ADT 2010

MonetDB/XQuery (2/2): High-Performance, Purely Relational XQuery Processing

http://pathfinder-xquery.org/

http://monetdb-xquery.org/

Stefan Manegold

Stefan.Manegold@cwi.nl

http://www.cwi.nl/~manegold/

XPath evaluation (SQL)

Example query:

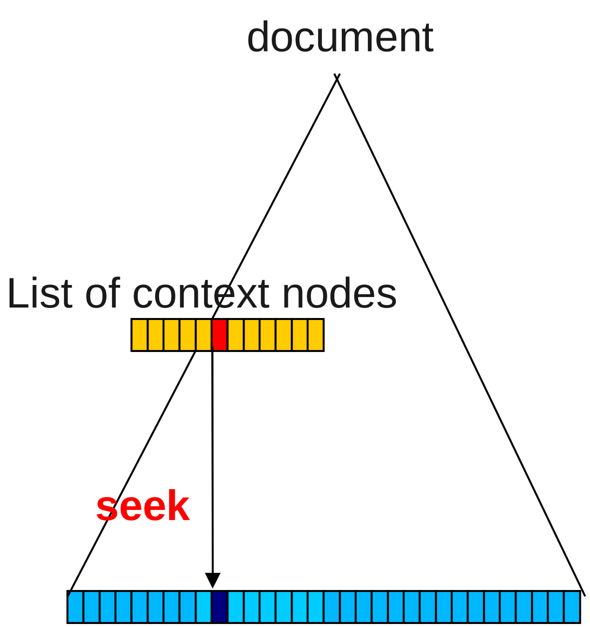
/descendant::open_auction[./bidder]/annotation

```
SELECT DISTINCT a.pre
  FROM doc r, doc oa, doc b, doc a
 WHERE r.pre=0
   AND oa.pre > r.pre AND oa.post < r.post        <- descendant
   AND oa.name = “open_auction” AND oa.kind = “elem”
   AND b.pre > oa.pre AND b.post < oa.post        } child
   AND b.level = oa.level + 1
   AND b.name = “bidder” AND b.kind < “elem”
   AND a.pre > oa.pre AND a.post < oa.post        } child
   AND a.level = oa.level + 1
   AND a.name = “annotation” AND a.kind = “elem”
ORDER BY a.pre
```

- (potentially?) expensive joins due to range predicates
- (potentially?) expensive duplicate elimination

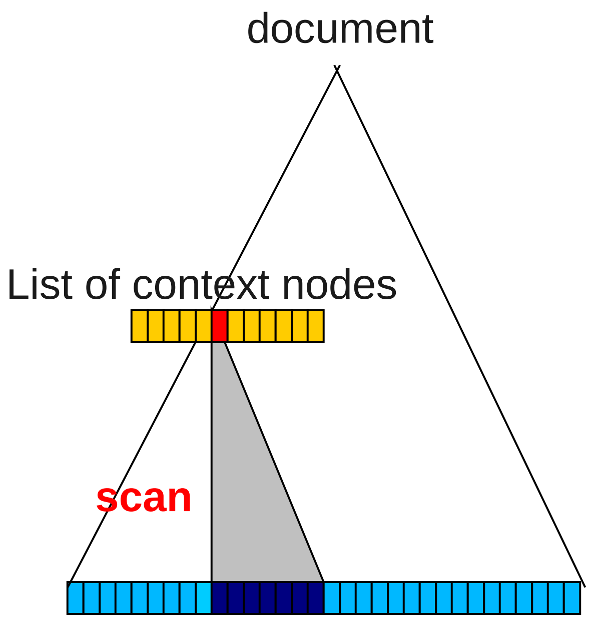
Staircase Join [VLDB03]

`pre|post` are not random numbers:
`=>` exploit the tree properties encoded in them



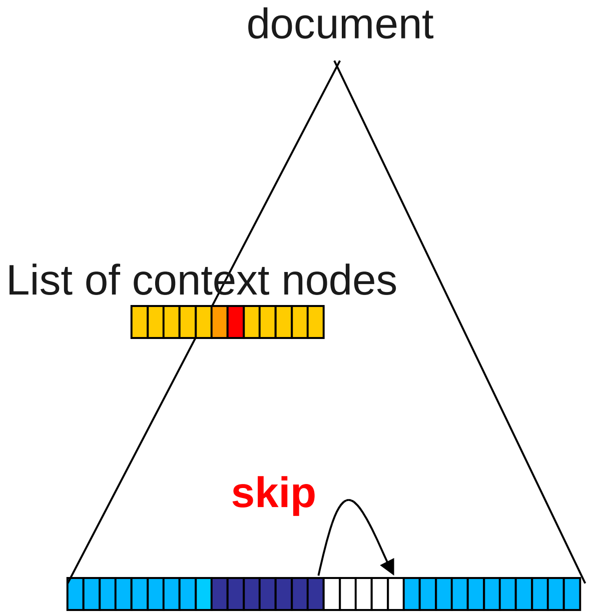
Staircase Join [VLDB03]

`pre|post` are not random numbers:
=> exploit the tree properties encoded in them



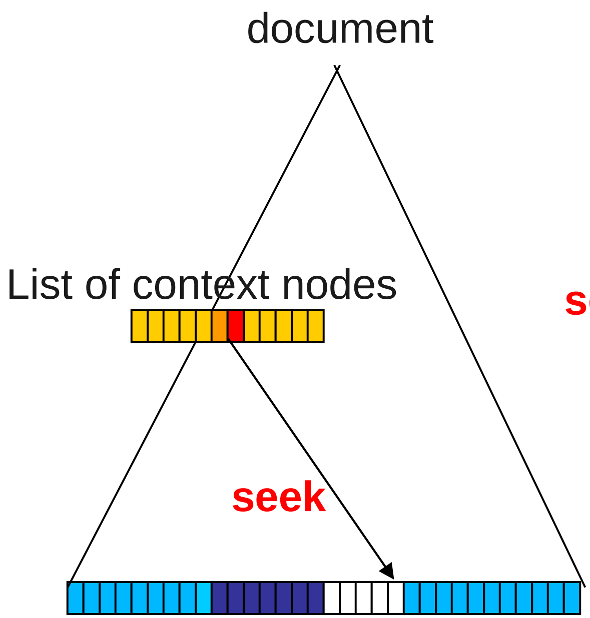
Staircase Join [VLDB03]

`pre|post` are not random numbers:
=> exploit the tree properties encoded in them



Staircase Join [VLDB03]

`pre|post` are not random numbers:
`=>` exploit the tree properties encoded in them

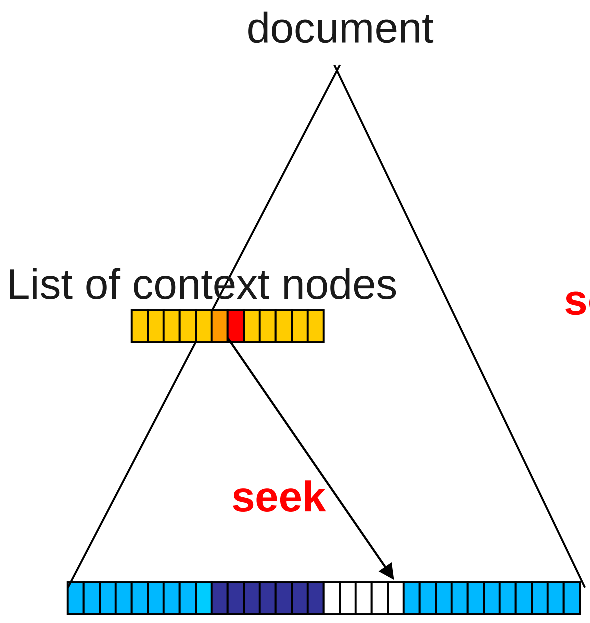


seek ➔ scan ➔ skip ➔seek➔ scan ➔ skip ➔...

