Chapter 8

Tree Algorithms - Twig Query

Acknowledgements

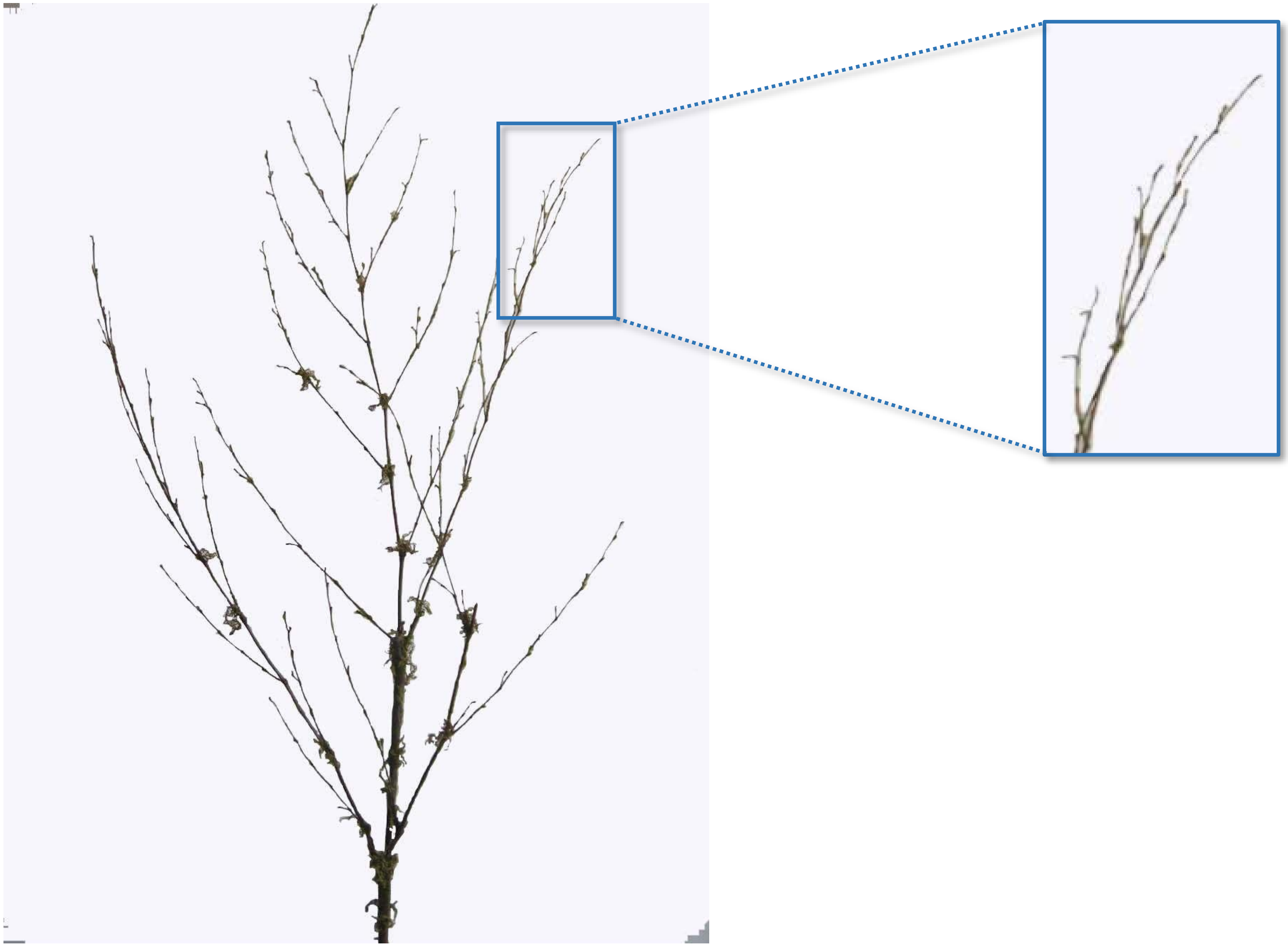
- Nicolas Bruno, Nick Koudas, Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. SIGMOD’02.
- Perter Buneman, Gao Cong, Wenfei Fan, Anastasios Kementsietsidis. Using Partial Evaluation in Distributed Query Evaluation. VLDB’06.
- Gao Cong, Wenfei Fan, Anastasios Kementsietsidis. Distributed Query Evaluation with Performance Guarantees. SIGMOD’07.
- Xin Bi, Guoren Wang, Xiangguo Zhao, Zhen Zhang, and Shuang Chen. Distributed XML Twig Query Processing Using MapReduce. APWeb’15.
- Xin Bi, Xiang-Guo Zhao, and Guo-Ren Wang. Efficient Processing of Distributed Twig Queries Based on Node Distribution. JCST’17.

Chapter Outline

- Holistic twig query processing
- Distributed algorithms

Holistic Twig Pattern Matching

Tree & Twig

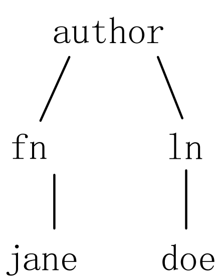


Twig Pattern Query

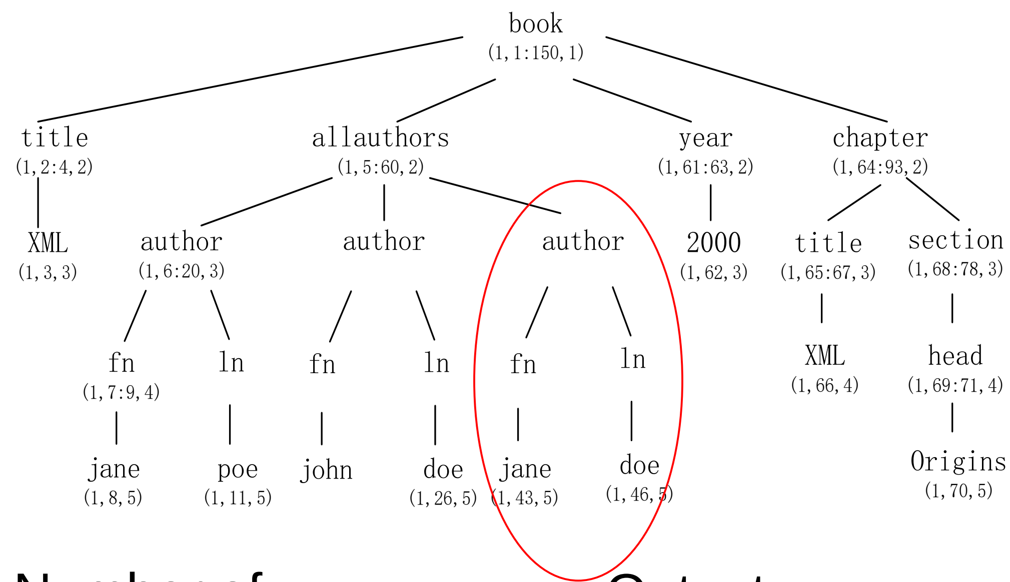
- Twig Pattern Query
  - Binary Structural Joins
    - The approach
      - Decompose the twig pattern into **binary** structural relationships
      - Use structural join algorithms to match the binary relationships against the XML database
      - Stitch together the basic matches
    - The problem
      - The intermediate result sizes can get large, even when the input and output sizes are more manageable

Binary Structural Join Example

Query



XML document



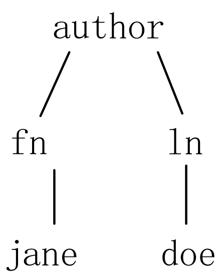
Decomposition	Number of Intermediate Results	Output
author – fn	3	1
author – ln	3	
fn – jane	2	
ln – doe	2	

Holistic Structural Joins

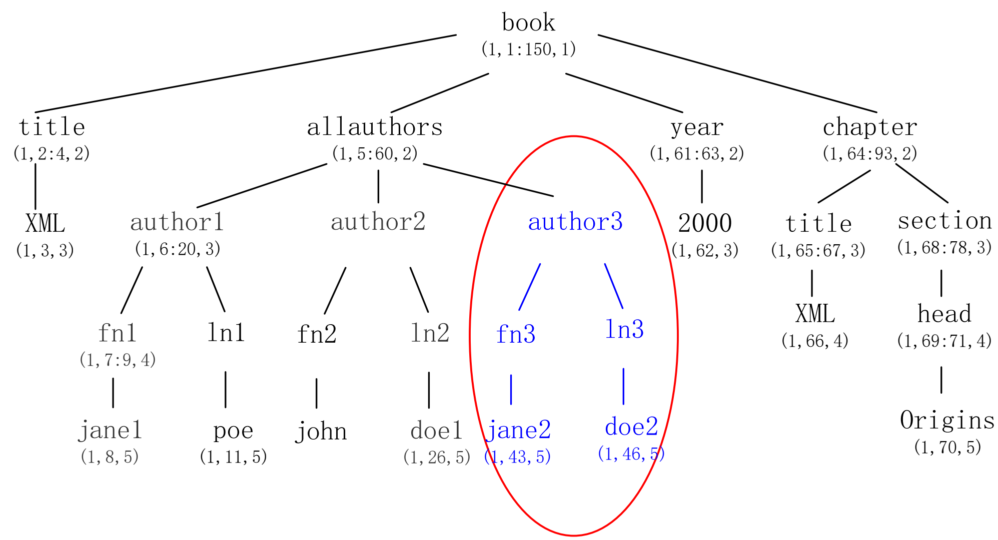
- The approach
  - Uses linked stacks to compactly represent partial results to **query paths**
  - Merges results to query paths to obtain matches for the twig pattern
- The advantage
  - It ensures that no intermediate solutions is larger than the final answer to the query
- **PathStack** and **TwigStack** [N.Bruno, SIGMOD'02]

Holistic Structural Join Example

Query



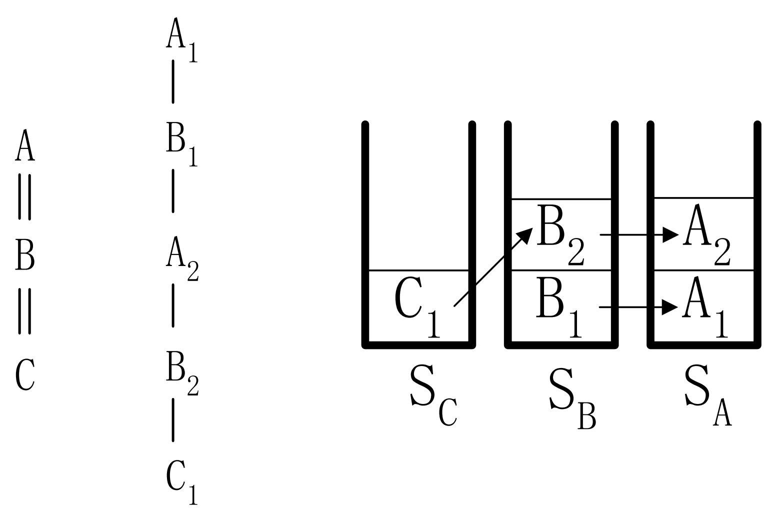
XML document



Decomposition	Intermediate Results	Number of Intermediate Results	Output
author – fn – jane	author3 – fn3 – jane2	1	1
author – ln – doe	author3 – ln3 – doe2	1	

PathStack

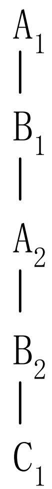
- Intuition
  - While the streams of the leaves are not empty (i.e. a solution could be found) do:
    - push the node with minimal LeftPos value into stack
    - if it is a leaf, print the solution
  - Example

