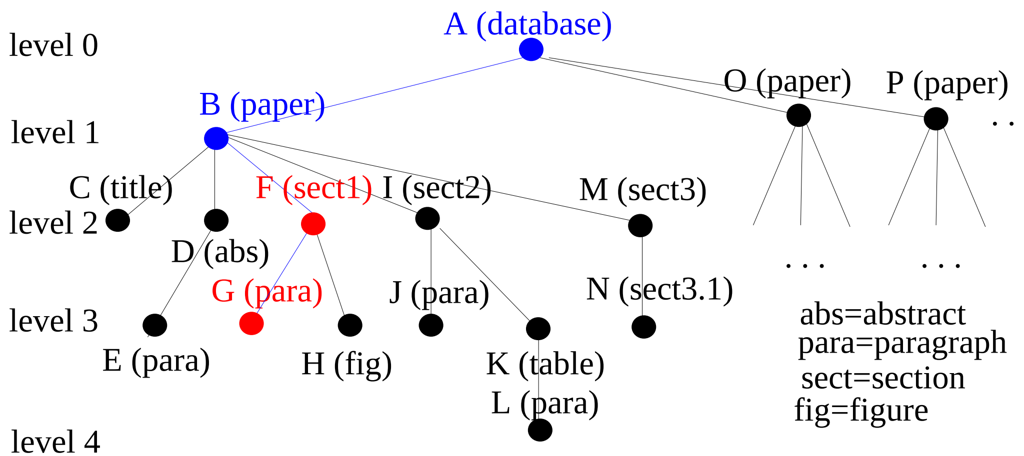
Efficient XML Path Expression Processing Techniques

Kaiyang Liu

Supervisor: Frederick H. Lochovsky

Hong Kong University of Science and Technology

Problem Definition



- An XML tree for a database of articles
- Query "//section1//paragraph"
  - Node pair <*F*, *G*>
  - Other <section1, paragraph> pairs
- Simple Containment Query

- Need efficient method to determine if one node N_2 is a child/descendent of another node N_1
- Pre-order traversal based encoding [ZND+01]
- The general form of containment queries is **"//NA_1{$cond_1$}//NA_2{$cond_2$}//...//NA_k{$cond_k$}"** (NA_i denotes a node name, and $cond_i$ refers to an optional condition that applies on the attributes of NA_i)

- "//paper {year = 2000} // section1 {*wrdcnt* ≤ 800}// paragraph": all paragraphs that appear in the first section (with no more than 800 words) of any paper published in 2000.

THE INTUITION

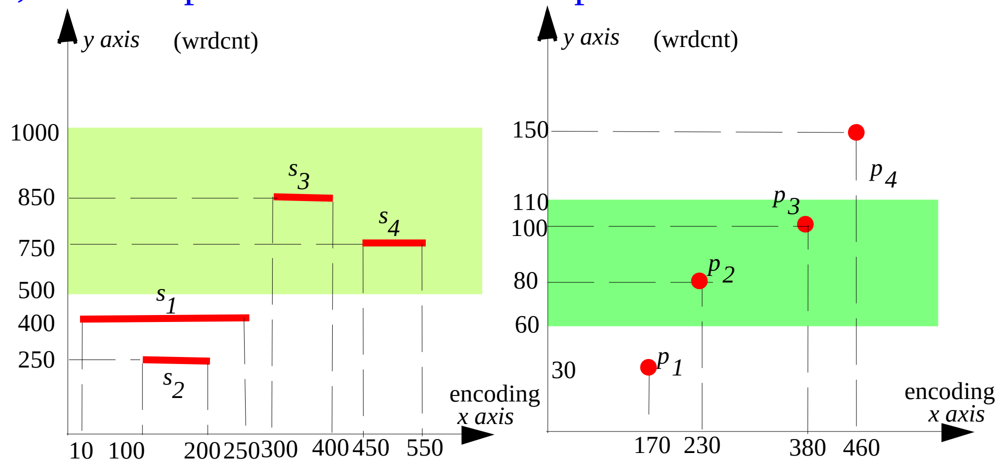
Indexing attributes in XML documents requires a multi-dimensional structure, since one dimension should store the traversal order to efficiently process containment.