Staircase Join [VLDB03]

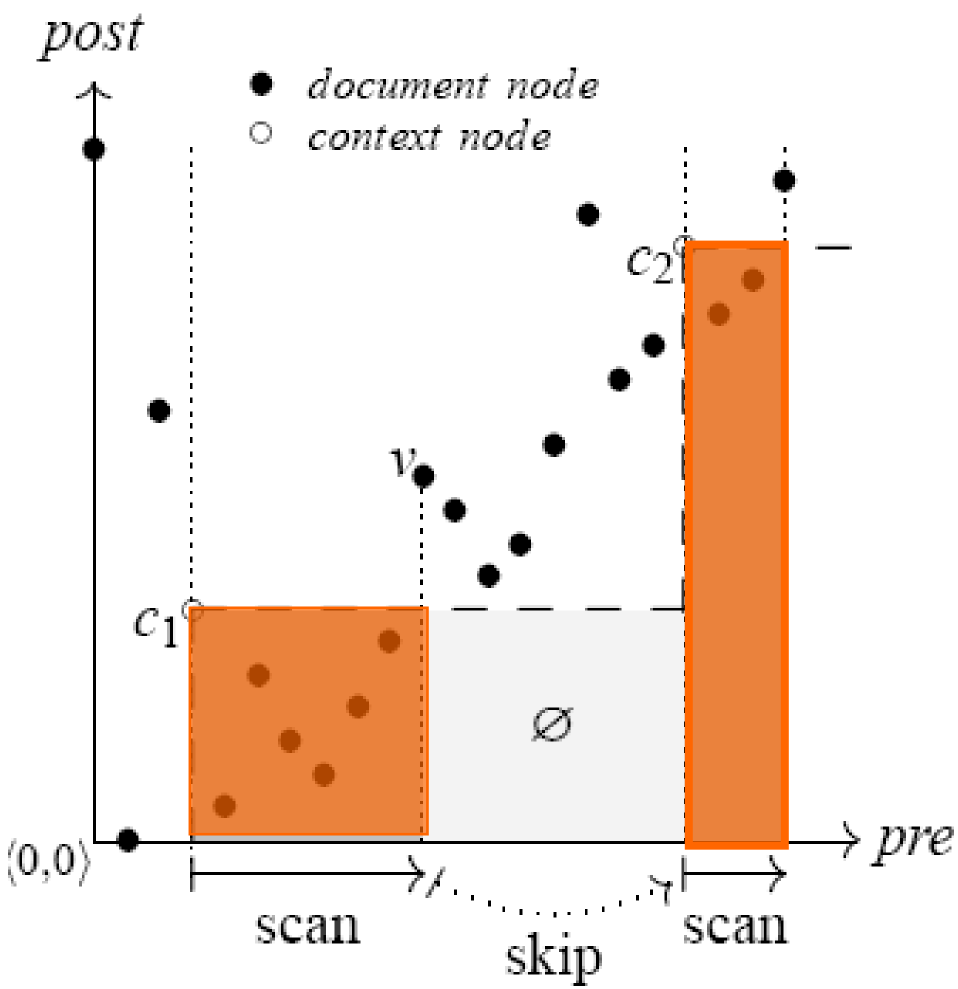
- **skipping**: avoid touching node ranges that cannot contain results

Generate a duplicate-free result in document order
- **pruning**: reduce the context set a-priori
- **partitioning**: single sequential pass over the document



seek ➔ scan ➔ skip ➔seek➔ scan ➔ skip ➔...

Staircase Join: Skipping



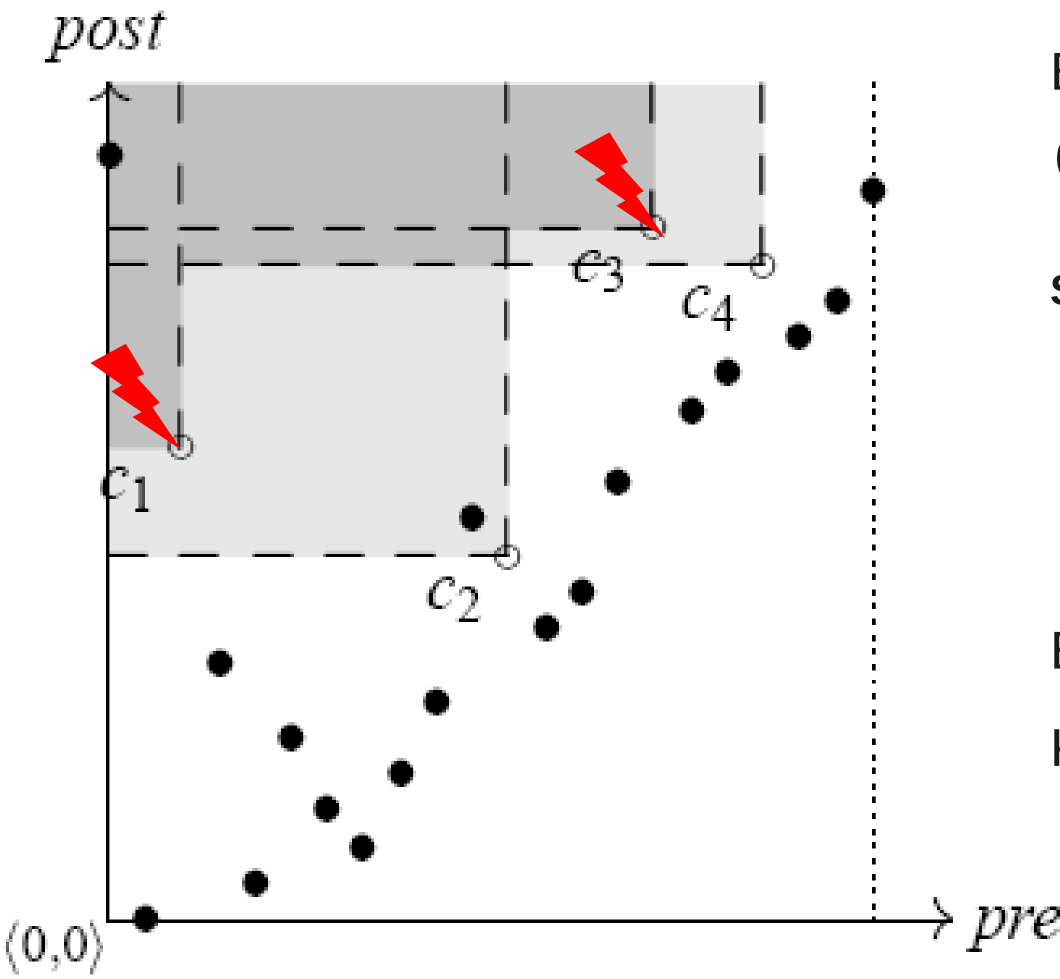
Example:

`(c1,c2)/descendant:*`

SELECT DISTINCT doc.pre
FROM c, doc
WHERE doc.pre > c.pre
AND doc.post < c.post

Avoid comparing large chunks of the document table.

Staircase Join: Pruning



Example:

