

Efficient Handling of Positional Predicates within XML Query Processing

Zografoula Vagena (UC Riverside)

Nick Koudas (University of Toronto)

Divesh Srivastava (AT&T Labs-Research)

Vassilis J. Tsotras (UC Riverside)

Motivation: Positional Predicates

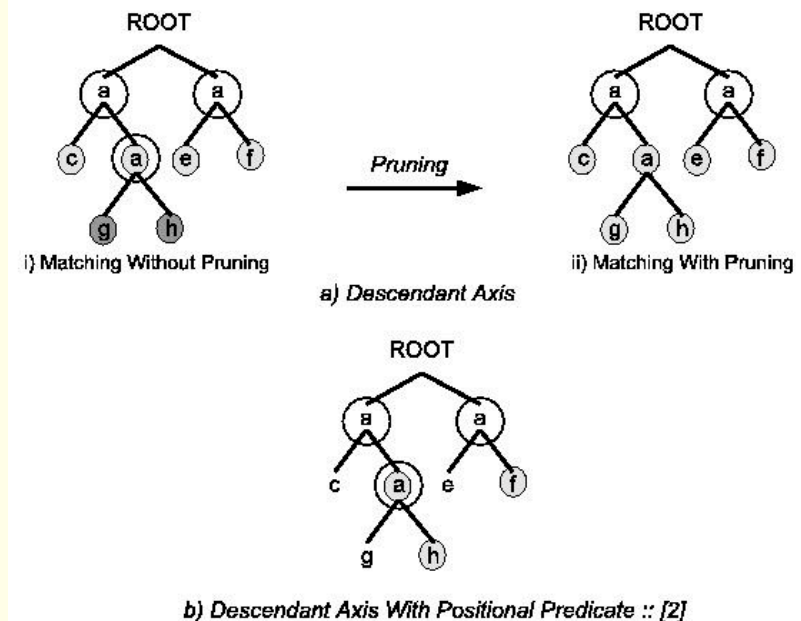
- XML data representation: **ordered**, rooted tree
 - Important especially in the document-centric view of XML
 - XPath, XQuery naturally express order-sensitive queries
- Order-sensitive XPath queries
 - `// article / child::author [position() <= 2]`
 - `// article / section [title = "Related Work"] / following-sibling::section [1]`
- Goal: Efficient evaluation of queries with positional predicates

Related Work

- Navigation-based approaches [Saxon, Xalan]
 - Simple, best for small data sets that fit in main memory
 - Outperformed by set based approaches for large data sets
- Queries in relational databases [TVB+02, G02]
 - RDBMS performs well, can benefit from being XML aware
- Set-based approaches
 - Initially for child, descendant axes [ZND+01, BKS02, GvKT03]
 - Recently for other axes [VKST05, SK05]
 - Scalable, often with provable linear worst-case complexity

Extending Staircase Join [GvKT03]

- Staircase join has linear input + output complexity
- Goal: extend staircase join to handle positional predicates
- Example: illustrates difficulty
- Decision: extend structural and holistic joins instead