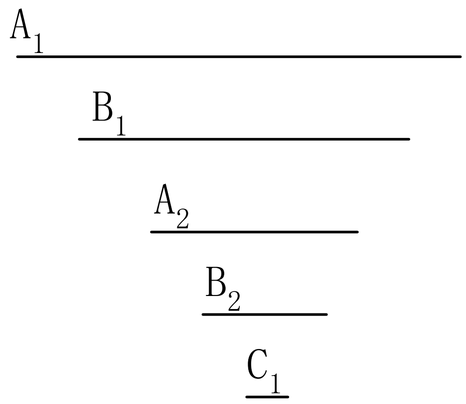


$A_1B_1C_1$

$A_1B_2C_1$

$A_2B_2C_1$





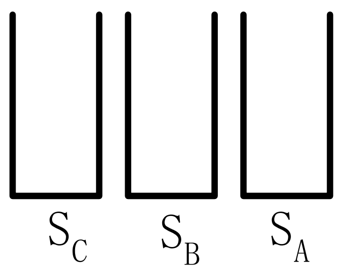
Streams

T_A: A_1, A_2

T_B: B_1, B_2

T_C: C_1

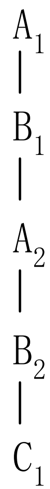
Stacks

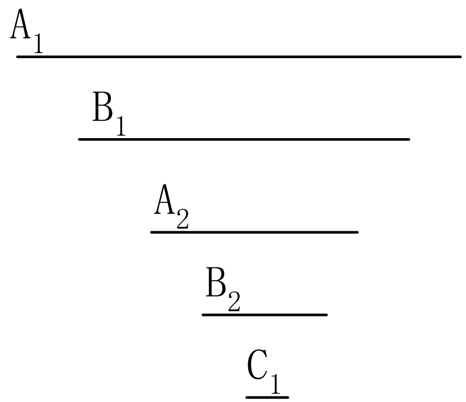


Comments

q_{min} = A

06) moveStreamToStack(T_A, S_A, null)





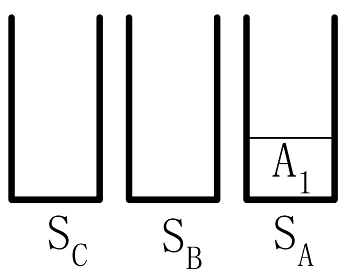
Streams

T_A: A_1, A_2

T_B: B_1, B_2

T_C: C_1

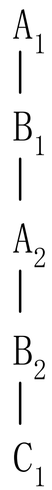
Stacks

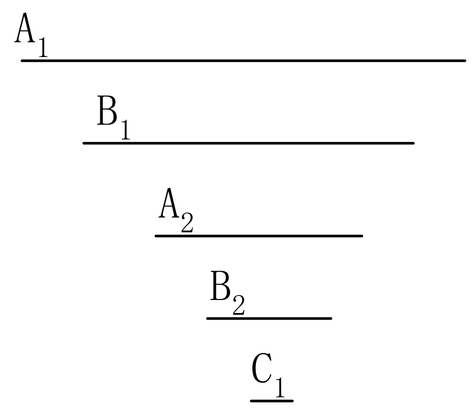


Comments

q_{min} = B

06) moveStreamToStack(T_B, S_B, A_1)





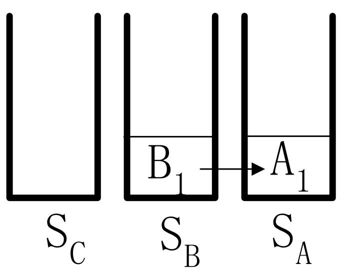
Streams

T_A: A_1, A_2

T_B: B_1, B_2

T_C: C_1

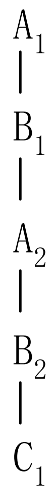
Stacks

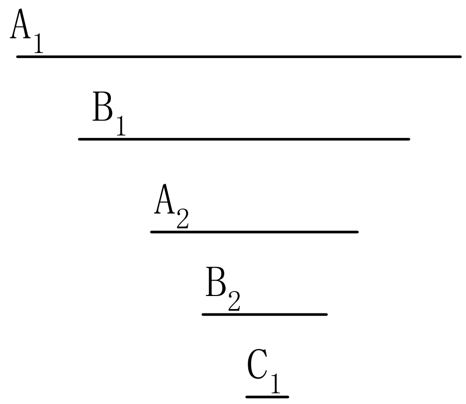


Comments

q_{min} = A

06) moveStreamToStack(T_A, S_A, null)





Streams

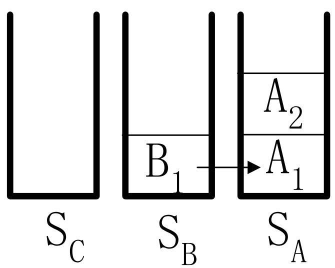
Stacks

Comments

T_A: A_1, A_2

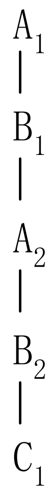
T_B: B_1, B_2

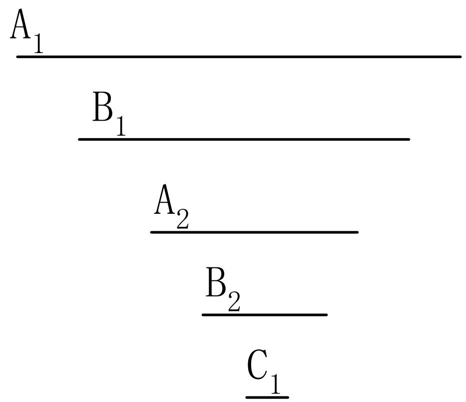
T_C: C_1



q_{min} = B

06) moveStreamToStack(T_B, S_B, A_2)





Streams

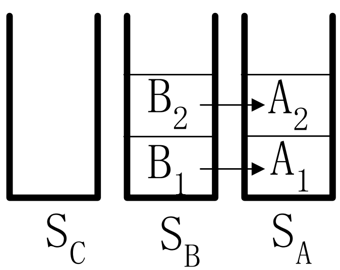
Stacks

Comments

T_A: A_1, A_2

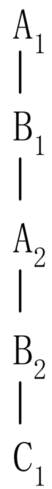
T_B: B_1, B_2

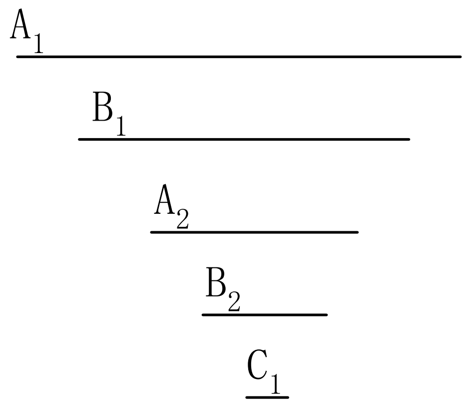
T_C: C_1



q_{min} = C

06) moveStreamToStack(T_C, S_C, B_2)





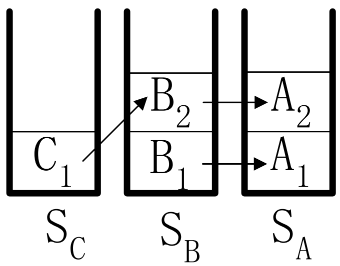
Streams

T_A: A_1, A_2

T_B: B_1, B_2

T_C: C_1

Stacks

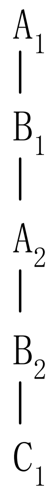


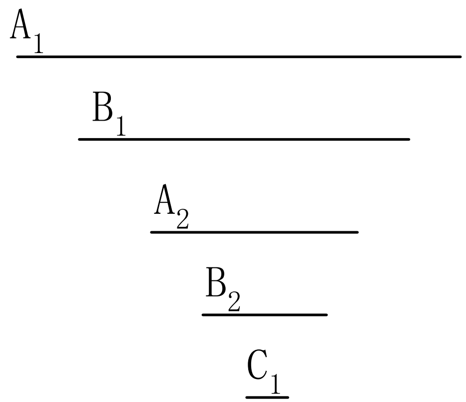
Comments

07) isLeaf(C) = true

08) showSolutions(S_C, 1)

09) pop(S_C)





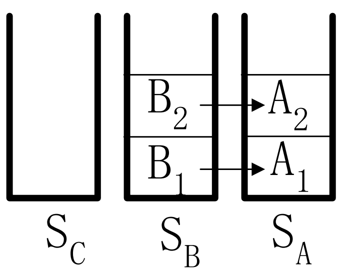
Streams

T_A: A_1, A_2

T_B: B_1, B_2

T_C: C_1

Stacks



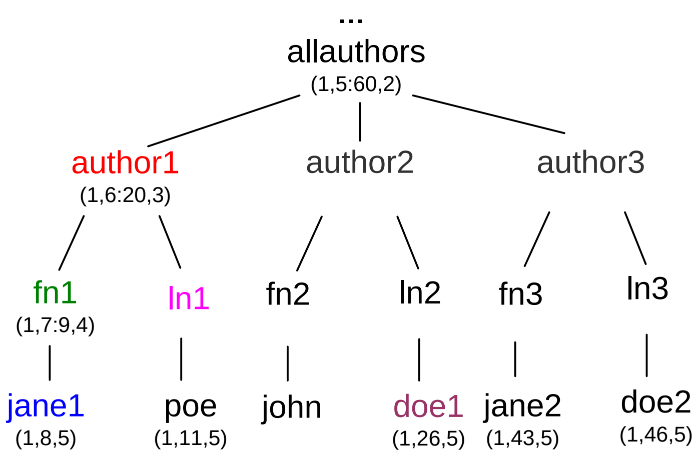
Comments

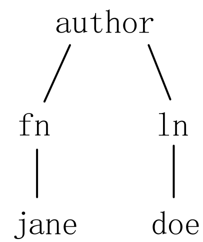
01) end(q) = true

Algorithm ends.

TwigStack

- Intuition
  - While the streams of the leaves are not empty (i.e. a solution could be found) do:
    - select a node that could be expanded to a solution
    - if it is a leaf, print the solution





Streams

T_a: a1, a2, a3

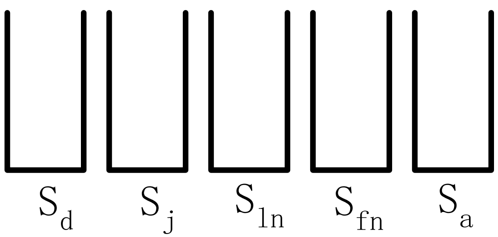
T_{fn}: fn1, fn2, fn3

T_{ln}: ln1, ln2, ln3

T_j: j1, j2

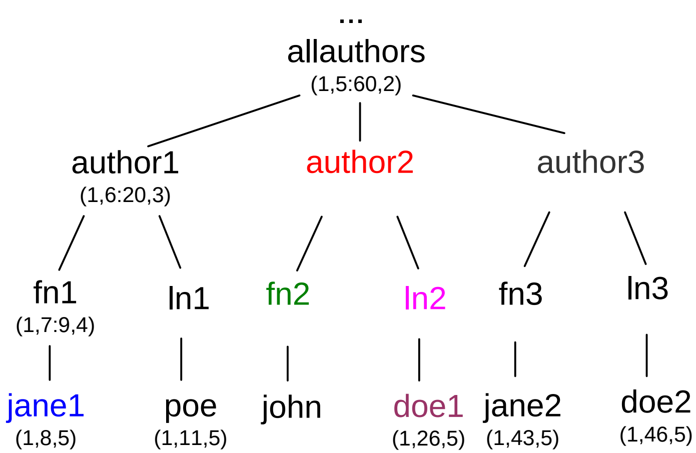
T_d: d1, d2

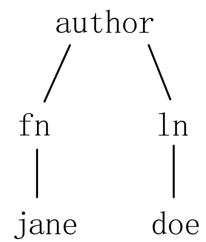
Stacks



Comments: Phase1

01: while (notEmpty(T_j) || notEmpty(T_d)) do:





Streams

T_a: a1, a2, a3

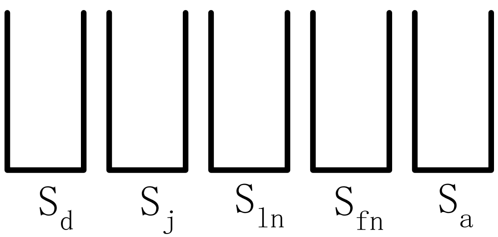
T_{fn}: fn1, fn2, fn3

T_{ln}: ln1, ln2, ln3

T_j: j1, j2

T_d: d1, d2

Stacks



Comments: iteration1

q_{act} = getNext(a) → *fn*

getNext(fn) → *fn*

getNext(j) → *j*

n_{min}=n_{max}=8 (j1)

getNext(ln) → *ln*

getNext(d) → *d*

n_{min}=n_{max}=26 (d1)