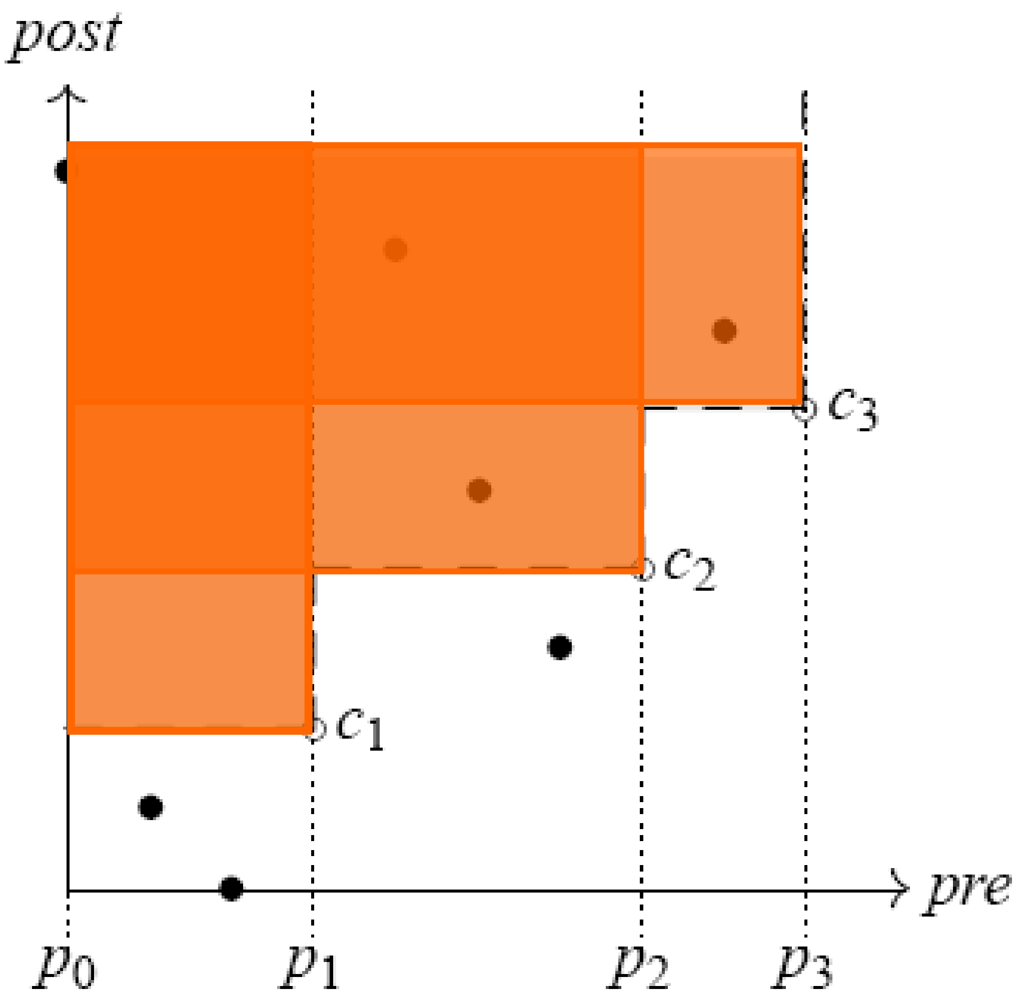
`(c1,c2,c3,c4)/ancestor:*`

SELECT DISTINCT doc.pre
FROM c, doc
WHERE doc.pre < c.pre
AND doc.post > c.post

Eliminate: *c1, c3*

Keep: *c2, c4*

Staircase Join: Partitioning



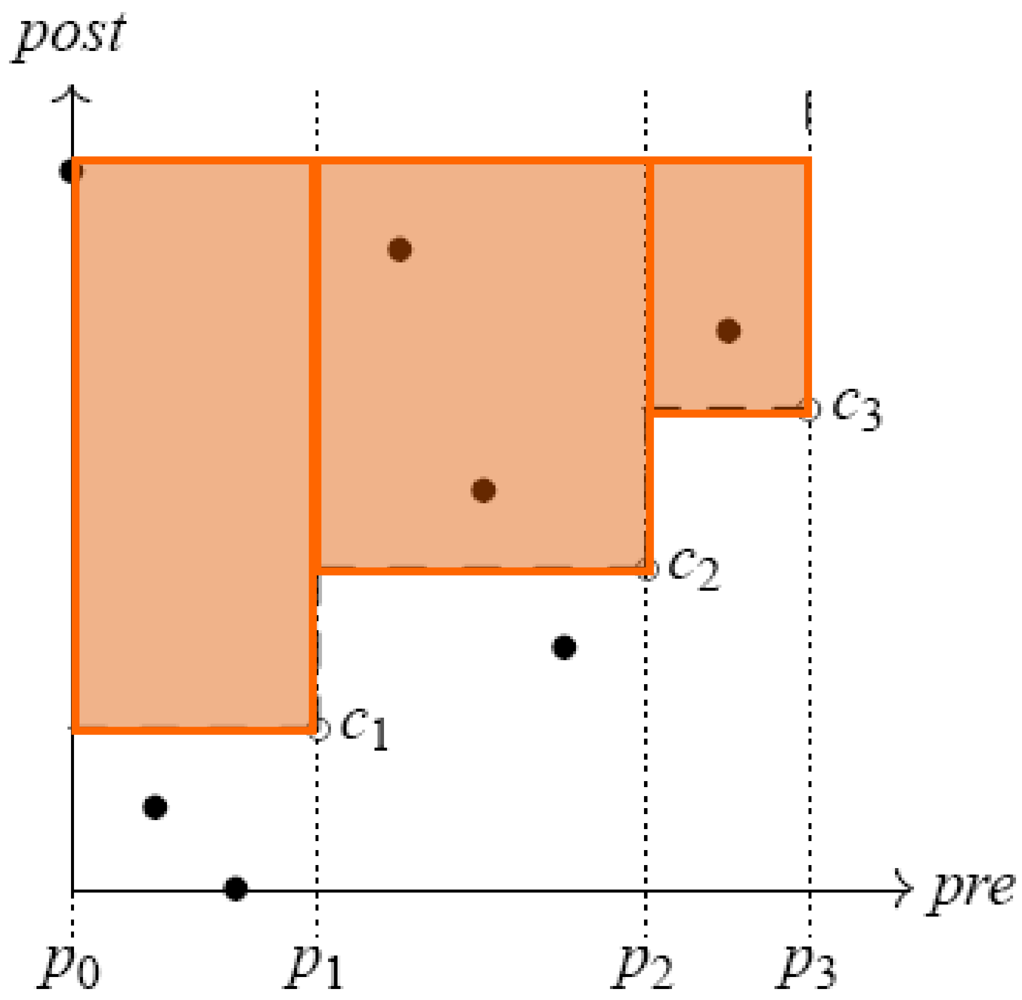
Example:

(c1,c2,c3)/ancestor:*

SELECT DISTINCT doc.pre
FROM c, doc
WHERE doc.pre < c.pre
AND doc.post > c.post

Single-pass algorithm that avoids generating duplicates

Staircase Join: Partitioning



Example:

`(c1,c2,c3)/ancestor:*`

SELECT **DISTINCT** doc.pre
FROM c, doc
WHERE doc.pre < c.pre
AND doc.post > c.post

Single-pass algorithm that avoids generating duplicates

Schedule

- So far:
  - RDBMS back-end support for XML/XQuery (1/2):
    - Document Representation (*XPath Accelerator*, *Pre/Post plane*)
    - XPath navigation (*Staircase Join*)

Schedule

- So far
  - RDBMS back-end support for XML/XQuery (1/2):
    - Document Representation (*XPath Accelerator*, *Pre/Post plane*)
    - XPath navigation (*Staircase Join*)

- Now:
  - XQuery to Relational Algebra Compiler:
    - Item- & Sequence- Representation
    - Efficient FLWoR Evaluation (*Loop-Lifting*)
    - Optimization
  - RDBMS back-end support for XML/XQuery (2/2):
    - Updateable Document Representation

Source Language: XQuery Core

XQuery is a lot more than just XPath.

literals	`42`, `"foo"`, `()`, ...
arithmetics	e_1 `+` e_2, e_1 `-` e_2, ...
builtin functions	`fn:sum(`e`)`, `fn:count(`e`)`, `fn:doc(`uri`)`
variable bindings	`let $`v `:=` e_1 `return` e_2
iteration	`for $`v `at $`p `in` e_1 `return` e_2
conditionals	`if` p `then` e_1 `else` e_2
sequence construction	e_1 `,` e_2
function calls	f `(`e_1`,` e_2`,` `...,` e_n`)`
element construction	`element` e_1 `{` e_2 `}`
XPath steps	$e/\alpha::\nu$
$\vdots$	$\vdots$

Target Language: Relational Algebra

Operators

σ_a	row selection
$\pi_{a,b:c}$	projection/renaming
$\varrho_{a:(b,..,c)/d}$	**row numbering**
$\_ \times \_$	Cartesian product
$\_ \bowtie_p \_$	join
$\_ \dot{\cup} \_$	disjoint union
$\_ \setminus \_$	difference
δ	duplicate elimination
$\circledcirc_{a:(b,..,c)}$	apply $\circ \in \{*, =, <, ..\}$

a	b
8	'e'
5	'f'
2	'o'
6	's'
3	't'
9	'n'

$-\varrho_{c:(a)}\rightarrow$

a	b	c
8	'e'	5
5	'f'	3
2	'o'	1
6	's'	4
3	't'	2
9	'n'	6

- RDBMS kernels implement ϱ in terms of SQL's `DENSE_RANK`.
- Most conceivable implementations of ϱ require a sorted input.