Contributions of Paper

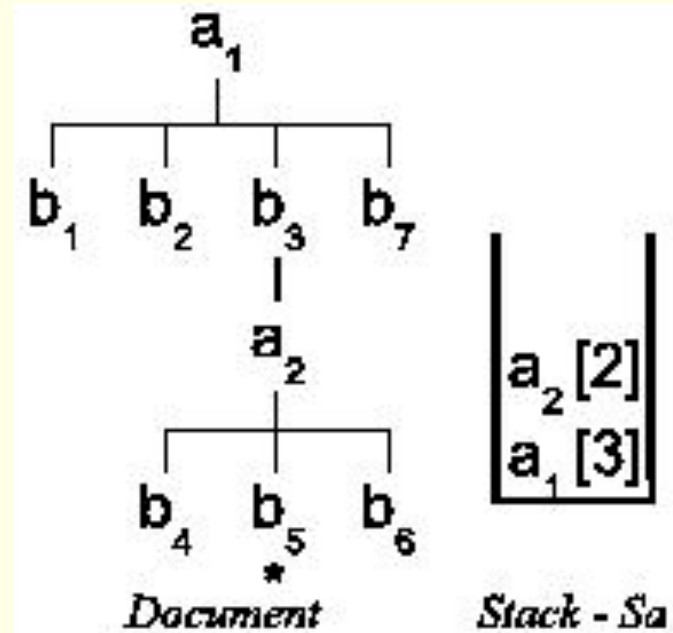
- Extend “structural join” algorithm for single step queries
 - Worst-case time complexity: $O(|\text{Input}| + |\text{Output}|)$
 - Worst-case space complexity: $O(\text{height} * \text{fanout})$ for following-sibling, $O(\text{height})$ for child
- Extend “holistic” algorithms for multiple step queries
 - Branching paths with “forward” + “backward” axes
- Experiments: superiority on real and benchmark data
 - Comparison with relational engines
 - Comparison with DOM based XPath query processors

Single Step: Child Axis

- Query pattern: `a / child::b [n]`
- Solution extends stack-based structural join algorithm [AJK+02]
 - Key idea: maintain a counter with each context (“a”) element that tracks the number of relevant (“b”) children
 - Property: an “a” element accessed before any “b” child
 - Property: sibling “b” elements accessed in document order

Single Step: Child Axis

- Stack S_a empty
- a_1 seen, put on S_a
 - b_1 seen $\rightarrow a_1[1]$
 - b_2 seen $\rightarrow a_1[2]$: output
 - b_3 seen $\rightarrow a_1[3]$
- a_2 seen, put on S_a above a_1
 - b_4 seen $\rightarrow a_2[1]$
 - b_5 seen $\rightarrow a_2[2]$: output