Existing Approaches

- Graph traversal based approaches
  - Simplify the original XML trees
  - Traversal method: Bottom-up, Top-down, and hybrid
  - Poor performance, due to possibly exponential number of children
- Pre-order traversal encoding based approaches
  - MPMJ, εε-join, and SJ
  - Much more efficient than graph traversal based ones
  - *false-hit* problem

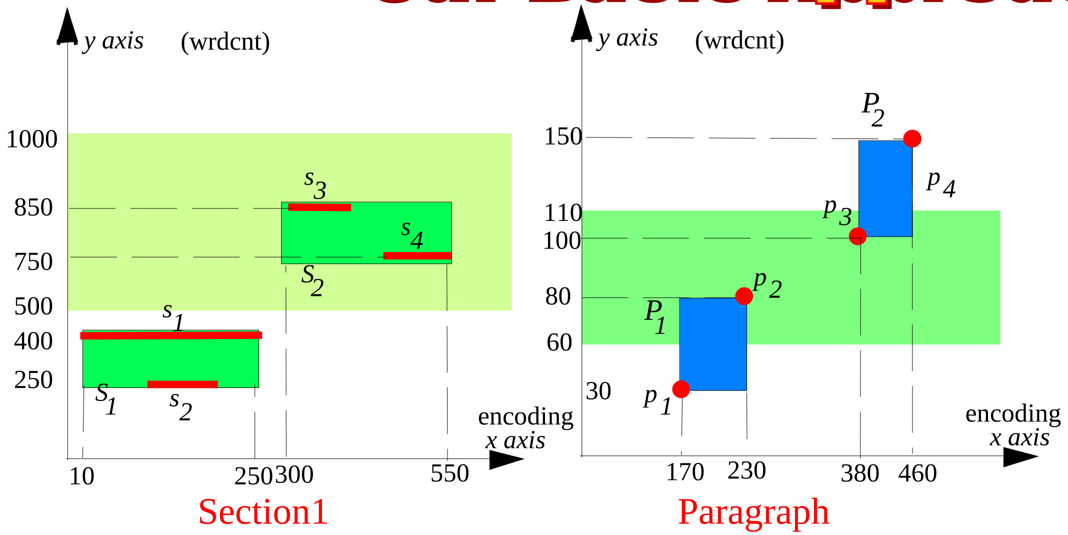
Our Basic Approach – CJ (Containment Join)

- A general containment query requires simultaneous evaluation on **multiple dimensions**:
  - **The traversal encoding for node containment**
  - **The attribute values for range conditions**
- One-dimensional structures, such as B-trees, cannot optimize search in multiple dimensions.
- Obtain the pre-order encoding [*I*, *J*] for each element
- Transform each element into a 2D line segment (or point) as follows:
  - *x*-axis: pre-order encoding [*I*, *J*]
  - *y*-axis: attribute value



Section1 Elements Paragraph Elements

Our Basic Approach – CJ Cont'd



- Use an R-tree to index the transformed intervals and points
- "//NA_1{$attr_1 \in [ql_1, qh_1]$}//NA_2{$attr_2 \in [ql_2, qh_2]$}"
  - $attr_1(_2)$ is an attribute of nodes with names $NA_1(_2)$
  - $[ql_1(_2), qh_1(_2)]$ is a range condition on $attr_1(_2)$
- If a pair of intermediate nodes (N_1, N_2) from the respective R-trees contains qualifying elements, their MBRs must satisfy the following *traversal conditions*:
  - ***N_1.y*-range (i.e., the *y*-range of *N_1.MBR*) intersects [ql_1, qh_1]**
  - ***N_2.y*-range intersects [ql_2, qh_2]**

- "//section1 {*wrdcnt* ∈ [500,1000]} // paragraph{*wrdcnt* ∈ [60,110]}"