Sequence Representation

$(i_1, i_2, \ldots, i_n)$

pos	item
1	i_1
2	i_2
$\vdots$	$\vdots$
n	i_n

i

pos	item
1	i

$()$

pos	item

- *sequence = table of items*
- *add pos column for maintaining order*

Sequence Representation

- Item sequences, sequence order

(10, “x”, <a/>, 10) →

	Item	
Pos	**Node**	**Atom**
1	NULL	10
2	NULL	“x”
3	pre(a)	NULL
4	NULL	10

→

Pos	Item
1	10
2	“X”
3	pre(a)
4	10

- Problems:
  - Polymorphic columns
  - Redundant storage
  - Copy overhead (especially with strings)

Item Representation

- Item sequences, sequence order

(10, “x”, <a/>, 10) →

Item

Pos	Node	Atom
1	NULL	10
2	NULL	“x”
3	pre(a)	NULL
4	NULL	10

→

Pos	Item
1	10
2	“X”
3	pre(a)
4	10

Pos	Item
1	10
2	“X”
3	pre(a)
4	10

→

Pos	Item	Kind
1	1@0	Int
2	1@0	Str
3	0@0	Node
4	1@0	Int

str_values	
1@0	“x”

int_values	
1@0	10

Iterations

- 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:

Iterations

- 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

Iterations

- 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:

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

Iterations

- 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:

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

Iterations

- 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
⋮	⋮
n	x_n

- Derive $\$v$ as follows:

iter	pos	item
1	1	x_1
2	1	x_2
⋮	⋮	⋮
n	1	x_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:

$loop(s_1)$ $\times$

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 $v in (`$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"

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:

$loop(s_1) \times$

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"

Deriving *loop*

XQuery `FLWOR` expressions define a new *loop* relation.

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

- How can we derive *loop* given this XQuery expression?

- $(x_1, x_2, \ldots, x_n)$

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

- Derive $\$v$:

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

- $loop(s_1)$

iter
1
2
$\vdots$
n

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 $\$v_0$, $\$v_1$

$\$v_0$ in s_1:

iter	pos	item
1	1	10
2	1	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

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

$loop(s_0)$

iter
1

$loop(s_1)$

iter
1
2

$loop(s_2)$

iter
1
2
3
4

- Derive $\$v_0$, $\$v_1$

$\$v_0$ 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

Loop-lifting

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

Full Example

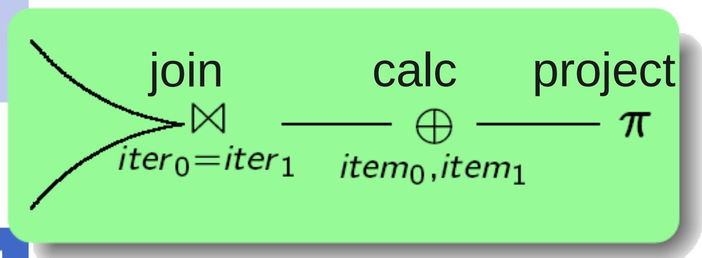
```
for $v0 in (10,20)
   for $v1 in (100,200)
s2 [   return $v0 + $v1
```

$\$v_0$

$iter_0$	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



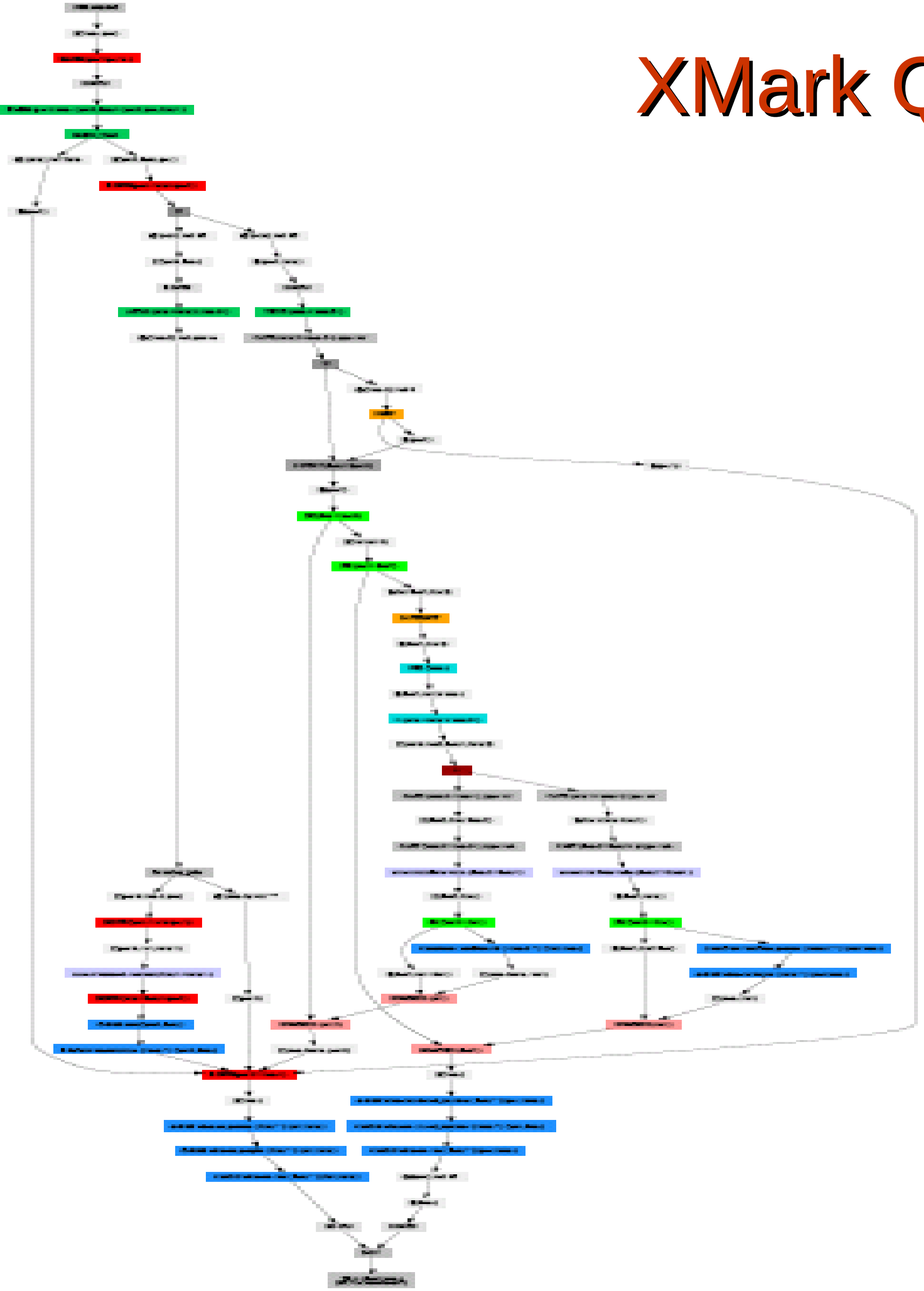
=

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

XQuery On SQL Hosts [VLDB04]

XQuery Construct	Relational Mapping
sequence construction	A union B
if-then-else	select(A=true,B) union select(A=false,C)
for-loops	cartesian product
calculations	project(A,x=expr(Y1,..Yn))
list functions, e.g. `fn:first()`	select(A,pos=1)
element construction	updates in temporary tables
XPath steps	staircase-join

XMark Query 8



```
let $auction := doc("auctions.xml")
return
 for $p in
  $auction/site/people/person
 let $a :=
  for $t in $auction/site/
   closed_auctions/closed_auction
  where $t/buyer/@person = $p/@id
  return $t
return
 <item person="{$p/name/text()}">
  {count($a)}
 </item>
```

Peephole Optimization
[Grust, XIME-P 2005]