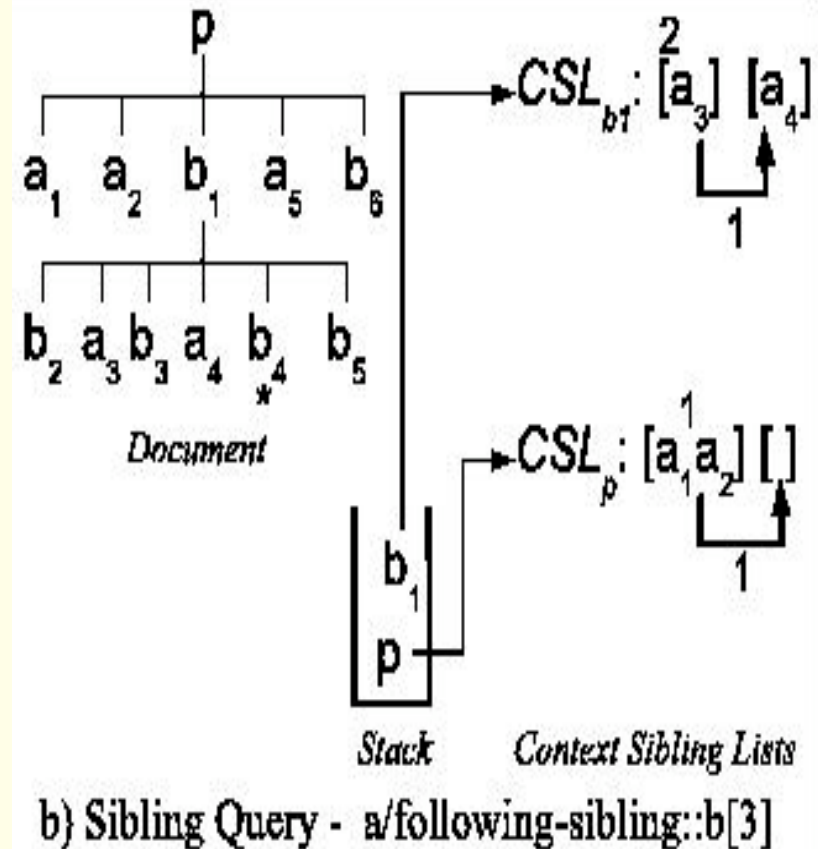
a) Child Query - $a/\text{child}::b[2]$

Single Step: Following-Sibling Axis

- Query pattern: `a / following-sibling::b [n]`
- Solution extends yet another structural join algorithm [VKST05]
 - Key idea: maintain context (“a”) elements in context sensitive lists (CSL), using a counter with each “a” element that tracks the number of relevant (“b”) following-siblings
 - Property: an “a” accessed before any “b” following-sibling
 - Property: sibling “b” elements accessed in document order

Single Step: Following-Sibling Axis

- a_1 seen: put parent p on S , add a_1 to CSL_p
- a_2 seen: add a_2 to CSL_p
- b_1 seen: increment CSL_p group counter, close group
- a_3 seen: put parent b_1 on S , add a_3 to CSL_{b_1}
- b_3 seen: increment CSL_{b_1} group counter, close group
- a_4 seen: add a_4 to CSL_{b_1}
- b_4 seen: increment CSL_{b_1} group counter, close group

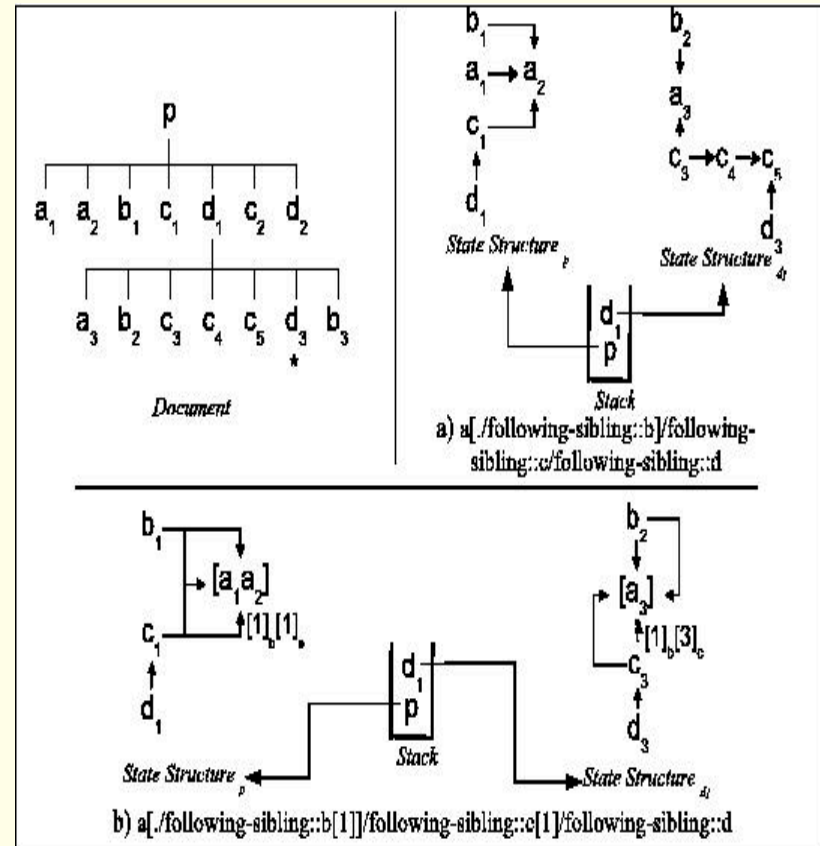


Single Step: Theorem

- Given a single step query of the form $a / \text{axis}::b [n]$, where axis is “child” or “following-sibling”:
 - The algorithm is correct
 - Worst-case time complexity: $O(|\text{Input}| + |\text{Output}|)$
 - Worst-case space complexity: $O(\text{height} * \text{fanout})$ for following-sibling, $O(\text{height})$ for child
- NB: $|\text{Output}|$ for $a / \text{axis}::b [n]$ is smaller than for $a / \text{axis}::b$

