person>,
                       text { fn:count($a) } } </item>
```

- Compiles into DAG of 120 algebraic operators, significant sharing

XQuery on SQL Hosts

XMark Query Q8

```
for $p in fn:doc("auction.xml")/site/people/person
return let $a := for $t in fn:doc("auction.xml")/site/
                                 closed_auctions/closed_auction
                 return if fn:data($t/buyer/person/text()) =
                           fn:data($p/id/text())
                        then $t else ()
       return <item> { <person> { $p/name/text() } </person>,
                       text { fn:count($a) } } </item>
```

- Compiles into DAG of 120 algebraic operators, significant sharing
  - ▷ Equivalent tree has ≈ 2000 nodes

XQuery on SQL Hosts

XMark Query Q8

```
for $p in fn:doc("auction.xml")/site/people/person
return let $a := for $t in fn:doc("auction.xml")/site/
                                 closed_auctions/closed_auction
                 return if fn:data($t/buyer/person/text()) =
                           fn:data($p/id/text())
                        then $t else ()
       return <item> { <person> { $p/name/text() } </person>,
                       text { fn:count($a) } } </item>
```

- Compiles into DAG of 120 algebraic operators, significant sharing
  - ▷ Equivalent tree has ≈ 2000 nodes
- **NB:** No optimizations applied yet (neither XQuery nor algebraic)

Static Analysis: Live Nodes

Nested Element Construction

```
( <even-or-odd>
    { for $x in (1,2,3,4)
      return if ($x mod 2 = 0)
             then <even/>
             else <odd/>
    }
  </even-or-odd>
 )/child::even
```

Static Analysis: Live Nodes

Nested Element Construction

```
( <even-or-odd>
    { for $x in (1,2,3,4)
      return if ($x mod 2 = 0)
             then <even/>  ①
             else <odd/>
    }
  </even-or-odd>
 )/child::even
```

Static Analysis: Live Nodes

Nested Element Construction

```
( <even-or-odd>
    { for $x in (1,2,3,4)
      return if ($x mod 2 = 0)
             then <even/> •——— ①
             else <odd/>
    }
  </even-or-odd>
 )/child::even
```

Live Nodes at Point ⓧ

① `<even/>` (loop-lifted)

Static Analysis: Live Nodes

Nested Element Construction

```
( <even-or-odd>
    { for $x in (1,2,3,4)
      return if ($x mod 2 = 0)
             then <even/>  ①
             else <odd/>   ②
    }
  </even-or-odd>
 )/child::even
```

Live Nodes at Point ⓧ

① `<even/>` (loop-lifted)

Static Analysis: Live Nodes

Nested Element Construction

```
( <even-or-odd>
    { for $x in (1,2,3,4)
      return if ($x mod 2 = 0)
             then <even/>          ①
             else <odd/>           ②
    }
  </even-or-odd>
 )/child::even
```

Live Nodes at Point ⓧ

①	`<even/>`	(loop-lifted)
②	`<odd/>`	(loop-lifted)

Static Analysis: Live Nodes

Nested Element Construction

```
( <even-or-odd>
    { for $x in (1,2,3,4)
      return if ($x mod 2 = 0)
             then <even/>          ①
             else <odd/>           ②
③   }
  </even-or-odd>
 )/child::even
```

Live Nodes at Point ⓧ

①	`<even/>`	(loop-lifted)
②	`<odd/>`	(loop-lifted)

Static Analysis: Live Nodes

Nested Element Construction

```
( <even-or-odd>
    { for $x in (1,2,3,4)
      return if ($x mod 2 = 0)
             then <even/>                ①
             else <odd/>                 ②
  ③ }
  </even-or-odd>
 )/child::even
```

Live Nodes at Point ⓧ

①	`<even/>`	(loop-lifted)
②	`<odd/>`	(loop-lifted)
③	① $\dot\cup$ ②	

Static Analysis: Live Nodes

Nested Element Construction

```
(• <even-or-odd>
   { for $x in (1,2,3,4)
     return •if ($x mod 2 = 0)
             then <even/> •           ①
④            else <odd/> •            ②
    ③ }
   </even-or-odd>
 )/child::even
```

Live Nodes at Point ⓧ

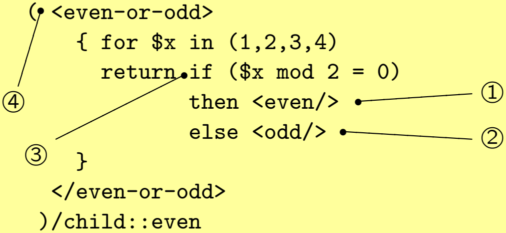
① `<even/>` (loop-lifted)

② `<odd/>` (loop-lifted)

③ ① $\dot\cup$ ②

Static Analysis: Live Nodes

Nested Element Construction