Input: XQuery

```
for $x in (k,...,2,1)
  return $x * 5
```

Output: Relational Algebra

iter	pos	item
1	1	$k*5$
⋮	⋮	⋮
$k-1$	1	10
k	1	5

$\pi_{\text{iter,pos,item:res}}$

$\circledast_{\text{res:(item,item1)}}$

$\bowtie_{\text{iter=iter1}}$

iter	pos	item
1	1	k
⋮	⋮	⋮
$k-1$	1	2
k	1	1

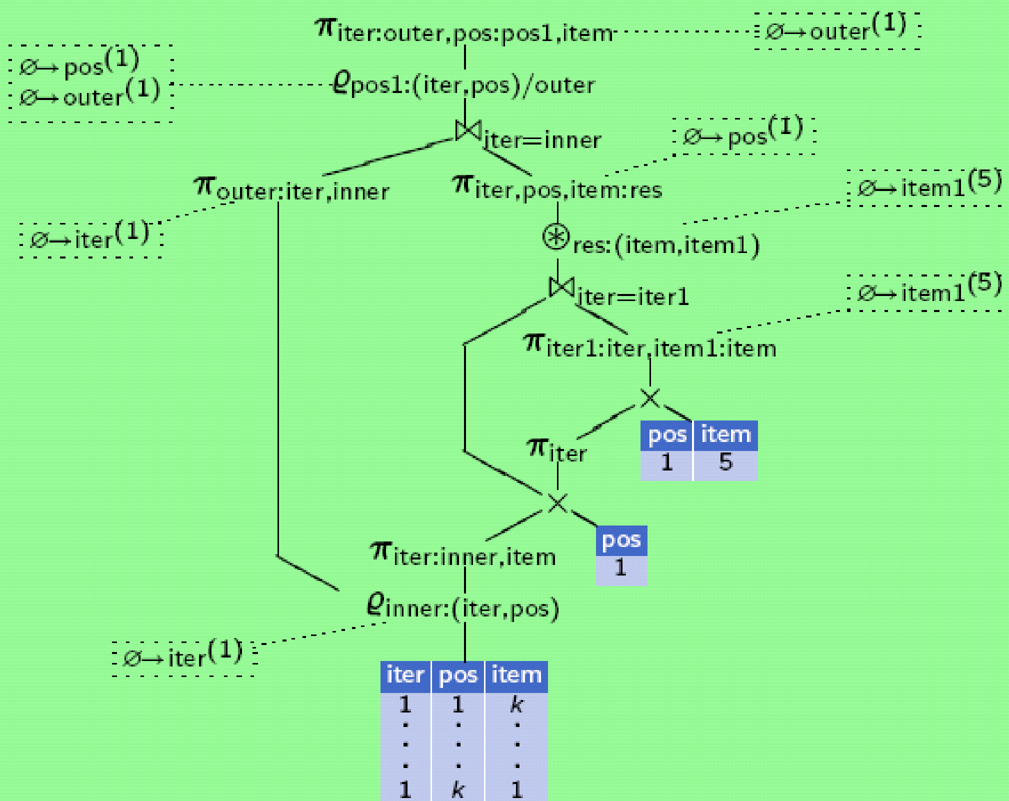
iter	pos	item
1	1	5
⋮	⋮	⋮
$k-1$	1	5
k	1	5

Peephole Optimization
[Grust, XIME-P 2005]

Input: XQuery

```
for $x in (k,...,2,1)
  return $x * 5
```

Plan Property: Constant Columns

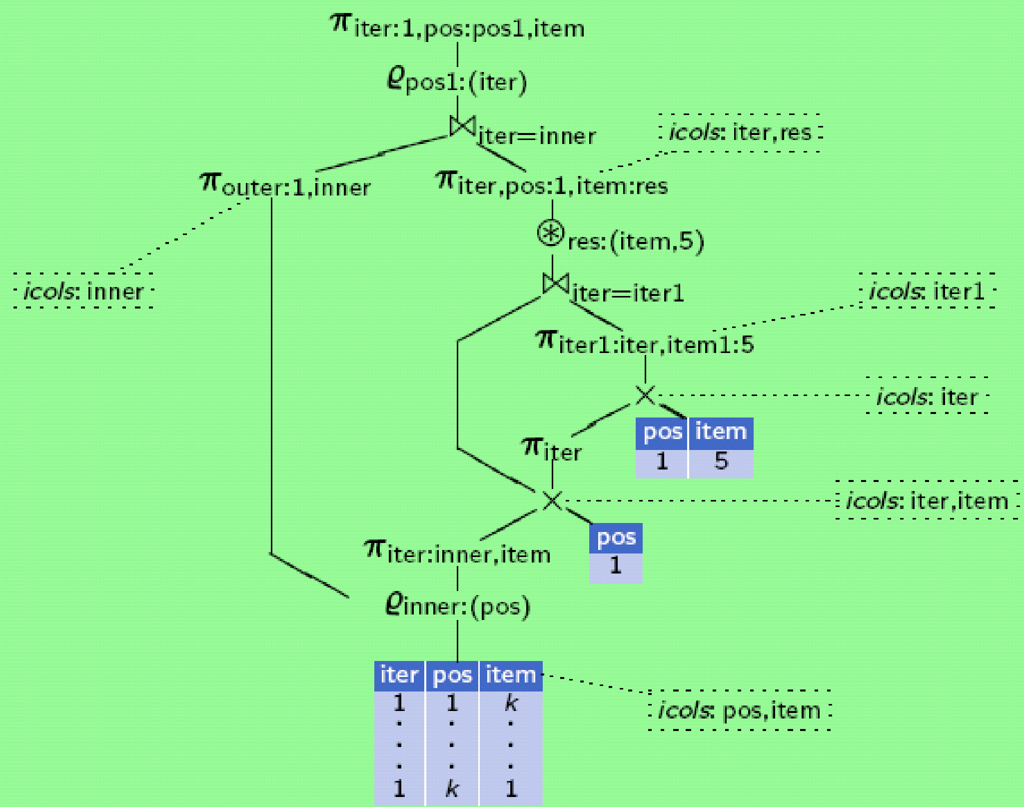


Peephole Optimization
[Grust, XIME-P 2005]

Input: XQuery

```
for $x in (k,...,2,1)
    return $x * 5
```

Plan Property: Strictly Required Columns

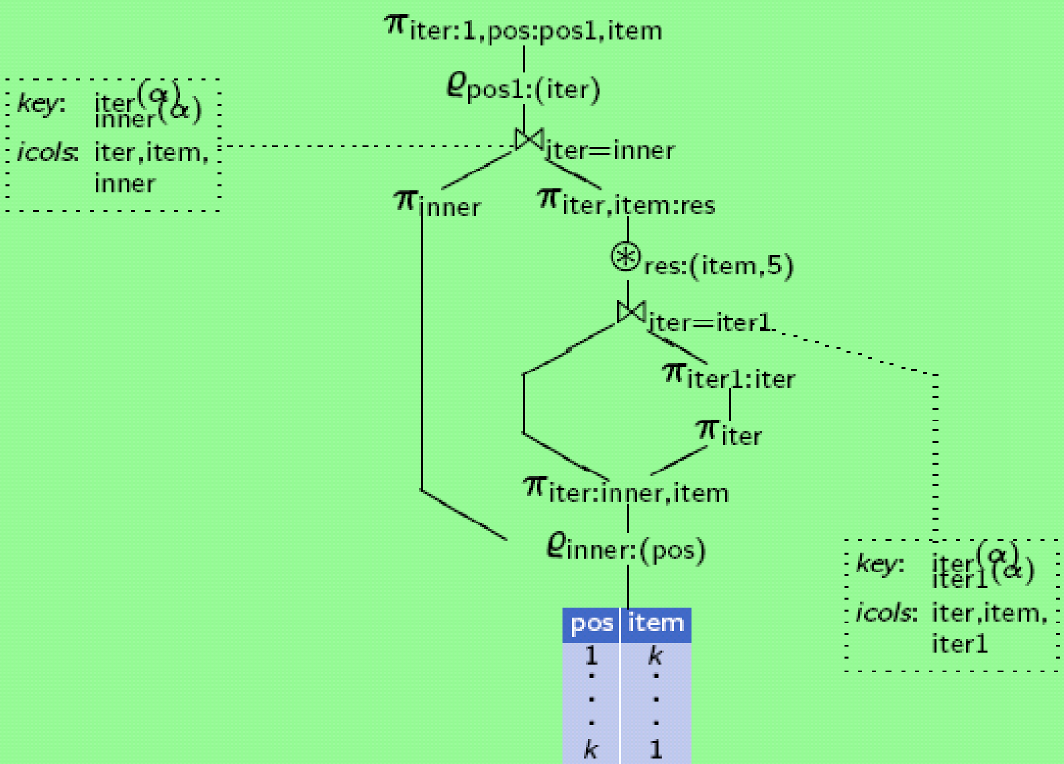


Peephole Optimization
[Grust, XIME-P 2005]

Input: XQuery

```
for $x in (k,...,2,1)
  return $x * 5
```

Plan Property: Key Candidate Columns



Peephole Optimization
[Grust, XIME-P 2005]