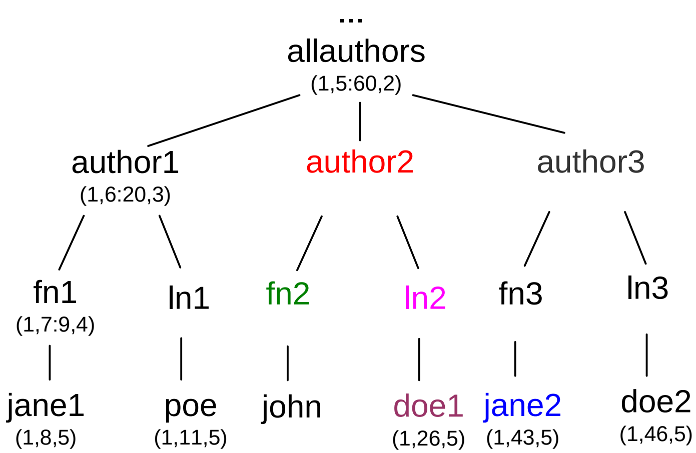
advance(ln)

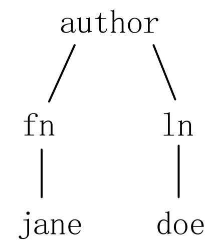
n_{min}=7(fn1)

n_{max}=ln2

advance(T_a)

advance(T_{fn})





Streams

T_a: a1, a2, a3

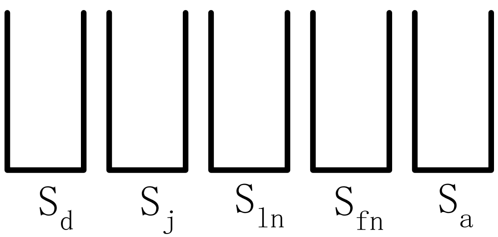
T_{fn}: fn1, fn2, fn3

T_{ln}: ln1, ln2, ln3

T_j: j1, j2

T_d: d1, d2

Stacks



Comments: iteration2

q_{act} = *getNext(a)* $\longrightarrow$ *j*

getNext(fn) $\longrightarrow$ *j*

getNext(j) $\longrightarrow$ *j*

$n_{min}=n_{max}=8$ *(j1)*

getNext(ln) $\longrightarrow$ *ln*

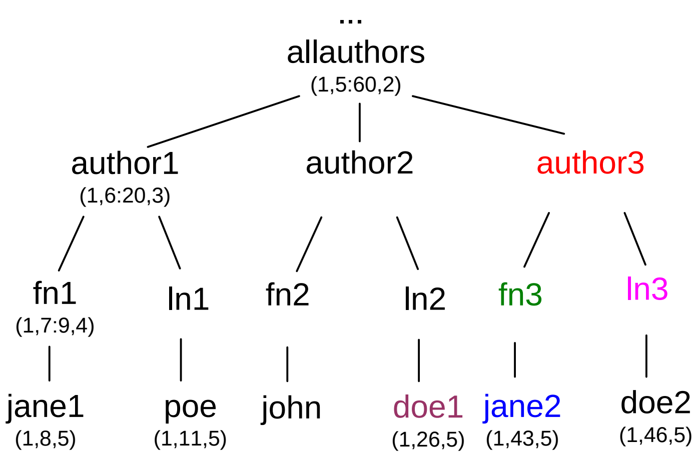
getNext(d) $\longrightarrow$ *d*

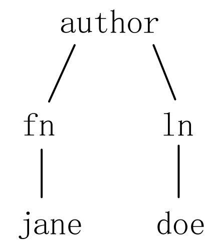
$n_{min}=n_{max}=26$ *(d1)*

$n_{min}=8$*(j1)*

n_{max}=*ln2*

advance(T_j)





Streams

T_a: a1, a2, a3

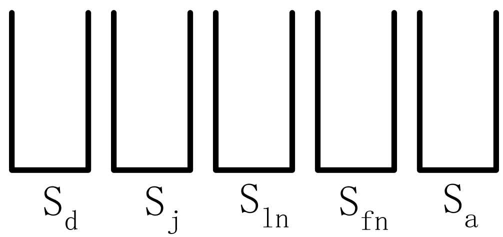
T_{fn}: fn1, fn2, fn3

T_{ln}: ln1, ln2, ln3

T_j: j1, j2

T_d: d1, d2

Stacks



Comments: iteration3

q_{act} = *getNext(a)* $\longrightarrow$ *ln*

getNext(fn) $\longrightarrow$ *fn*

getNext(j) $\longrightarrow$ *j*

$n_{min}=n_{max}=43$ *(j2)*

advance(fn)

getNext(ln) $\longrightarrow$ *ln*

getNext(d) $\longrightarrow$ *d*

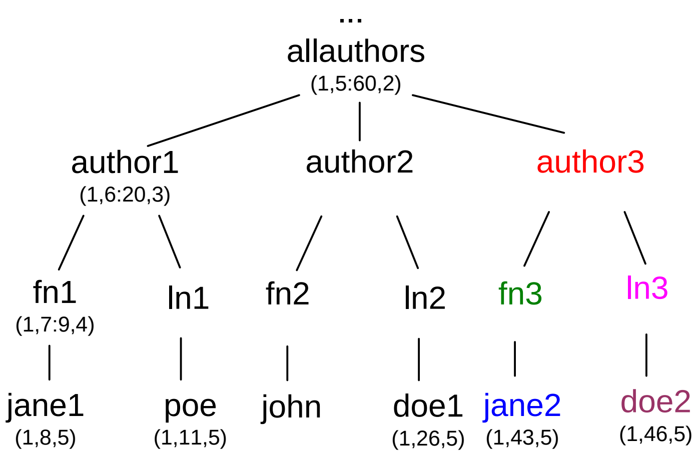
$n_{min}=n_{max}=26$ *(d1)*

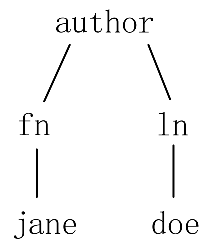
n_{min}*=ln2*

n_{max}*=fn3*

advance(T_a)

advance(T_{ln})





Streams

T_a: a1, a2, a3

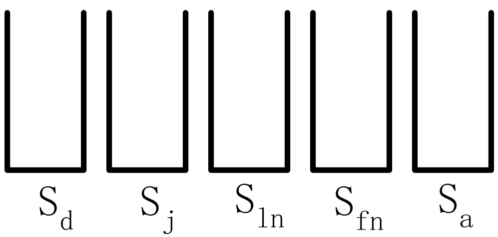
T_{fn}: fn1, fn2, fn3

T_{ln}: ln1, ln2, ln3

T_j: j1, j2

T_d: d1, d2

Stacks



Comments: iteration4

q_{act} = getNext(a) → *d*

getNext(fn) → *fn*

getNext(j) → *j*

n_{min}=n_{max}=43 (j2)

getNext(ln) → *d*

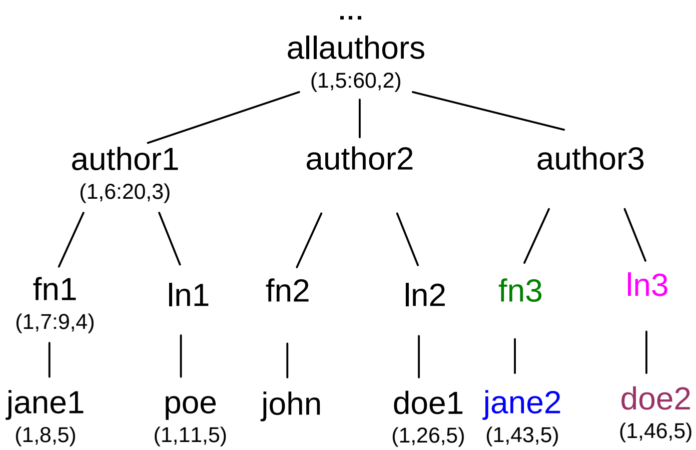
getNext(d) → *d*

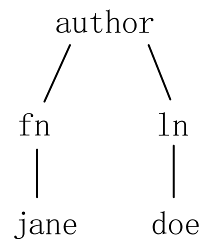
n_{min}=n_{max}=26 (d1)

n_{min}=26(d1)

n_{max}=fn3

advance(T_d)





Streams

T_a: a1, a2, a3

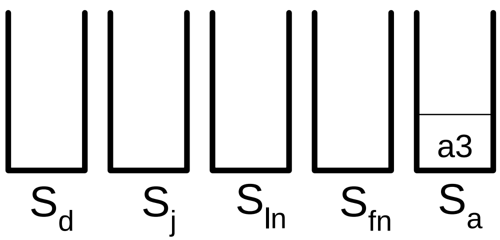
T_{fn}: fn1, fn2, fn3

T_{ln}: ln1, ln2, ln3

T_j: j1, j2

T_d: d1, d2

Stacks



Comments: iteration5

q_{act} = *getNext(a)* → *a*

getNext(fn) → *fn*

getNext(j) → *j*

$n_{min}=n_{max}=43$ *(j2)*

getNext(ln) → *ln*

getNext(d) → *d*

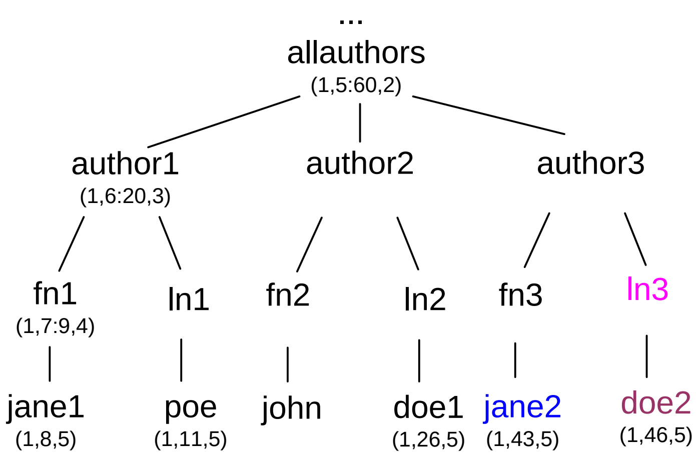
$n_{min}=n_{max}=46$ *(d2)*

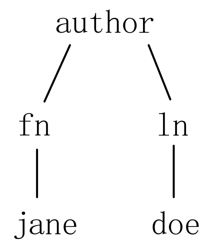
n_{min}*=fn3*

n_{max}*=ln3*

moveStreamToStack(T_a)

advance(T_a)





Streams

T_a: a1, a2, a3

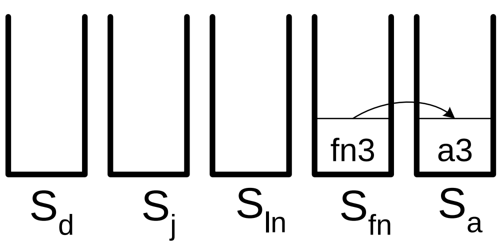
T_{fn}: fn1, fn2, fn3

T_{ln}: ln1, ln2, ln3

T_j: j1, j2

T_d: d1, d2

Stacks



Comments: iteration6

q_{act} = *getNext(a)* → *fn*

getNext(fn) → *fn*

getNext(j) → *j*

$n_{min}=n_{max}=43$ *(j2)*

getNext(ln) → *ln*

getNext(d) → *d*

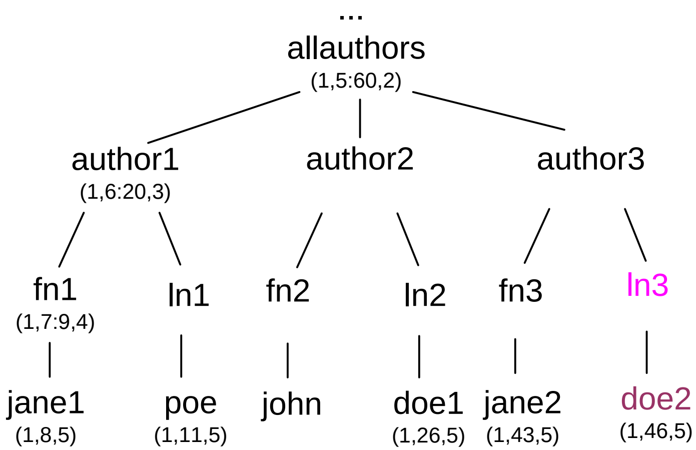
$n_{min}=n_{max}=46$ *(d2)*

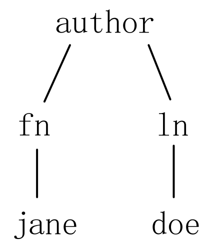
n_{min}*=fn3*

n_{max}*=ln3*

moveStreamToStack(T_{fn})

advance(T_{fn})