Multiple Steps: Following-Sibling

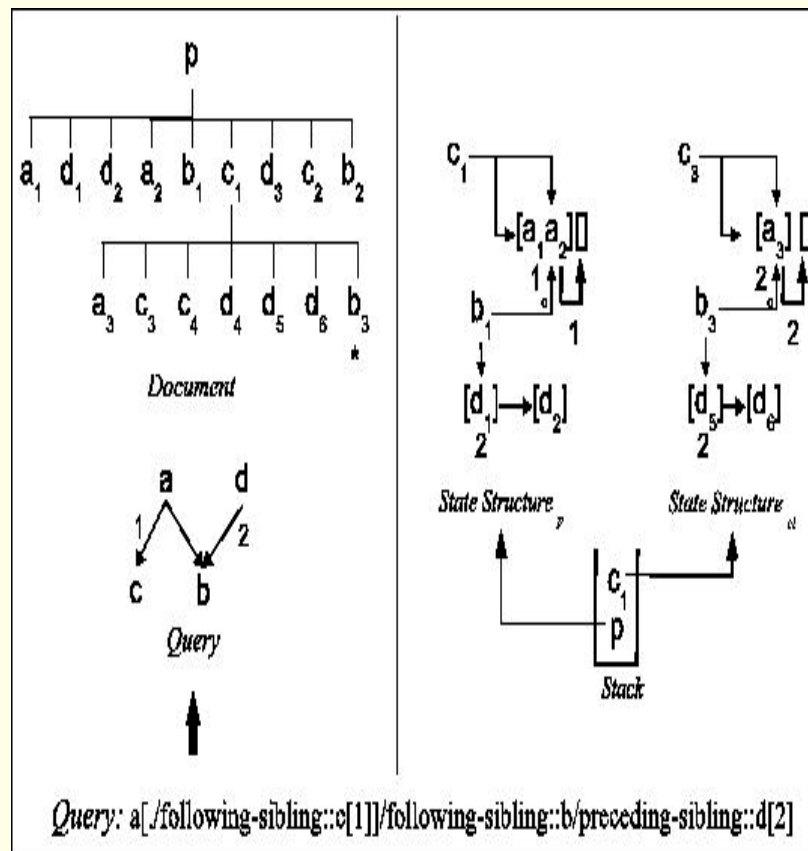
- Query is a tree structure
- Observation: result instance involves only sibling nodes
- Holistic approach:
 - State per sibling group
 - State structure has one CSL per “step” parent
 - Multiple sibling groups simultaneously active



Multiple Steps: Preceding-Sibling

- Forward and Backward axes of the same type
 - Preceding-sibling → following-sibling
 - Tree → Special DAG

- Deal with “join” nodes:
 - Check CSL in **each** query parent
 - Positional predicates treated slightly differently