Input: XQuery

```
for $x in (k,...,2,1)
  return $x * 5
```

Plan Property: Dense Columns

$\pi_{\text{iter:1,pos:pos1,item}}$

$\varrho_{\text{pos1:(iter)}}$ — *dense*: iter

$\pi_{\text{iter,item,inner:iter}}$

$\pi_{\text{iter,item:res}}$

$\circledast_{\text{res:(item,5)}}$

$\pi_{\text{iter,item,iter1:iter}}$

$\pi_{\text{iter:inner,item}}$

$\varrho_{\text{inner:(pos)}}$ — *dense*: pos

pos	item
1	k
⋮	⋮
k	1

Peephole Optimization
[Grust, XIME-P 2005]

Input: XQuery

```
for $x in (k,...,2,1)
  return $x * 5
```

Final Plan

$$\pi_{\text{iter:1,pos,item:res}}$$

$$\circledast_{\text{res:(item,5)}}$$

pos	item
1	k
⋮	⋮
k	1

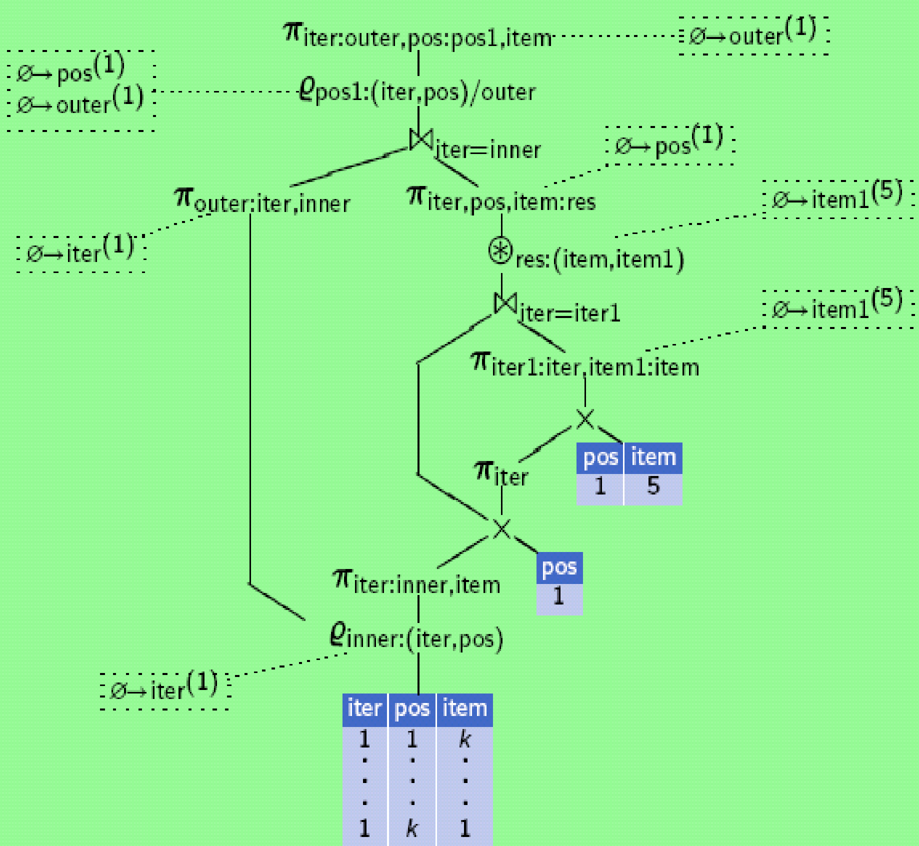
Peephole Optimization
[Grust, XIME-P 2005]

Input: XQuery

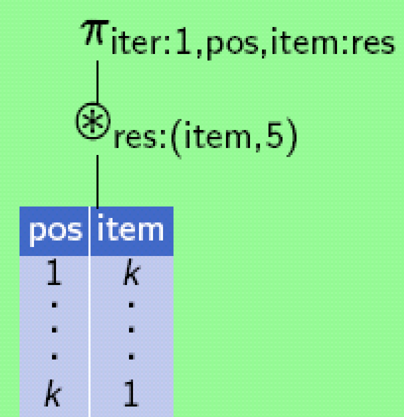
```
for $x in (k,...,2,1)
  return $x * 5
```

- **Plan simplification**

Plan Property: Constant Columns



Final Plan



Peephole Optimization
[Boncz et al., SIGMOD 2006]

- **Sort Avoidance**

generated by `DENSE RANK` **`pos`** `ORDER BY` **`pos`** `PARTITION BY` **`iter`**

by tracking secondary ordering properties **`[pos|iter]`**, [Wang&Cherniack, VLDB'03]

iter	pos	item
1	4	X
1	5	Y
2	10	Z
3	1	A
3	2	B
3	4	C

[iter,pos]

➔

Peephole Optimization
[Boncz et al., SIGMOD 2006]

- **Sort Avoidance**

generated by `DENSE RANK` **`pos`** `ORDER BY` **`pos`** `PARTITION BY` **`iter`**

by tracking secondary ordering properties **`[pos|iter]`**, [Wang&Cherniack, VLDB'03]

iter	pos	item
1	4	X
1	5	Y
2	10	Z
3	1	A
3	2	B
3	4	C

[iter,pos]

➔

iter	pos	item
1	1	X
1	2	Y
2	1	Z
3	1	A
3	2	B
3	3	C

Peephole Optimization
[Boncz et al., SIGMOD 2006]

- **Sort Avoidance**

generated by `DENSE RANK` **`pos`** `ORDER BY` **`pos`** `PARTITION BY` **`iter`**

by tracking secondary ordering properties **`[pos|iter]`**, [Wang&Cherniack, VLDB'03]

➔ hash-based (streaming) DENSE_RANK

iter	pos	item
1	4	X
2	10	Z
3	1	A
1	5	Y
3	2	B
3	4	C

➔

iter	pos	item
1	1	X
2	1	Z
3	1	A
1	2	Y
3	2	B
3	3	C

[pos|iter]

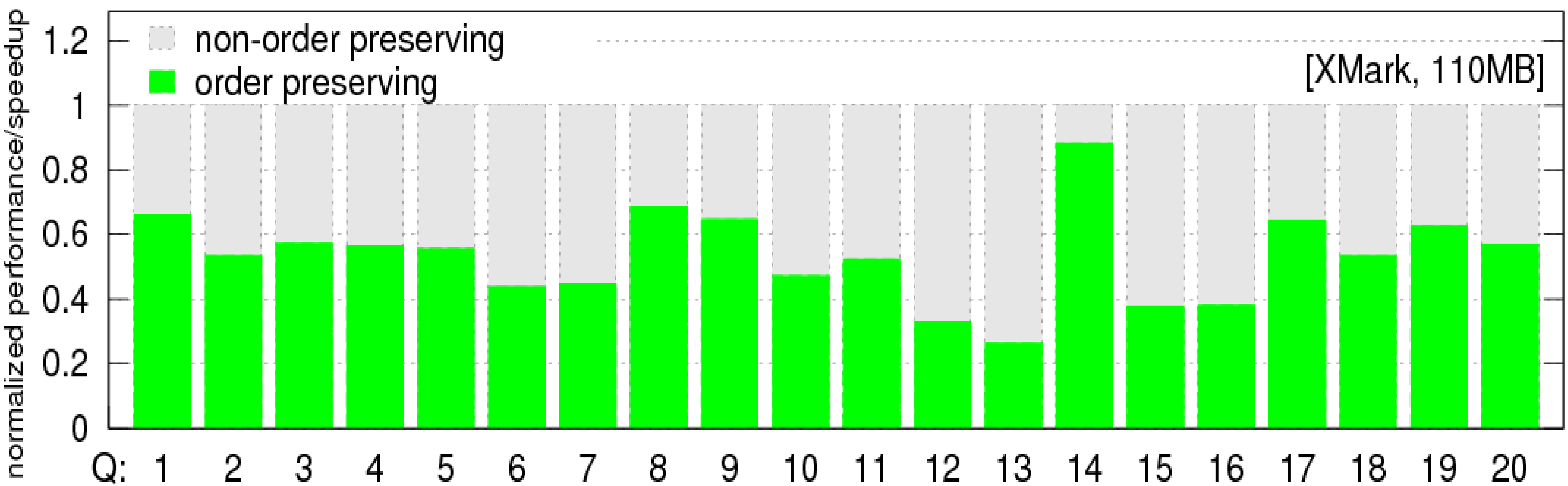
Peephole Optimization
[Boncz et al., SIGMOD 2006]

- **Sort Avoidance**

generated by `DENSE RANK` **`pos`** `ORDER BY` **`pos`** `PARTITION BY` **`iter`**

by tracking secondary ordering properties **`[pos|iter]`**, [Wang&Cherniack, VLDB'03]

➔ hash-based (streaming) DENSE_RANK



Peephole Optimization
[Boncz et al., SIGMOD 2006]

- Sort Avoidance

  generated by `DENSE RANK` **`pos`** `ORDER BY` **`pos`** `PARTITION BY` **`iter`**

  by tracking secondary ordering properties **`[pos|iter]`**, [Wang&Cherniack, VLDB'03]

  ➔ hash-based (streaming) DENSE_RANK

- **Sort Reduction**

  if [iter,item] order is required, and [iter] only is present

  use a **refine-sort** rather than a full sort.