Streams

T_a: a1, a2, a3

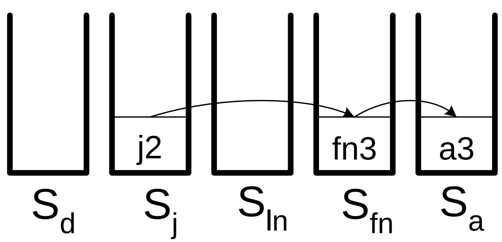
T_{fn}: fn1, fn2, fn3

T_{ln}: ln1, ln2, ln3

T_j: j1, j2

T_d: d1, d2

Stacks



"Merge-joinable" root-to-leaf path: (j2, fn3, a3)

Comments: iteration7

q_{act} = getNext(a) $\longrightarrow$ *j*

getNext(fn) $\longrightarrow$ *j*

getNext(j) $\longrightarrow$ *j*

n_{min}=n_{max}=43 (j2)

getNext(ln) $\longrightarrow$ *ln*

getNext(d) $\longrightarrow$ *d*

n_{min}=n_{max}=46 (d2)

n_{min}=43(j2)

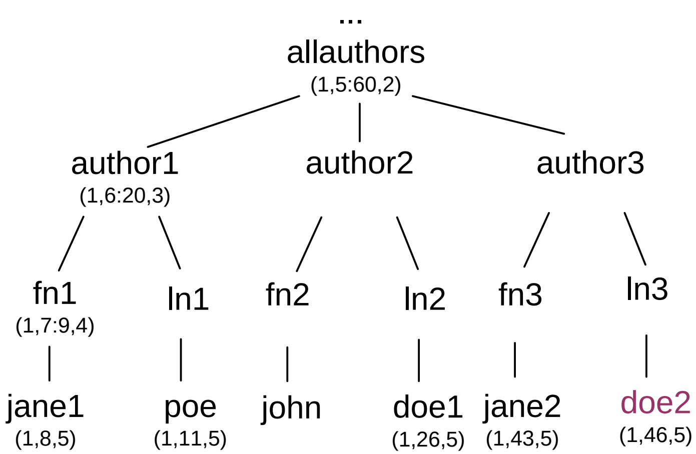
n_{max}=ln3

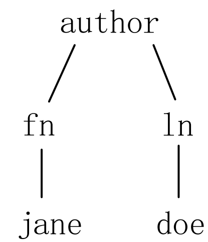
moveStreamToStack(T_j)

advance(T_j)

pop(S_j)

showSolutionsWithBlocking(j)





Streams

T_a: a1, a2, a3

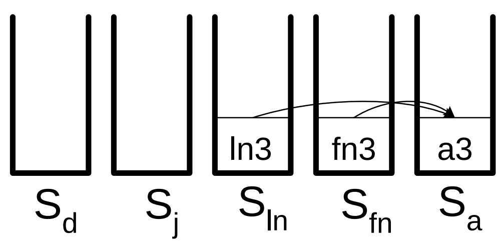
T_{fn}: fn1, fn2, fn3

T_{ln}: ln1, ln2, ln3

T_j: j1, j2

T_d: d1, d2

Stacks



"Merge-joinable" root-to-leaf path: (j2, fn3, a3)

Comments: iteration8

q_{act} = getNext(a) $\longrightarrow$ *ln3*

getNext(fn) $\longrightarrow$ *nil*

getNext(j) $\longrightarrow$ *nil*

n_{min}=n_{max}=nil

getNext(ln) $\longrightarrow$ *ln*

getNext(d) $\longrightarrow$ *d*

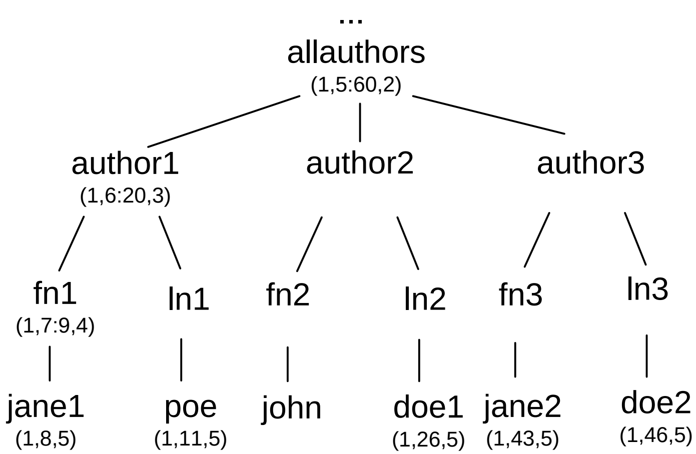
n_{min}=n_{max}=46 (d2)

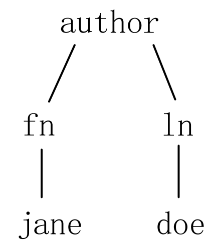
n_{min}=ln3

n_{max}=ln3

moveStreamToStack(T_{ln})

advance(T_{ln})





Streams

T_a: a1, a2, a3

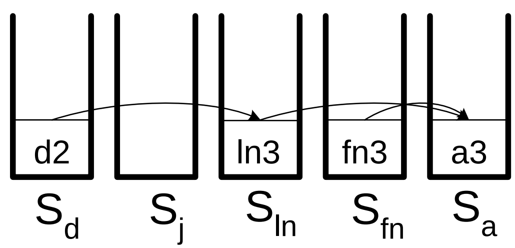
T_{fn}: fn1, fn2, fn3

T_{ln}: ln1, ln2, ln3

T_j: j1, j2

T_d: d1, d2

Stacks



"Merge-joinable" root-to-leaf paths: (j2, fn3, a3)

(d2, ln3, a3)

Comments: iteration9

q_{act} = getNext(a) → *ln3*

getNext(fn) → *nil*

getNext(j) → *nil*

n_{min}=n_{max}=nil

getNext(ln) → *d*

getNext(d) → *d*

n_{min}=n_{max}=46 (d2)

n_{min}=d

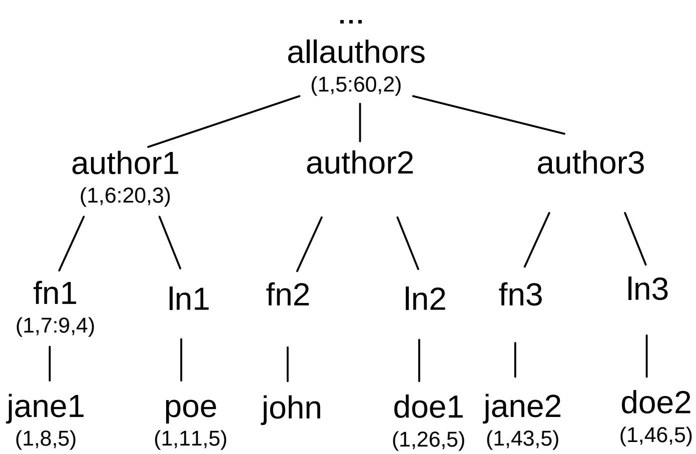
n_{max}=d

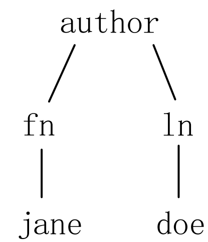
moveStreamToStack(T_d)

advance(T_d)

pop(S_d)

showSolutionsWithBlocking(d)





Streams

T_a: a1, a2, a3

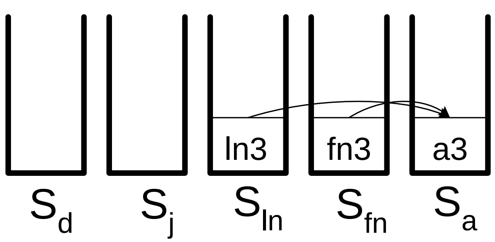
T_{fn}: fn1, fn2, fn3

T_{ln}: ln1, ln2, ln3

T_j: j1, j2

T_d: d1, d2

Stacks



TwigStack solution:

(j2, fn3, d2, ln3, a3)

Comments: Phase2

12: MergeAllPathSolutions()

Distributed Twig Query Processing

Distributed Algorithms

- Distributed TwigStack
- DisT3/DisT2
- ParBoX
- PaX3/PaX2

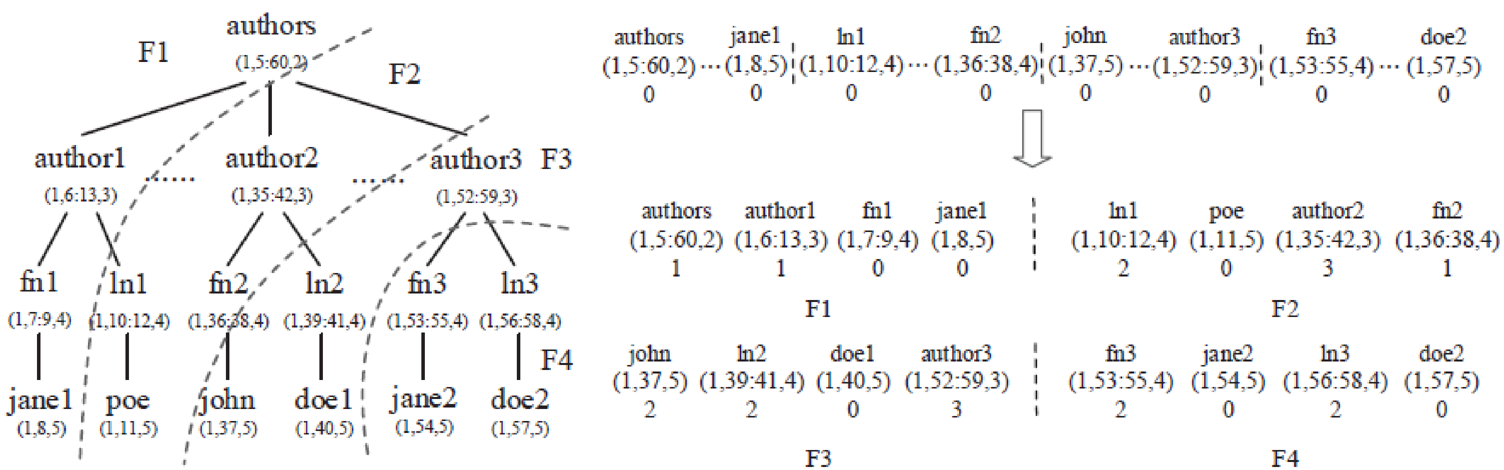
Distributed TwigStack

Arbitrary XML partition

- Arbitrariness
  - Stored on distributed file system before query processing
  - Impractical to repartition and restore according to each query
- <label,rcode,type> as an XML node
  - no-cut (type 0): neither the parent edge nor a child edge is cut;
  - child-edge-cut (type 1): at least a child edges is cut, but not the parent edge;
  - parent-edge-cut (type 2): the parent edge is cut, but not the child edges;
  - all-cut (type 3): both the parent edge and at least one child edge are cut.