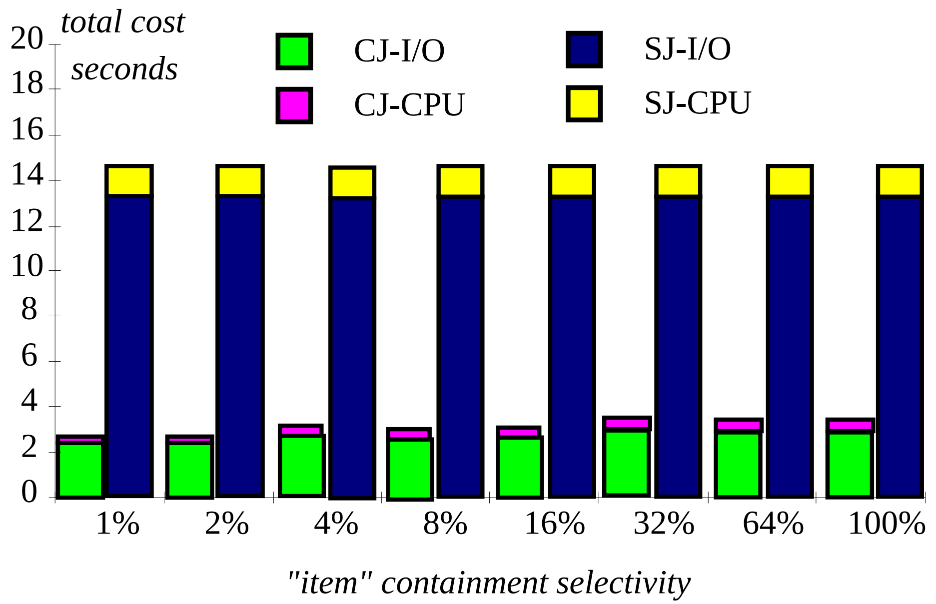
The shaded regions in each tree show the query conditions. The sub-tree of S_1 does not contain any qualifying entries because the S_1.*y*-range does not intersect the query region [500, 1000]. Thus, the pair <S_1, P_1> will not be followed, even though the *x*-ranges of the nodes intersect. In fact, it can be verified that only the sub-trees of S_2, P_2 need to be explored.

- In general, CJ performs joins by testing only the *x* coordinates for intersection, and using the *y* coordinates to restrict the number of nodes.
- It constitutes a combination of RJ and window query processing.

Experiment – General Settings

- Use [XMark] to create synthetic data sets.
- *Cardinality(item)=10,000*
- *Cardinality(mail)=100,000*
- Simple containment query: //item//mail
  - CS_{item}: the percentage of *item* nodes that contain *mail* nodes in their sub-tree
  - CS_{mail}: shows the percentage of *mail* nodes that reside in the sub-trees of *item* nodes.
- General containment query: //*item* [q_1 < quantity] // *mail* [y_1 < year < y_2]
  - AS_{item}: The percentage of *items* whose *quantity* value is larger than the query condition q_1
  - AS_{item}: The percentage of *mail* whose *year* value lies in between y_1 and y_2

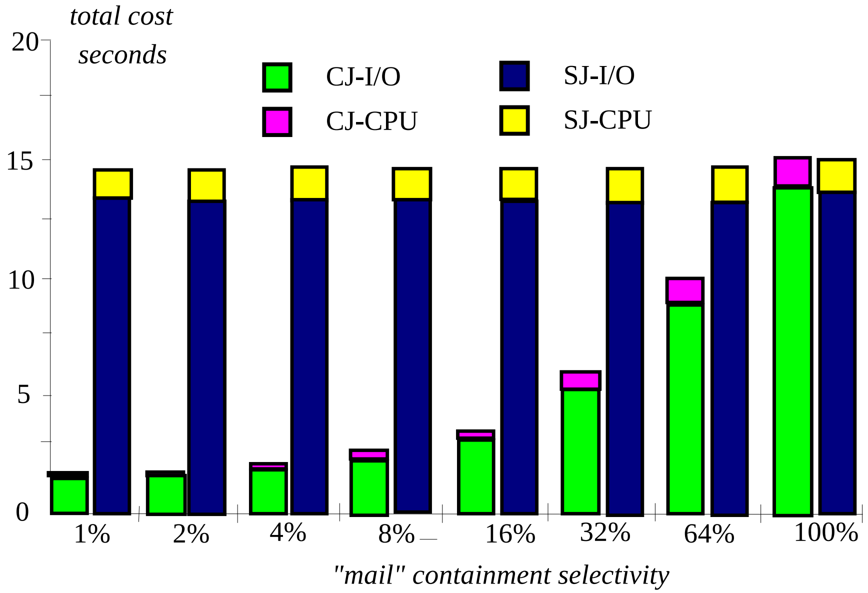
Experiments on Simple Containment Query



- Compare CJ with SJ
- Fix CS_{mail} = 10%
- As the cardinality of *mail* is larger than that of *item* by an order of magnitude, different values of CS_{item} do not have much influence on the performance of either CJ or SJ