  ➔ pipelinable sort

Peephole Optimization
[Boncz et al., SIGMOD 2006]

- Sort Avoidance

  generated by `DENSE RANK` **`pos`** `ORDER BY` **`pos`** `PARTITION BY` **`iter`**

  by tracking secondary ordering properties **`[item|iter]`**, [Wang&Cherniack, VLDB’03]

  ➔ hash-based (streaming) DENSE_RANK

- Sort Reduction

  if [iter,item] order is required, and [iter] only is present

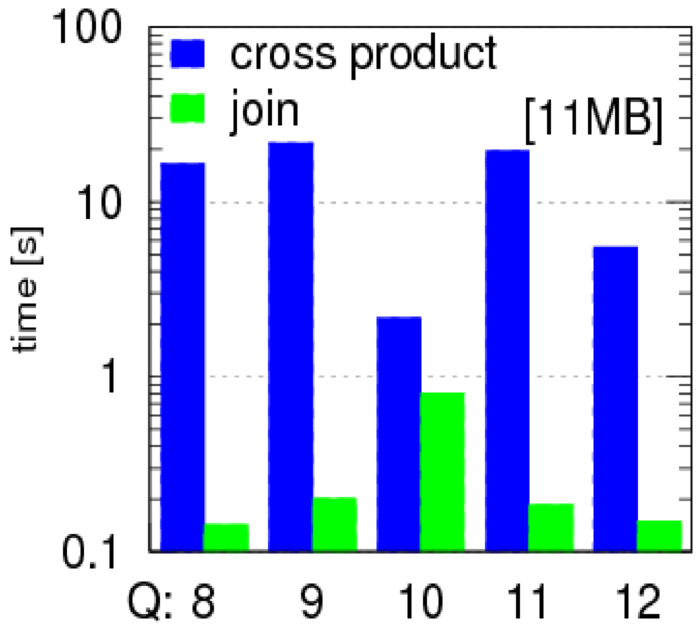
  use a **refine-sort** rather than a full sort.

  ➔ pipelinable sort

- **Join Detection**

  detect cartesian products as multi-valued dependencies

  in the precense of a theta-selection ➔ theta-join



Loop-lifted XPath Steps

Many algorithms have been proposed & studied for XPath evaluation:

- Dataguide based,
- Structural Join,
- Staircase Join,
- Holistic Twig Join

IN: sequence of context nodes in (doc order)
OUT: sequence of document nodes (unique, in doc order)

Loop-lifted XPath Steps

In XQuery, expressions generally occur inside FLWR blocks, i.e. inside a for-loop

```
for $x in doc()//employee
    $x/ancestor::department
```

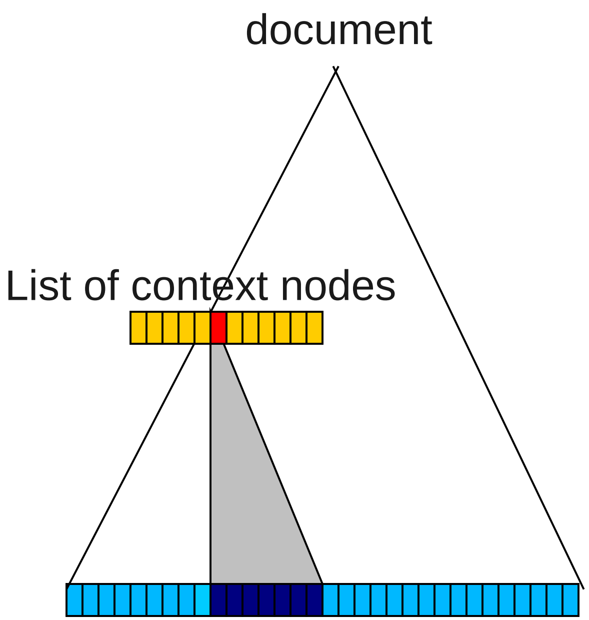
Choice:

• call XPath algorithm N times, accessing document and index structures N times.

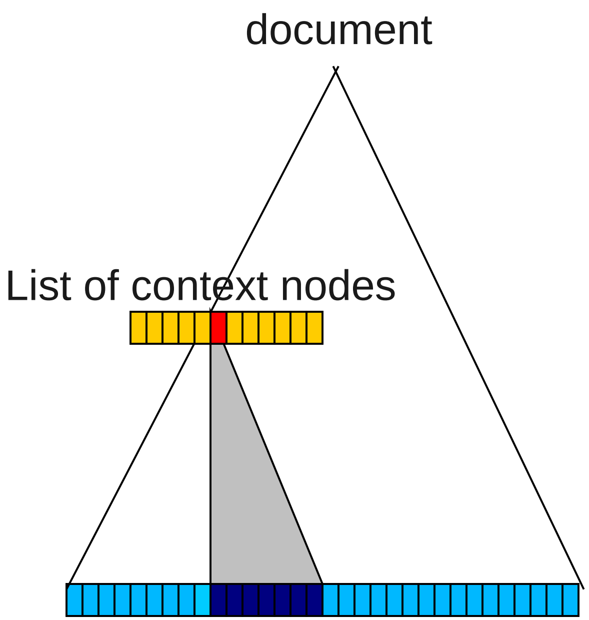
• use a **loop-lifted** algorithm:

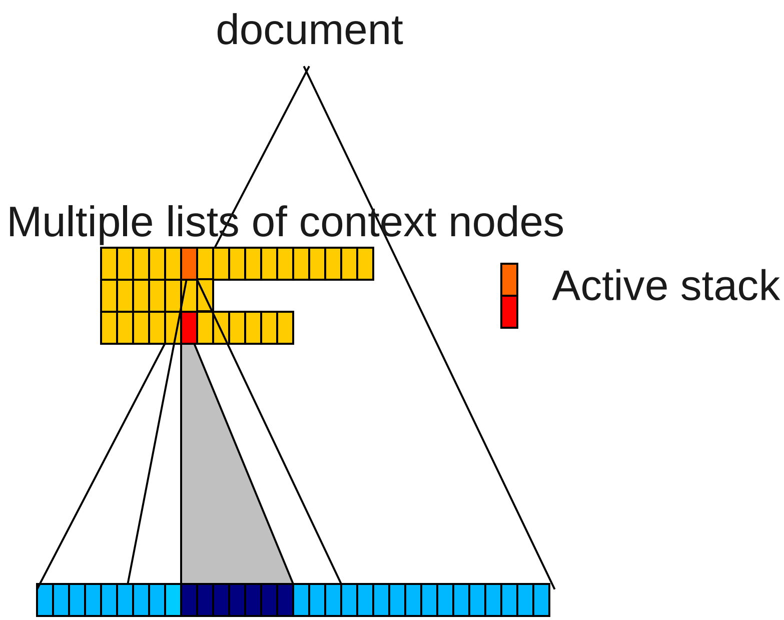
IN: for each iteration, a sequence of context nodes
OUT: for each iteration, a sequence of document nodes
(per iteration unique, in doc order)