Arbitrary XML partition

- An example



(a) An XML document (b) The node stream

Distributed query processing

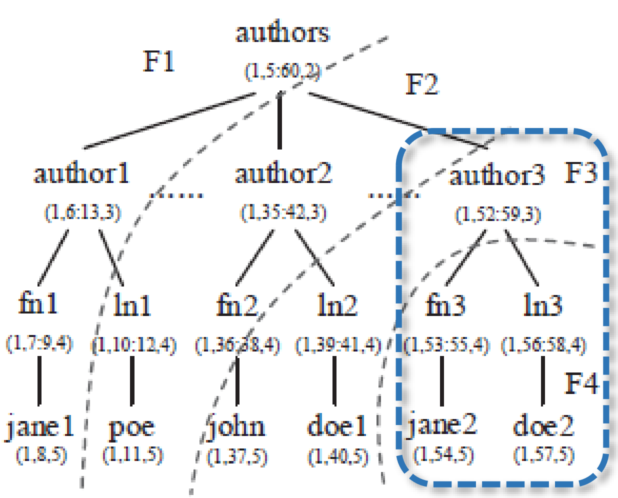
- Naïve solution (2-round MapReduce)
  - Round 1
    - Map: find all the matches to the query nodes on the local machine, emit key-value pairs using *the match node* as value and its *highest ancestor which is also a match of root of the query* as key;
    - Reduce: collect the key-value pairs for the next round. Each final result can be derived from a sub-tree rooted by a key;
  - Round 2
    - Map: run a twig algorithm to generate final results;
    - Reduce: collect all the final results.
- Drawbacks
  - Unnecessary intermediate results in the first round
  - MapReduce job start-up is time-consuming

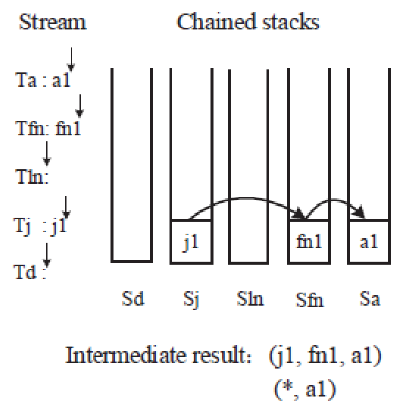
Distributed TwigStack

- Intuition
  - Relaxed matching rules to retain results across partitions
  - Coordinator to guarantee the distribution of intermediate results
- DTS
  - Relaxed Local TwigStack (RL-TwigStack)
    - Relaxed Local GetNext (RL-GetNext)
    - Relaxed Local ShowSolutions (RL-ShowSolutions)
    - Emit <key, solution>
  - Coordinator
    - Refined key-value pairs <refinedKey, solution>

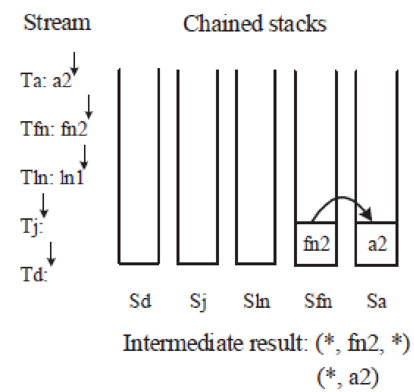
Distributed TwigStack

- //author[fn/text()="jane"∧ln/text()="doe"]

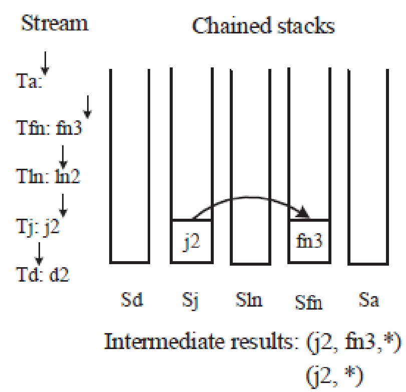




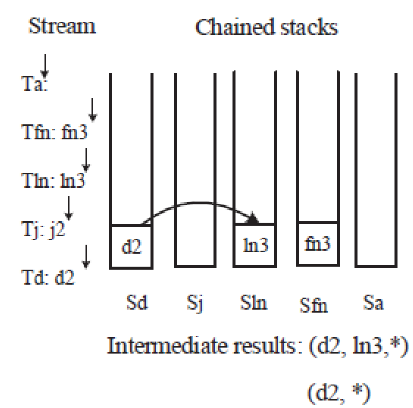
(a) F1



(b) F2



(c) F4



(d) F4

DisT3/DisT2

Motivation

1. repartitioning and restoring large-scale XML data for different applications are impractical.
2. the intrusion into the existing distributed framework, such as Hadoop, should be minimized.
3. any partition strategy should be supported, including the default partition strategy of Hadoop.

Aim at processing large-scale tree data efficiently no matter how the data are partitioned and stored on the distributed file system

Major Concerns

- ***Data Storage***. The algorithm should support arbitrary partitions without the prior knowledge of query patterns, especially in the cloud environment.
- ***Local Computation***. The local computation should achieve high efficiency and pruning ability to reduce computation cost.
- ***Network Traffic***. The parallelism should be improved so that the algorithm consumes minimal communication cost among machines in the cluster.

DisT3: Principles

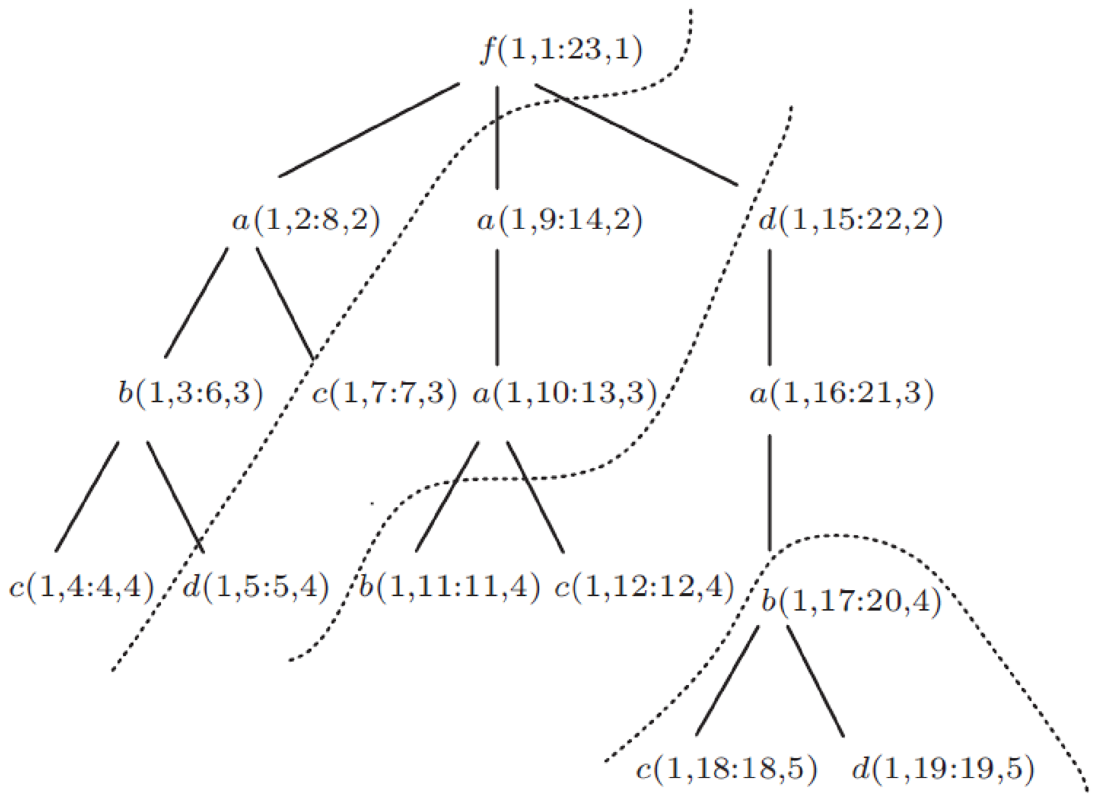
- Since the region code can be used to determine the relationship between different nodes, we use it as the key for tuple identification.
- To be specific, we use *rc* of the match nodes of the root node in query *Q* as the key, so that all the match nodes of *Q* will have the same key value.
- All the keys from each partition have to be gathered by a coordinator, so that all the keys can be reset to their highest ancestors.

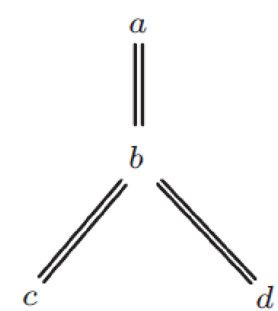
DisT3: Phases

1. Each machine scans each partition and sends *rc* of each match node to the coordinator.
2. The coordinator resets the key to the *rc* of its highest ancestors, and sends them back to all the partitions.
3. Each partition emits the nodes with their refined keys. All the nodes with the same keys are gathered in the same machines and processed by local twig join operations.

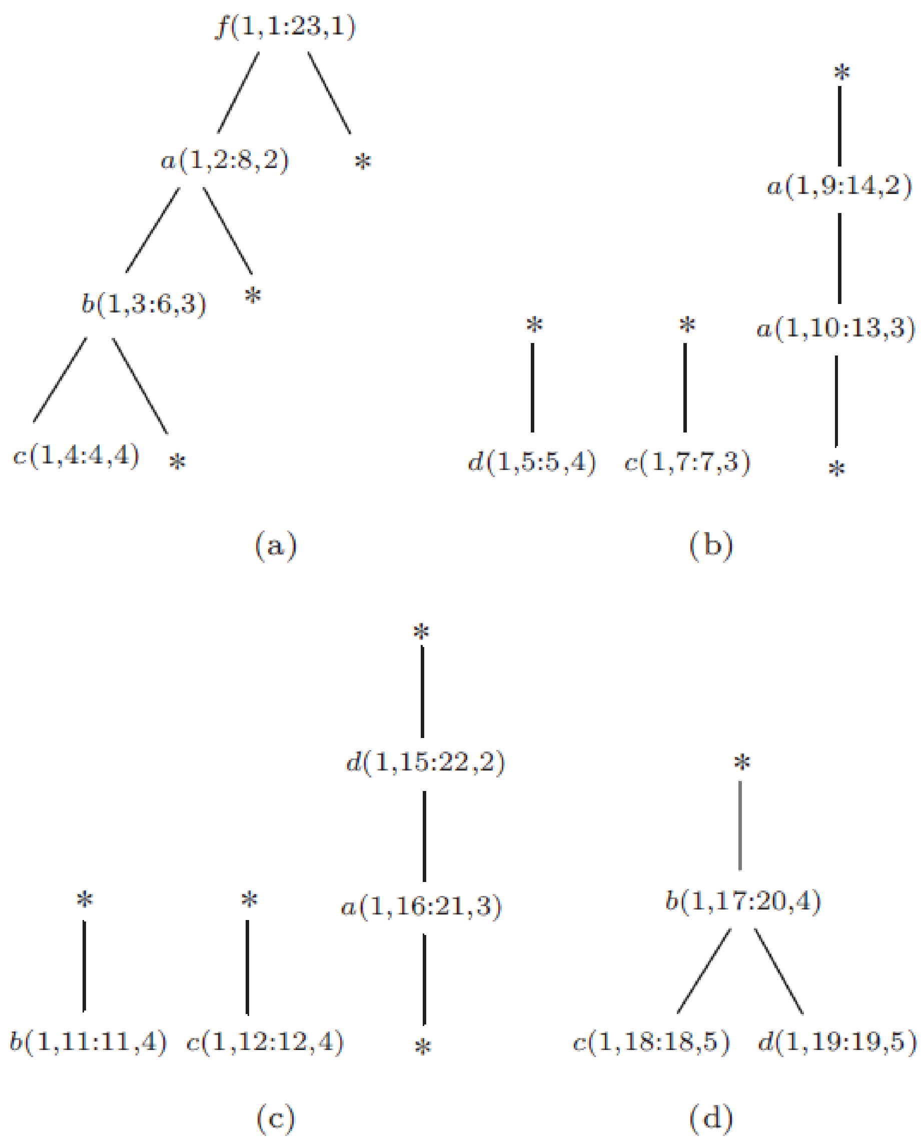
Example: Raw Data

- Each node is represented as a 2-tuple <*l*, *rc*>, where *l* is the label and *rc* is the region code