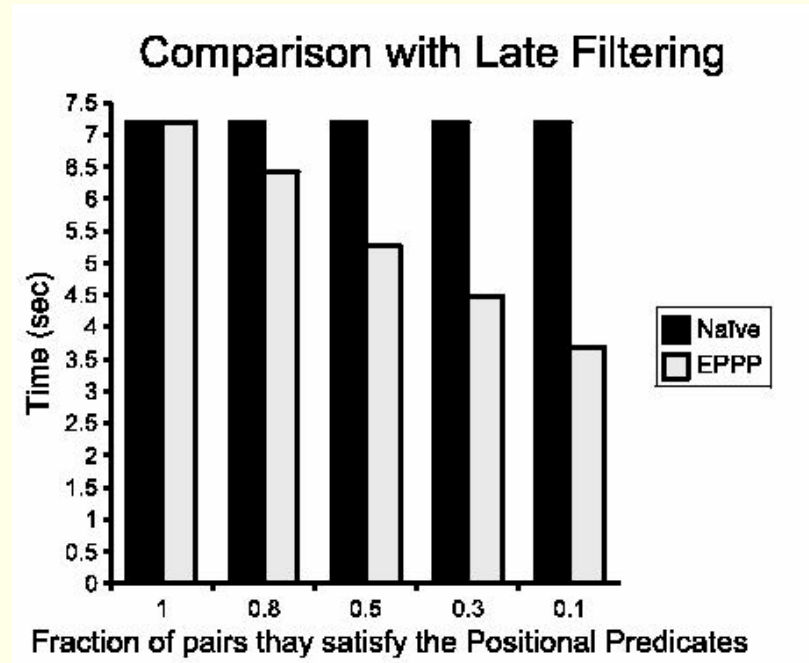


Experimental Setup

- Algorithms in C++ on top of a native storage manager
- Experiments conducted on a 2.6 GHz Pentium 4, 512 MB
- Code compiled with GNU compiler 3.2.2, on RedHat Linux 9
- Measurements:
 - Using 8K pages, 100 block buffer cache
 - Total execution time reported with hot cache

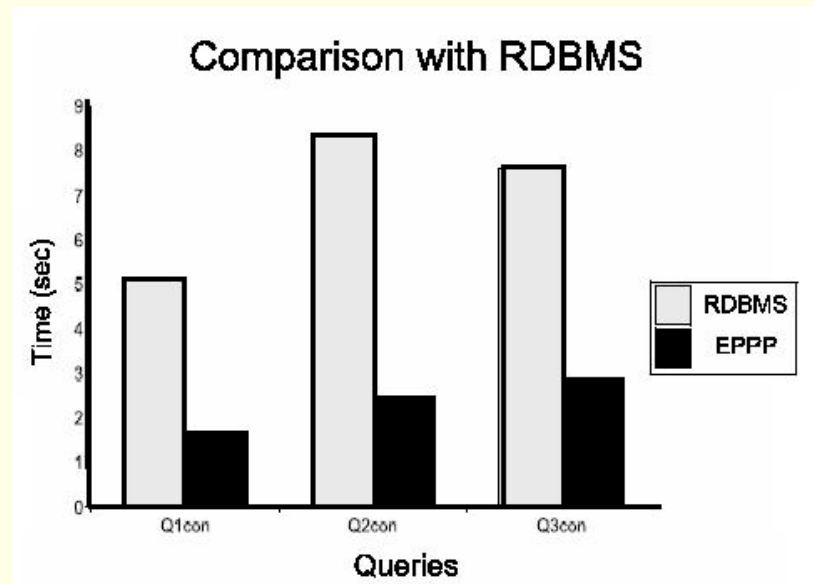
Importance of Early Filtering

- Goal: demonstrate benefits of algorithm over late filtering
- Query: a / following-sibling::b [position() <= n], vary n
- Data: 100K “a”, 1M “b”
- Results:
 - More selectivity → better
 - Very low overheads



Comparison with RDMBS

- Goal: demonstrate benefits of algorithm over RDBMS
- Queries: step, path, twig
- Data: XMark, 1G (text)
- Results:
 - Generated SQL complex, more optimization errors



Comparison with XPath Engines

- Goal: demonstrate benefits of algorithm over Xalan, Saxon
- Queries: step, path, twig
- Data: Shakespeare's plays
- Results:
 - Our algorithm is efficient in main memory too

	Xalan (with)	Xalan (w/o)	Saxon (with)	Saxon (w/o)	EPPP (with)	EPPP (w/o)
Q4	0.20	0.21	0.20	0.12	0.08	0.05
Q5	0.41	0.41	0.18	0.14	0.05	0.07
Q6	0.22	0.21	0.21	0.11	0.04	0.04

Conclusions

- Goal: Efficient evaluation of queries with positional predicates
- Structural and holistic join algorithms can be extended
 - Intermediate state suffices to handle positional predicates
 - Provably efficient for single step queries
- Experiments: superiority on real and benchmark data
- Takeaway: Scalable solution for positional predicates