Staircase join



Loop-lifted staircase join





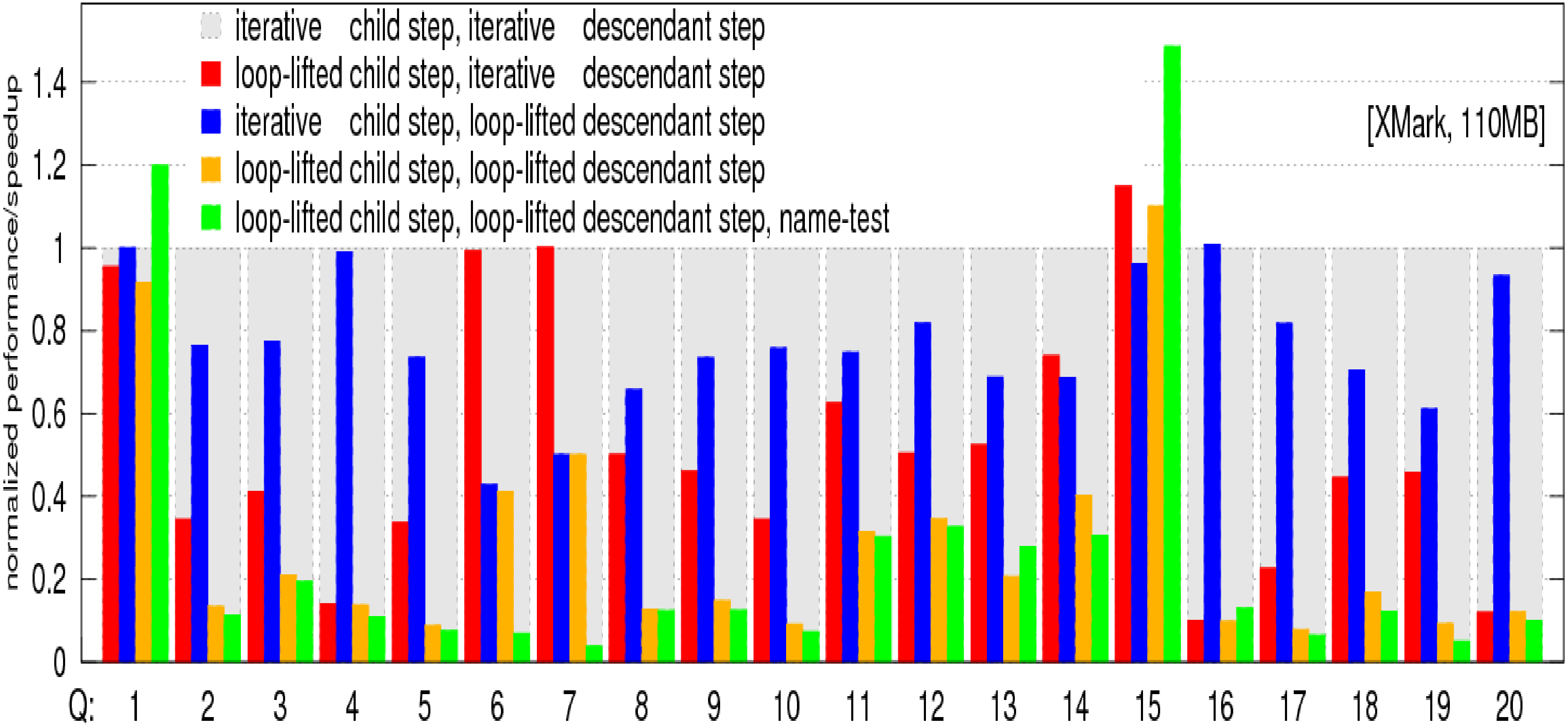
Adapt:

pruning, **partitioning** and **skipping** rules

to correctly deal with multiple context sets

Loop-lifted staircase join

Results on the 20 XMark queries:



Performance Evaluation

Extensive performance Evaluation on XMark

- data sizes 110KB, 1MB, 11MB, 110MB, 1.1GB, 11GB
- MonetDB/XQuery, Galax0.6, X-Hive 6.0, Berkeley DB XML 2.0, eXisT
- 8GB RAM

Extensive XMark performance Literature Overview

- IPSI-XQ v1.1.1b , Dynamic Interval Encoding , Kweelt , QuiP, FluX , TurboXPath, Timber, Qizx/Open (Version 0.4/p1), Saxon (Version 8.0), BEA/XQRL, VX
- Crude comparison (normalized by CPU SPECint)

XMark Benchmark

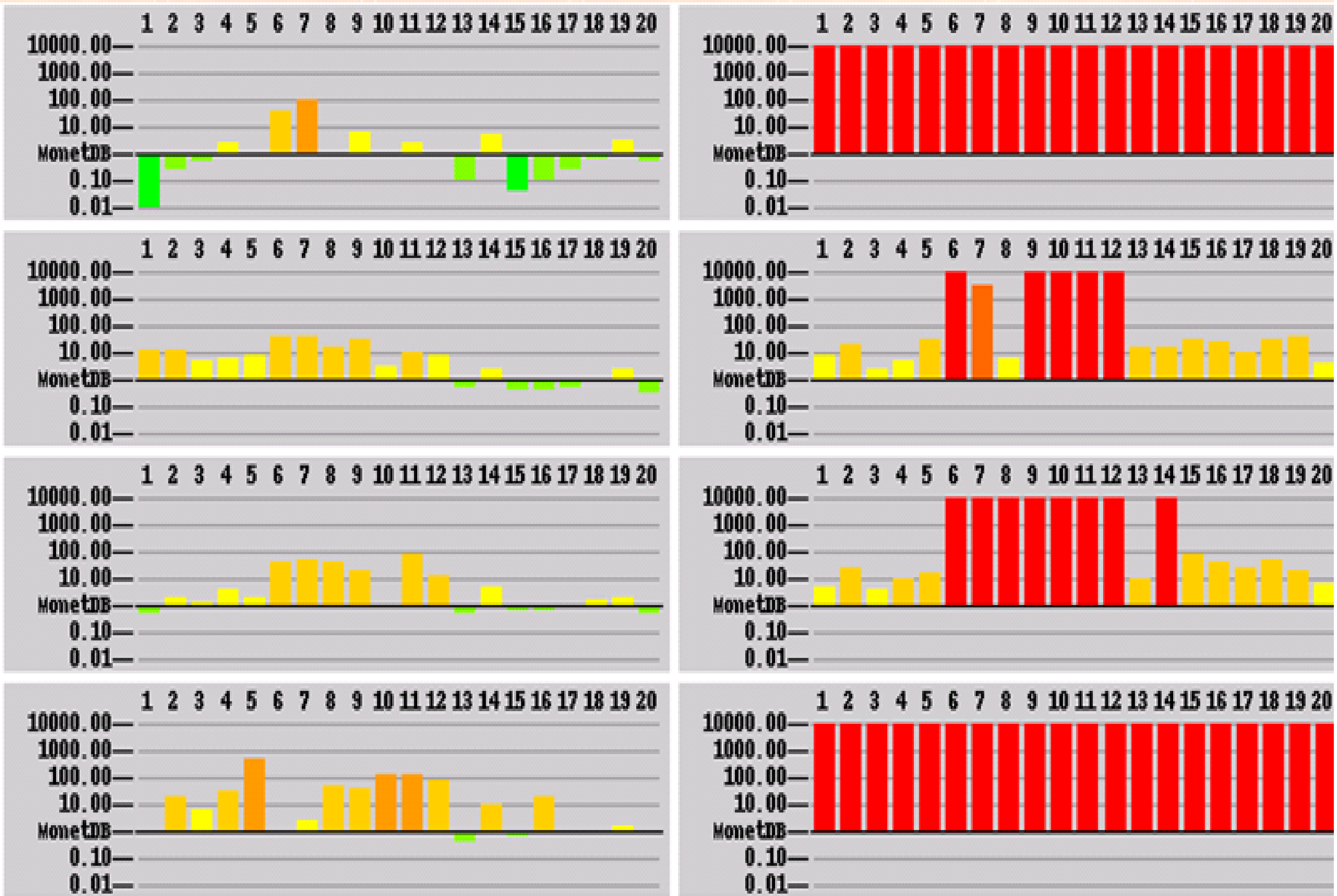
1MB XML | 1GB XML

Galax

X-Hive

Berkeley DB XML

eXisT



More Benchmarks & Performance Results?

- http://monetdb.cwi.nl/XQuery/Benchmark/
- S. Manegold: “An Empirical Evaluation of XQuery Processors”, *Information Systems*, 33:203-220, April 2008. http://repository.cwi.nl/search/fullrecord.php?publnr=13811