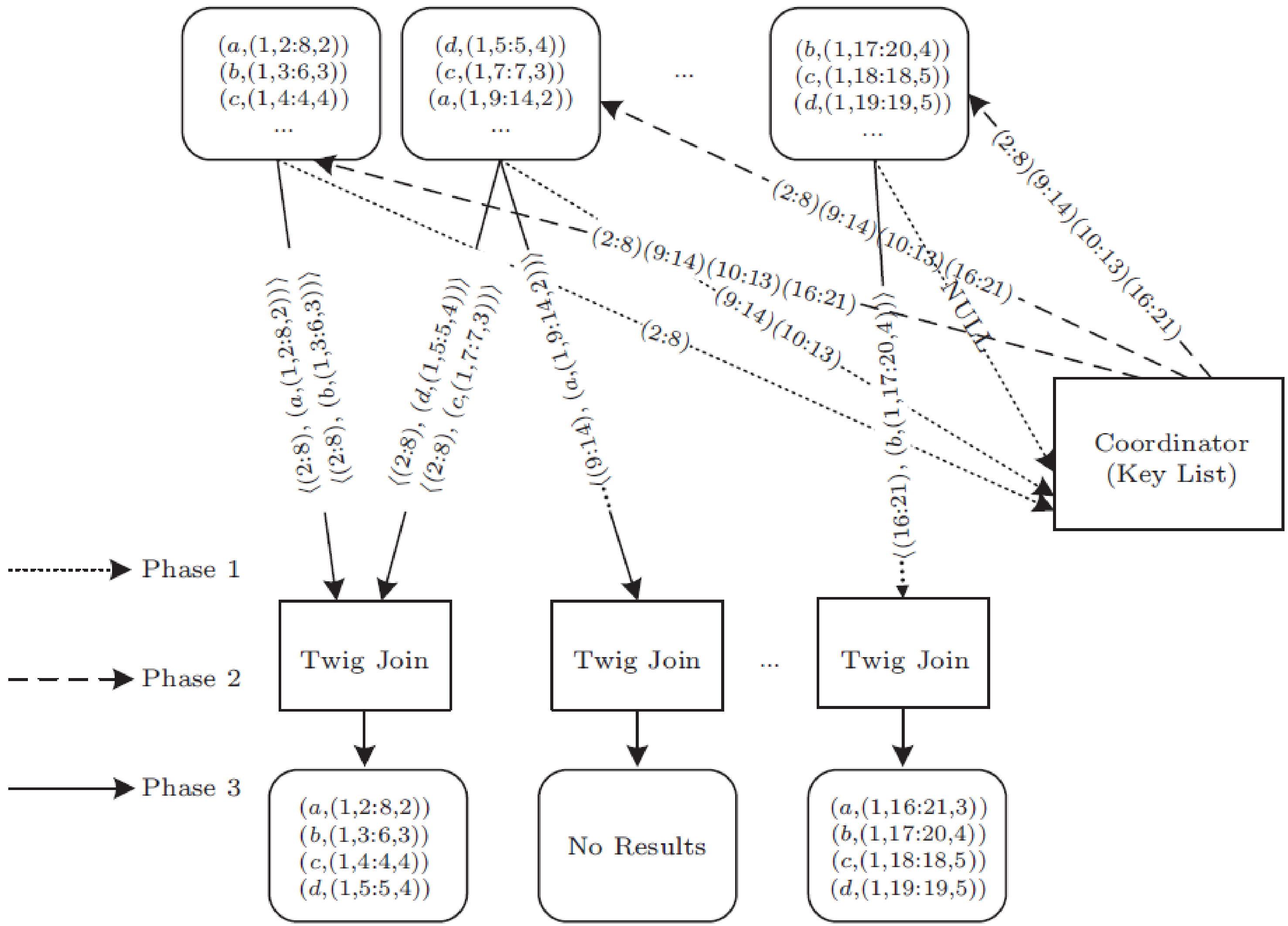




Example: Partitions



Example: Flowchart



DisT2ReP

- Definitions

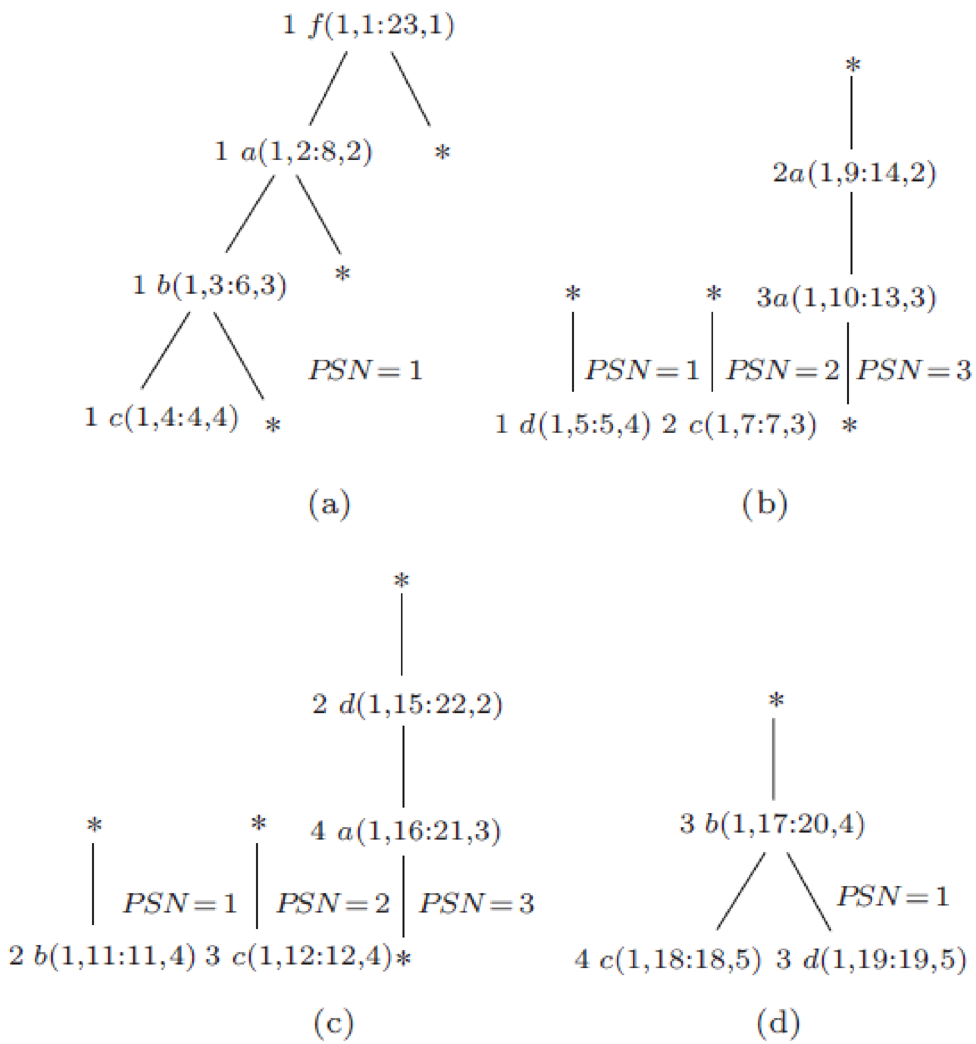
Definition 2 (Virtual Ancestor Label). *Given an XML document D and its partitions ($F1$, $F2$, ..., Fn), for Fi, if a node v on the other partitions has descendent nodes in Fi, then v is a virtual ancestor of Fi, and the label of v is a virtual ancestor label, denoted as VAL.*

Definition 3 (Global Label Number). *Given an XML document D and its streams S of each label, for a node v with label a and its label stream $S_a \in S$, the sequence number of node v in S_a is the global label number of v, denoted as GLN.*

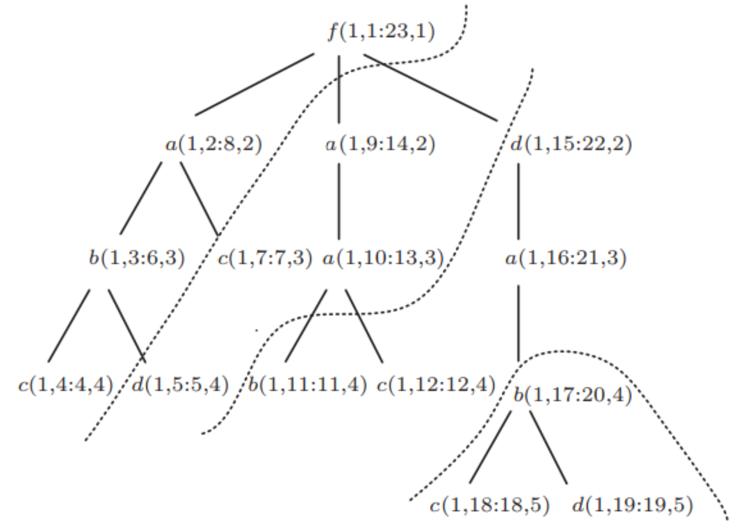
Definition 4 (Partition Subtree Number). *Given a partition Fi and all the subtrees (t_1^i, t_2^i, ..., t_m^i) on Fi, all the nodes in each subtree t_j have the same partition subtree number j, denoted as PSN.*

Definition 5 (Partition Subtree Region). *Given a prtition Fi, all the subtrees (t_1^i, t_2^i, ..., t_m^i) on Fi and a virtual ancestor u, if node u is an ancestor of subtrees (t_j^i, ..., t_k^i), then the partition subtree region of node u is denoted as $PSR = (GLN_u, PSN_j^i{:}PSN_k^i)$, where GLN_u is the GLN of node u, and PSN_j^i and PSN_k^i are the PSN of subtrees t_j^i and t_k^i on partition Fi respectively.*

DisT2ReP: PaSS and ReP Index



DisT2ReP



label:	*f*		*c* ①		*d*		*a* ②		*b*		*a* ③		*b*		*d* ④
rc:	(1,1:23,1)	⋯	(1,4:4,4)	|	(1,5:5,4)	⋯	(1,10:13,3)	|	(1,11:11,4)	⋯	(1,16:21,3)	|	(1,17:20,4)	⋯	(1,19:19,5)
GLN:	1		1	|	1		3	|	2		4	|	3		3
PSN:	1		1	|	1		3	|	1		3	|	1		1

(a)

Partition

⟨*label*, *regionCode*, *reCount*, *sid*⟩

f	(1,1:23,1)	1	1
a	(1,2:8,2)	1	1
b	(1,3:6,3)	1	1
c	(1,4:4,4)	1	1

ReP Index

⟨*ancLabel*, (*reCount*, *sidRegion*)⟩

Null	Null

(b)

Partition

⟨*label*, *regionCode*, *reCount*, *sid*⟩

d	(1,5:5,4)	1	1
c	(1,7:7,3)	2	2
a	(1,9:14,2)	2	3
a	(1,10:13,3)	3	3

ReP Index

⟨*ancLabel*, (*reCount*, *sidRegion*)⟩

f	(1,1:3)
a	(1,1:2)
b	(1,1:1)

(c)

Partition

⟨*label*, *regionCode*, *reCount*, *sid*⟩

b	(1,11:11,4)	2	1
c	(1,12:12,4)	3	2
d	(1,15:22,2)	2	3
a	(1,16:21,3)	4	3

ReP Index

⟨*ancLabel*, (*reCount*, *sidRegion*)⟩

f	(1,1:3)
a	(2,1:2)(3,1:2)

(d)

Partition

⟨*label*, *regionCode*, *reCount*, *sid*⟩

b	(1,17:20,4)	3	1
c	(1,18:18,5)	4	1
d	(1,19:19,5)	3	1

ReP Index

⟨*ancLabel*, (*reCount*, *sidRegion*)⟩

f	(1,1:1)
d	(2,1:1)
a	(4,1:1)

(e)

Example

label:	*f*		*c* ①	*d*		*a* ②	*b*		*a* ③	*b*		*d* ④
rc:	(1,1:23,1)	···	(1,4:4,4)	(1,5:5,4)	···	(1,10:13,3)	(1,11:11,4)	···	(1,16:21,3)	(1,17:20,4)	···	(1,19:19,5)
GLN:	1		1	1		3	2		4	3		3
PSN:	1		1	1		3	1		3	1		1

(a)

Partition
$\langle label, regionCode, reCount, sid\rangle$

f	(1,1:23,1)	1	1
a	(1,2:8,2)	1	1
b	(1,3:6,3)	1	1
c	(1,4:4,4)	1	1

ReP Index
$\langle ancLabel, (reCount, sidRegion)\rangle$

Null	Null

(b)