```
(<even-or-odd>
   { for $x in (1,2,3,4)
     return if ($x mod 2 = 0)
            then <even/>
            else <odd/>
   }
  </even-or-odd>
 )/child::even
```

Live Nodes at Point ⓧ

① `<even/>` (loop-lifted)

② `<odd/>` (loop-lifted)

③ ① $\dot{\cup}$ ②

④ `<even-or-odd>` ③ `</even-or-odd>`

Live Node Inference

Inference rule `fn:doc(`e`)`

$$\frac{\Gamma; loop \vdash e \Mapsto (\texttt{"uri"}, lv)}{\Gamma; loop \vdash \texttt{fn:doc(}e\texttt{)} \Mapsto \left(loop \times \begin{array}{|c|c|}\hline \text{pos} & \text{item} \\ \hline 1 & root(\texttt{uri}) \\ \hline\end{array}, extn(\texttt{uri})\right)}$$

Inference rule `if` $\cdots$ `then` $\cdots$ `else`

$$\frac{\begin{array}{c} \Gamma; loop \vdash e_1 \Mapsto (q_1, lv_1) \\ loop_2 \equiv \pi_{iter}(\sigma_{item}(q_1)) \qquad loop_3 \equiv \pi_{iter}(\sigma_{item}(\ominus_{item}(q_1))) \\ \Gamma; loop_2 \vdash e_2 \Mapsto (q_2, lv_2) \qquad \Gamma; loop_3 \vdash e_3 \Mapsto (q_3, lv_3) \end{array}}{\Gamma; loop \vdash \texttt{if (}e_1\texttt{) then } e_2 \texttt{ else } e_3 \Mapsto (q_2 \dot{\cup} q_3, lv_2 \dot{\cup} lv_3)}$$

Live Node Inference

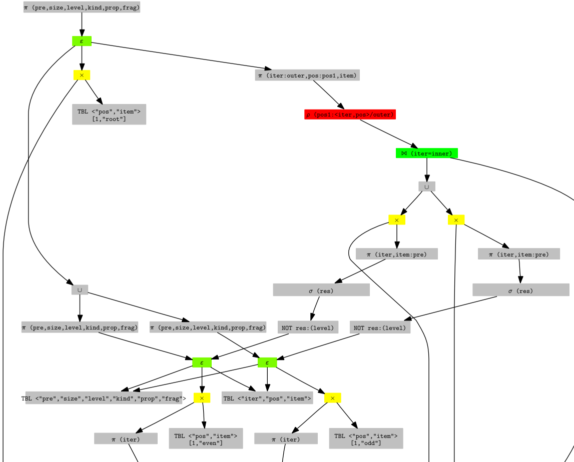
Inference rule `element {`e_1`} {`e_2`}`

$$\frac{\Gamma; loop \vdash e_1 \Mapsto (q_1, lv_1) \qquad \Gamma; loop \vdash e_2 \Mapsto (q_2, lv_2) \\ n \equiv \varepsilon\ lv_2\ q_1\ q_2}{\Gamma; loop \vdash \texttt{element}\ \texttt{\{}e_1\texttt{\}}\ \texttt{\{}e_2\texttt{\}} \Mapsto \left(\pi_{iter,item}(roots(n)) \times \begin{array}{|c|}\hline \text{pos} \\ \hline 1 \\ \hline \end{array}, n\right)}$$

Inference rule e/s (XPath location step)

$$\frac{\Gamma; loop \vdash e \Mapsto (q, lv)}{\Gamma; loop \vdash e/s \Mapsto \left(\underset{pos:\langle item\rangle / iter}{\varrho} (q \mathbin{\text{⊿}}_s\ lv), lv\right)}$$

Live Node Computation


π (pre,size,level,kind,prop,frag)
ε
×
TBL <"pos","item">
[1,"root"]
π (iter:outer,pos:pos1,item)
ρ (pos1:<iter,pos>/outer)
⋈ (iter=inner)
∪
×
×
π (iter,item:pre)
π (iter,item:pre)
σ (res)
σ (res)
∪
π (pre,size,level,kind,prop,frag)
π (pre,size,level,kind,prop,frag)
NOT res:(level)
NOT res:(level)
ε
ε
TBL <"pre","size","level","kind","prop","frag">
×
TBL <"iter","pos","item">
×
π (iter)
TBL <"pos","item">
[1,"even"]
π (iter)
TBL <"pos","item">
[1,"odd"]

Work in Flux

Optimizations

- Exploit **column properties**: *unique*, *constant*, *dense*

Work in Flux

Optimizations

- Exploit **column properties**: *unique*, *constant*, *dense*
- **Order awareness** [ICDE 2004, SIGMOD 1996]

Work in Flux

Optimizations

- Exploit **column properties**: *unique*, *constant*, *dense*
- **Order awareness** [ICDE 2004, SIGMOD 1996]
- Exploit **disjointness** of intermediate results

Work in Flux

Optimizations

- Exploit **column properties**: *unique*, *constant*, *dense*
- **Order awareness** [ICDE 2004, SIGMOD 1996]
- Exploit **disjointness** of intermediate results
- "**Live node analysis**": evaluate ⎍ over minimal tree fragments

Work in Flux

Optimizations

- Exploit **column properties**: *unique*, *constant*, *dense*
- **Order awareness** [ICDE 2004, SIGMOD 1996]
- Exploit **disjointness** of intermediate results
- "**Live node analysis**": evaluate ⎍ over minimal tree fragments

New implementation

- Main-memory DBMS kernel **MonetDB** (CWI, Amsterdam)

Work in Flux

Optimizations

- Exploit **column properties**: *unique*, *constant*, *dense*
- **Order awareness** [ICDE 2004, SIGMOD 1996]
- Exploit **disjointness** of intermediate results
- "**Live node analysis**": evaluate ⎍ over minimal tree fragments

New implementation

- Main-memory DBMS kernel **MonetDB** (CWI, Amsterdam)
- Target language is MIL, algebra over binary tables

Work in Flux

Optimizations

- Exploit **column properties**: *unique*, *constant*, *dense*
- **Order awareness** [ICDE 2004, SIGMOD 1996]
- Exploit **disjointness** of intermediate results
- "**Live node analysis**": evaluate ⎍ over minimal tree fragments

New implementation

- Main-memory DBMS kernel **MonetDB** (CWI, Amsterdam)
- Target language is MIL, algebra over binary tables
- Ordered data model (ϱ may largely become obsolete)