Partition
$\langle label, regionCode, reCount, sid\rangle$

d	(1,5:5,4)	1	1
c	(1,7:7,3)	2	2
a	(1,9:14,2)	2	3
a	(1,10:13,3)	3	3

ReP Index
$\langle ancLabel, (reCount, sidRegion)\rangle$

f	(1,1:3)
a	(1,1:2)
b	(1,1:1)

(c)

Partition
$\langle label, regionCode, reCount, sid\rangle$

b	(1,11:11,4)	2	1
c	(1,12:12,4)	3	2
d	(1,15:22,2)	2	3
a	(1,16:21,3)	4	3

ReP Index
$\langle ancLabel, (reCount, sidRegion)\rangle$

f	(1,1:3)
a	(2,1:2)(3,1:2)

(d)

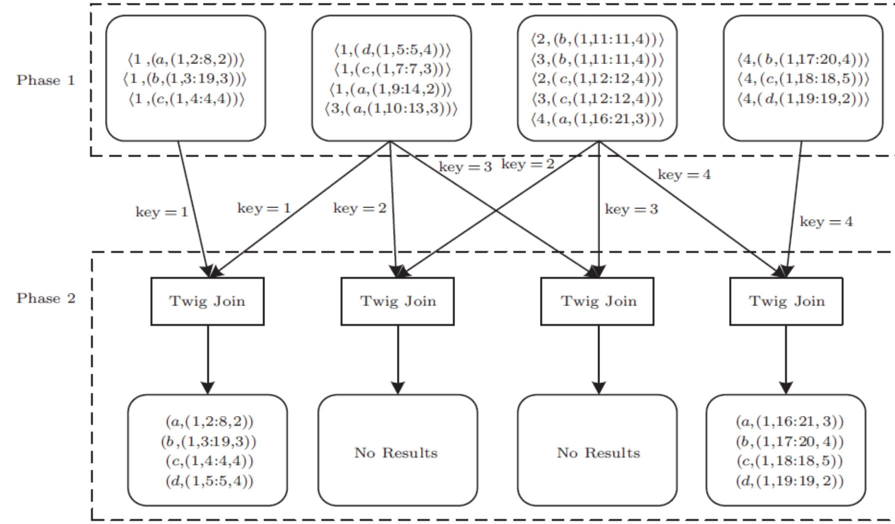
Partition
$\langle label, regionCode, reCount, sid\rangle$

b	(1,17:20,4)	3	1
c	(1,18:18,5)	4	1
d	(1,19:19,5)	3	1

ReP Index
$\langle ancLabel, (reCount, sidRegion)\rangle$

f	(1,1:1)
d	(2,1:1)
a	(4,1:1)

(e)



Example 4 (*Algorithm DisT2ReP*). Based on the ReP index in example 3, as to partition $F1$, the first node (f, (1, 1:23, 1), 1, 1) is not a match node of query root node a, and the ancestor array is empty, thereby we do not set the emit key of the node f. The next node (a, (1, 2:8, 2), 1, 1) is a match of query root node, thus we directly set its key, generate ⟨1, (a, (1, 2:8, 2))⟩, and save the 2-tuple of its region code and GLN (2:8, 1) into the ancestor array. The next node (b, (1, 3:6, 3), 1, 1) is not a match of query root node a, and we check the ancestor array and find its descendant (2:8, 1), thus we set its GLN as the key of this node b and (b, (1, 3:6, 3)) as its value to form ⟨1, (b, (1, 3:6, 3))⟩. Then we check the ReP index on $F1$, which is empty, thus we do nothing. Similarly, node (c, (1, 4:4, 4), 1, 1) generates tuple ⟨1, (c, (1, 4:4, 4))⟩. In the same way, on partition $F2$, each of nodes (d, (1, 5:5, 4), 1, 1), (c, (1, 7:7, 3), 2, 2), (a, (1, 9:14, 2), 2, 3) and (a, (1, 10:13, 3), 3, 3) generates a tuple separately. On partition $F3$, each of nodes (b, (1, 11:11, 4), 2, 1) and (c, (1, 12:12, 4), 3, 2) generates two tuples separately, and node (a, (1, 16:21, 3), 4, 3) generates a tuple. On partition $F4$, each of all the nodes generates a tuple. With the tuples generated in phase 1, the tuples with the same keys are distributed to the same node, where holistic twig algorithm is executed to generate final results

Performance Guarantees

- ***Visit Times***. Each machine traverses its own partitions only once.
- ***Parallelism***. All the local computations are executed in parallel on the machines with corresponding partitions.
- ***Computation Cost***. The local computation cost on each partition F_i is $O(|S^i_Q||ReP_i|)$, where $|S^i_Q|$ is the number of match nodes on F_i and $|ReP_i|$is the size of ReP index on F_i.
- ***Communication Cost***. The total communication cost is the size of match nodes having contributions to the final results, which is O(|*results*|) in the best case, and O($|S_Q|$) in the worst case. |*results*| is the size of final results and $|S_Q|$ is the size of streams with labels in the query pattern, i.e., the number of match nodes.

Implementation

- DisT2ReP can be implemented in any distributed computing framework, including MapReduce of Hadoop, due to the following advantages:
    1. Supporting any distributed storage mechanism, regardless of how the XML data are partitioned;
    2. Enabling to process twig queries without priori knowledge of the query patterns, so that it does not need to repartition and restore the large-scale data each time when a new query comes;
    3. Requiring no communication among computing nodes in each of the two phases.

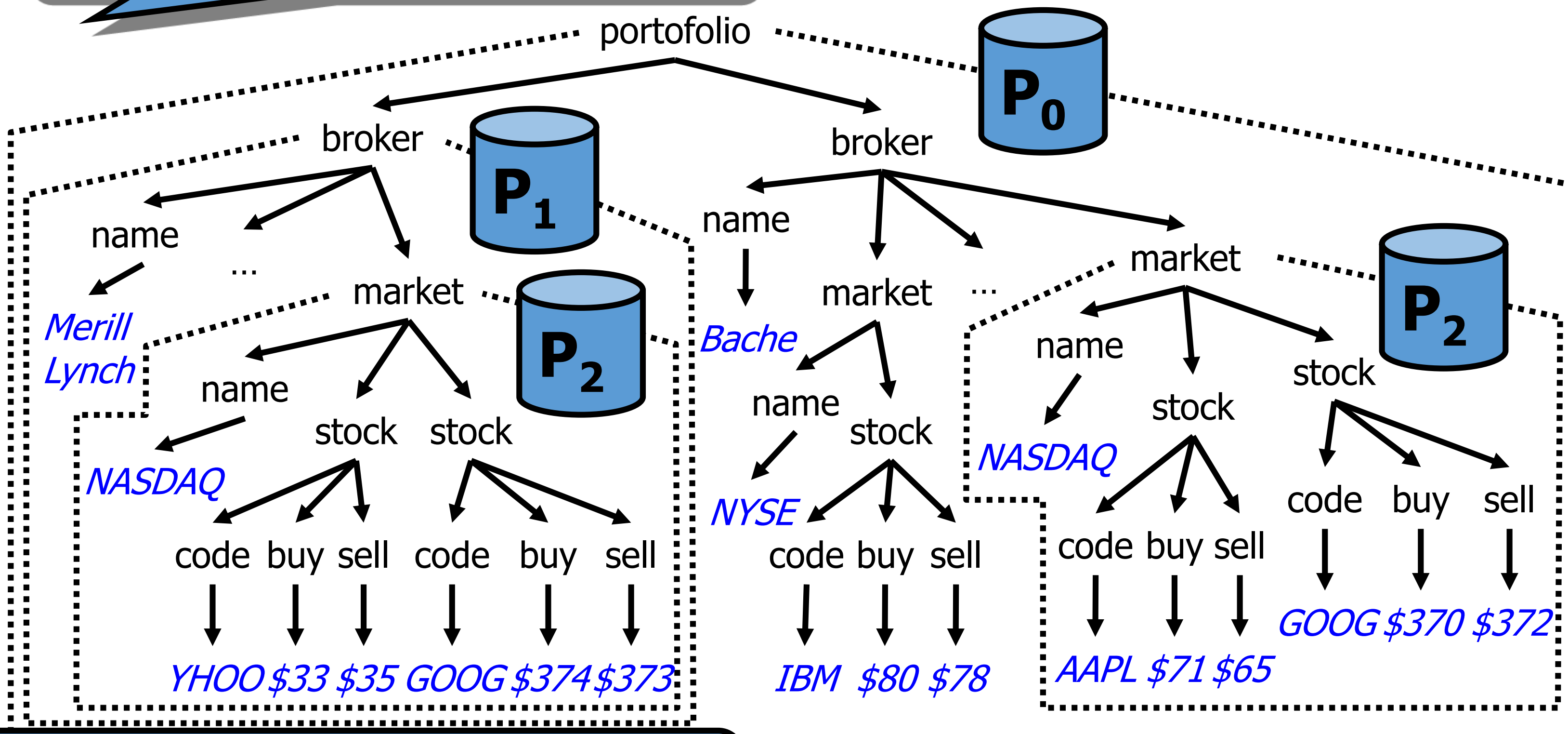
Using Partial Evaluation in Distributed Query Evaluation

Peter Buneman, Gao Cong, Wenfei Fan,
Anastasios (Tasos) Kementsietsidis

VLDB’06

Cutting Down Trees…

Tell me when GOOG stock sells for 376:
[//stock[code = "GOOG"$\wedge$sell = 376]



Let's do a Depth-first traversal. We visit:
$P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_1 \rightarrow P_0 \rightarrow P_2 \rightarrow P_0$

Status report…

- We have XML Trees arbitrarily **fragmented** and **distributed**
- We want to execute **Boolean Xpath** queries Q = [q] over the fragmented trees.

$$q := p \mid p/\text{text}()=\text{str} \mid \text{label}() = A \mid \neg q \mid q \wedge q \mid q \vee q$$

$$p := \varepsilon \mid A \mid * \mid p//p \mid p/p \mid p[q]$$

Lessons learned:

- We want to visit each peer only **once**, irrespectively of the number of (tree) fragments it stores.
- We want to **minimize** communication costs. Ideally, no fragment data should be send while evaluating a query.
  Our motto: Send **processing** to data **NOT** data to processing

Partial Evaluation

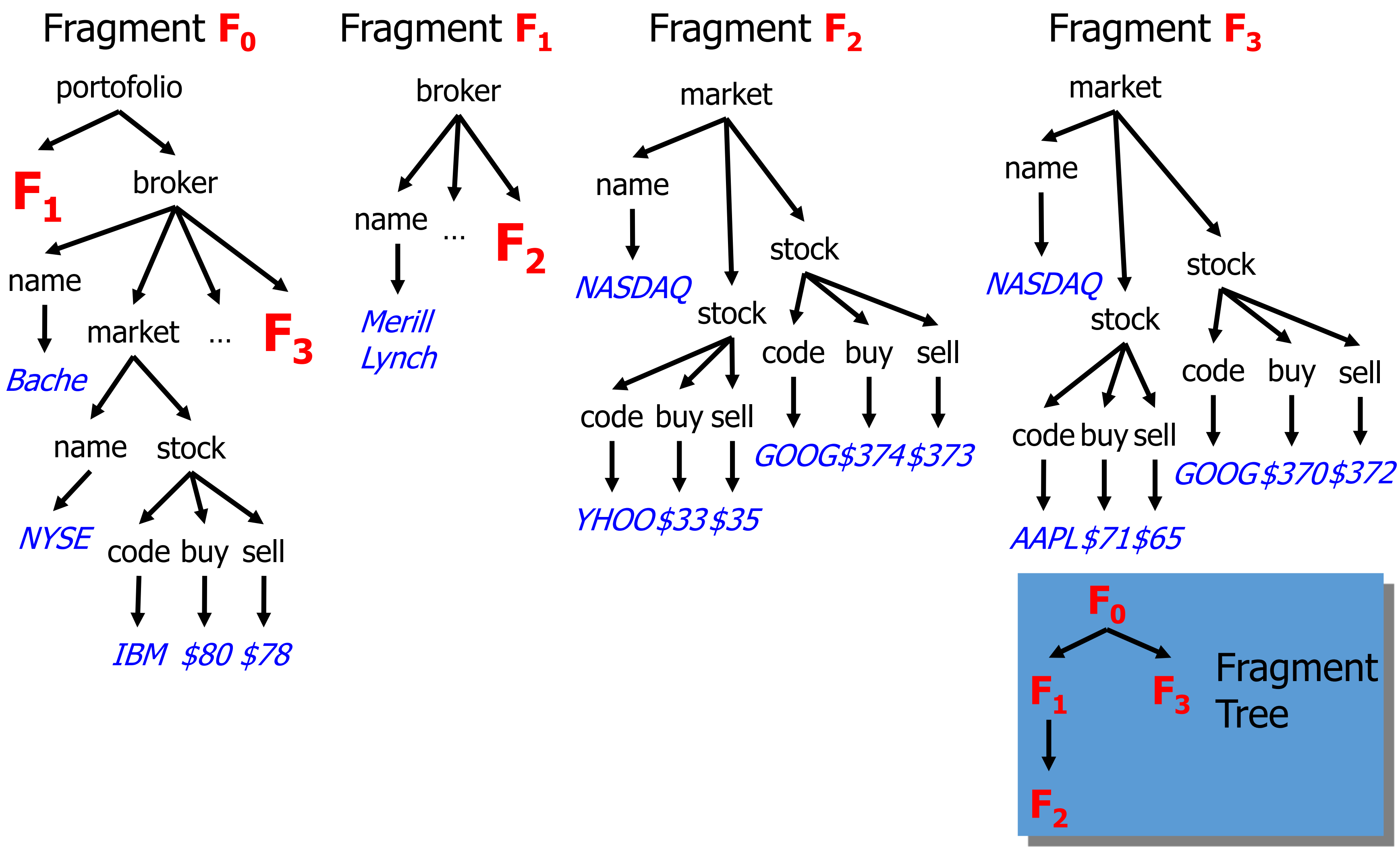
Consider a function $f(s, d)$ and part of its input, say s.

Then, **partial evaluation** is to specialize $f(s, d)$, i.e.,

to perform the part of f's computation that depends only on s.

This generates a **residual** function $g(d)$ that depends only on d.

Tree Fragments

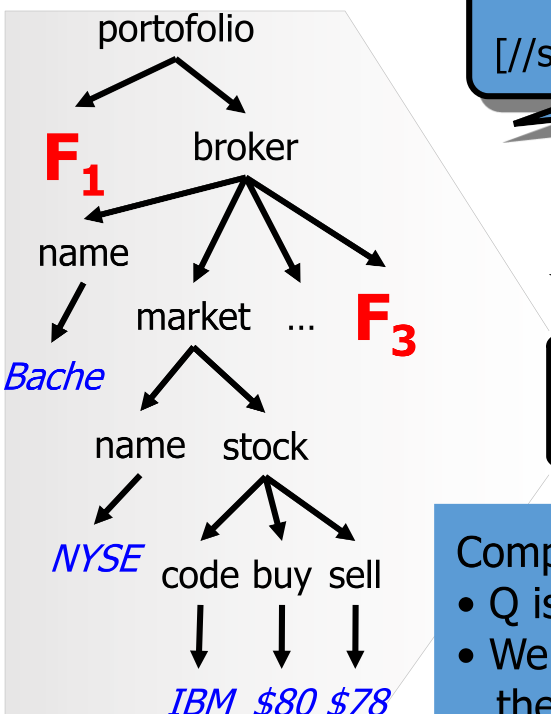


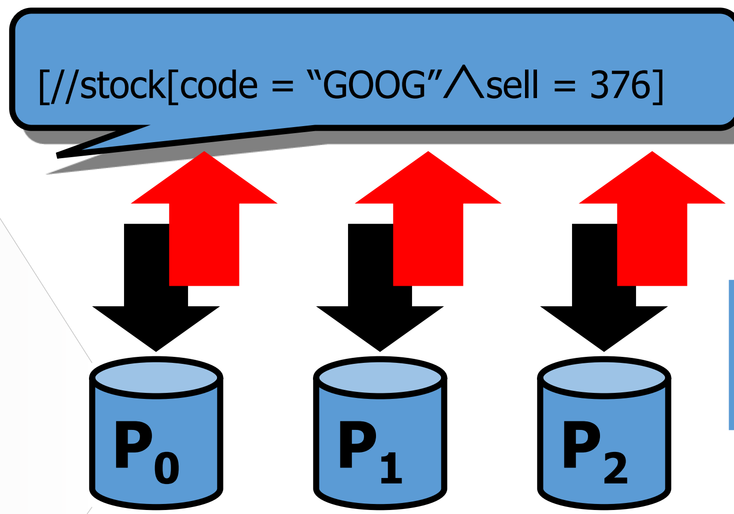
Tree Fragments

- Fragmentation
  - No constraints on the fragmentation
  - No constraints on how the fragmentation are distributed
- Storage
  - Root fragment and sub-fragment
  - Virtual node
  - Source tree

Distributed Partial Evaluation

- **Main idea**: Given a query *Q*, send *Q* to every peer holding a fragment





Answer of Q:Computed by solving a **linear** system of Boolean equations

P_2 has **two** fragments but is only visited **once**

Compute **Partial Answers** (**Boolean formulas**):
- Q is evaluated **bottom-up**
- We use **Boolean variables** for the evaluation of fragment nodes

Query Evaluation

Query Representation:

Q = [//stock[code = "GOOG"$\wedge$sell = 376]]

q_4: //q_3

q_3: stock[q_2]

q_2: */$q_0$$\wedge$*/$q_1$

q_0: code = "GOOG" q_1: sell = 376

Q = <q_0, q_1, q_2, q_3, q_4>

Query Evaluation Example 1:

<0, 0, 0, 0, 1> market

<0, 0, 1, 1, 1> stock ...

<1, 0, 0, 0, 0> code buy sell <0, 1, 0, 0, 0>

GOOG $370 $376

Query Evaluation Example 2:

<0, 0, 0, 0, x_1> market

<0, 0, x_1, x_1, x_1> stock ...

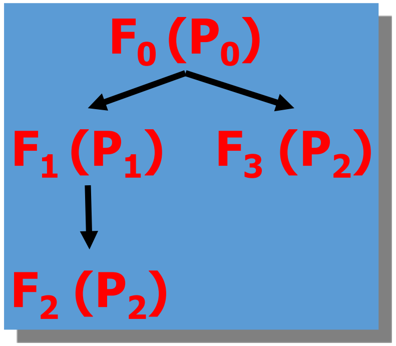
<1, 0, 0, 0, 0> code buy **F** <x_0, x_1, x_2, x_3, x_4>

GOOG $370

The ParBoX Algorithm

Three stages

- Stage 1: Querying peer **P_Q** sends query Q to all peers having a fragment (use the fragment tree to identify all such peers)
- Stage 2: Evaluate Q, in parallel, over each fragment **F_i** in peer **P_j**
- Stage 3**:** Collect partial answers in **P_Q** and compute the answer to Q.



Key considerations/concerns:

- (Total/Parallel) Computation costs.
- Communication costs.
- Level of fragmentation.

ParBoX comes in flavors:
- **HybridParBoX**
- **FullDistParBoX**
- **LazyParBoX**

Analysis of Algorithms

Communication costs are **LOW** and independent of *T* (the data)

Algorithm	Visits/Peer	Computation		Communication
NaiveCentralized	1	$\mathcal{O}(\lvert Q\rvert\ \lvert T\rvert)$		$\mathcal{O}(\lvert T\rvert)$
NaiveDistributed	$card(S_i)$	$\mathcal{O}(\lvert Q\rvert\ \lvert T\rvert)$		$\mathcal{O}(\lvert Q\rvert\, card(T))$
ParBoX	1	Tot	$\mathcal{O}(\lvert Q\rvert\ (\lvert T\rvert + card(T)))$	$\mathcal{O}(\lvert Q\rvert\, card(T))$
		Par	$\mathcal{O}(\lvert Q\rvert\ (\max_{Pj}\lvert F_{Pj}\rvert + card(T)))$	
HybridParBoX	1	Tot	$\mathcal{O}(\lvert Q\rvert\ \lvert T\rvert)$	$\mathcal{O}(\lvert T\rvert)$
		Par	$\mathcal{O}(\lvert Q\rvert\ (\max_{Pj}\lvert F_{Pj}\rvert + card(T)))$	
FullDistParBoX	$card(S_i)$	Tot	$\mathcal{O}(\lvert Q\rvert\ (\lvert T\rvert + card(T)))$	$\mathcal{O}(\lvert Q\rvert\, card(T))$
		Par	$\mathcal{O}(\lvert Q\rvert\ (\max_{Pj}\lvert F_{Pj}\rvert + card(T)))$	
LazyParBoX	$card(S_i)$	Tot	$\mathcal{O}(\lvert Q\rvert\ (\lvert T\rvert + card(T)))$	$\mathcal{O}(\lvert Q\rvert\, card(T))$
		Par	$\mathcal{O}(\lvert Q\rvert\ card(T) \max_T\lvert F_i\rvert\ )$	

$card(S_i)$ = # of fragments in peer P_i

$card(T)$ = # of fragments of tree T. Note that $card(T) \leq \lvert T\rvert$

$\lvert F_{Sj}\rvert$ = sum of fragments (sizes) in peer P_j

Distributed Query Evaluation with Performance Guarantees

Gao Cong, Wenfei Fan, Anastasios Kementsietsidis

SIGMOD'07

ParBoX

- Algorithm based on partial evaluation, which evaluates **Boolean xml queries** over a fragmented tree that is distributed over a number of different sites

- Partially evaluates the whole query Q, in parallel, over each fragment of the tree.

- Partial answers are all collected to a single coordinator site and are composed resulting in the final answer to Q.

Parallel XPath (PaX3)

- Evaluation algorithm, based on partial evaluation for **generic data-selecting XPath queries**. Guarantees:

1. Max 3 visits per site
2. Parallel query processing
3. Total computation comparable to the best-known centralized algorithm
4. Total network traffic determined by the size of:
    - the query
    - query answer
    - not the xml tree

Example

- Q = client [country/text() = “us”] /broker [market/name/text() = “nasdaq”]/name

- normalize(Q) = client/ ε [country/ ε [text()=“us”]] /broker/ ε[market/name/ ε [text() = “nasdaq”]]/name

- SVect(Q) = [q1, q2, q3] where
  - q1 = client, q2 = q1/broker, q3 = q2/name

- QVect(Q) = [q1, q2, q3, q4, q5, q6, q7, q8, q9], where
  - q1 = country, q2 = [text()=“us”], q3 = q1/ε [q2],
    q4 = * / ε [q3], q5 = name, q6 = [text()=“nasdaq”],
    q7 = q5/ ε [q6], q8 = market/q7, q9 = * / ε [q8]

Three stages of PaX3

- Each stage → single visit of a site holding tree fragments

1. Partially **evaluate the qualifiers** of query Q.

   At the end for each node we know:
   - the actual value of each qualifier or
   - a Boolean formula whose value is yet to be determined

2. Partially **evaluate the selection part** of query Q.

   At the end for each node we know:
   - whether or not the node is part of the answer of query Q
   - that the node is a candidate to be part of the answer

3. Determine **which candidate answer nodes are true** answer nodes → all nodes belonging to the answer of Q are transmitted to site S

Analysis

- **Communication cost** : $O((|Q|\ |F_T|) + |ans|)$ (optimal)
  - cost of transmitting our query over the various sites +
  - cost of retrieving the actual answers to our query
- **Total computation cost** : $O(|Q|\ |T|)$
  - at each node v of F_j at most $O(|Q|)$ operations are performed
  - total computation for each fragment is $O(|Q|\ |F_j|)$
- **Parallel computation** cost: $O(|Q|\ max_{Si}\ |F_{Si}|)$
  - $|F_{Si}|$: total size of the fragments in site Si
- **Correctness**: correct answer Q(T) on any xml tree T no matter how T is fragmented and distributed

PaX2

- Two stages and max two visits of each site

- **Combine the first two stages of PaX3** into a single stage
  - evaluation of qualifiers + evaluation of selection paths

1. Querying site S_Q makes a remote procedure call to all the sites holding fragments
2. At each such site, combines the partial evaluation of selection paths with that of qualifiers, over a fragment Fj .
3. The procedure performs a top-down traversal of fragment Fj .
4. At each node v of Fj , two types of computation are performed: a pre-order computation and a post-order computation.

End of Chapter 9