CJ outperforms SJ under all cases

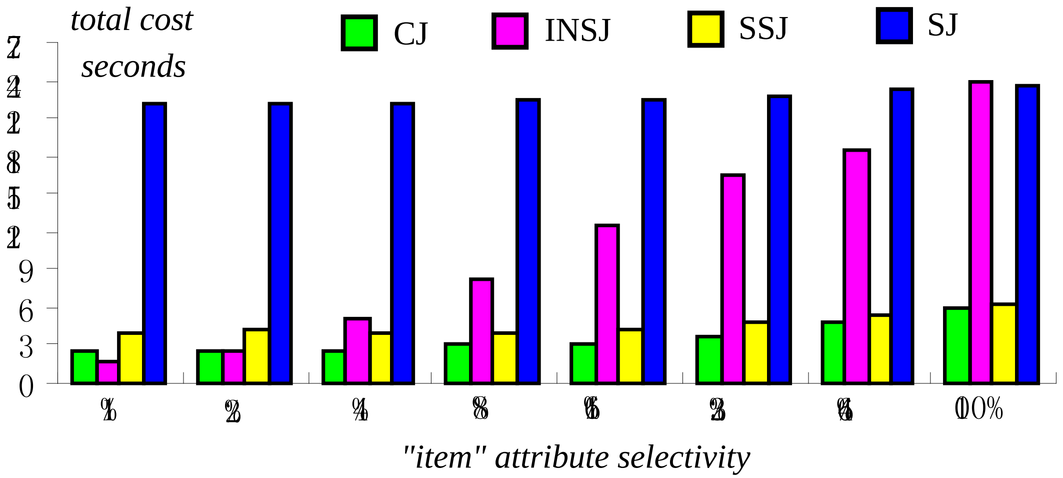
Experiments on Simple Containment Query Cont'd



- Fix CS_{item} to 10%
- The cost of SJ is the same as in the previous experiment
- The cost of CJ now increases with the percentage of qualifying *mail* nodes

The performance of the two algorithms is comparable only when CSmail = 100%

Experiments on General Containment Query



- CS_{item} = CS_{mail} = 100%, and AS_{mail} = 10%
- INSJ: *Index nested loop SJ*, first retrieve //*item* [q_1 < quantity], then evaluate //*item* [q_1 < quantity]//mail.
- SSJ: *Sorting SJ*, first retrieve //*item* [q_1 < quantity] and *mail* [y_1 < year < y_2], then sort and join them

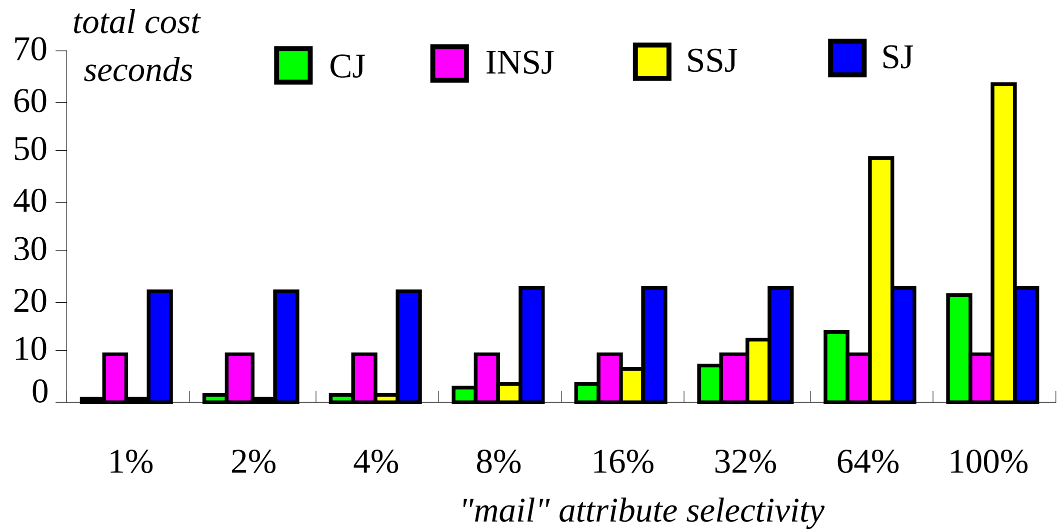
CJ is the best under most cases

Experiments on General Containment Query Cont'd

- $CS_{item} = CS_{mail} = 100\%$
- $AS_{item} = 10\%$

- The cost of SSJ grows with AS_{item} due to the high cost of external sorting as the qualifying *mail* elements increase.
- The performance of INSJ is stable since the cost of the first two steps does not depend on AS_{mail}.

CJ is rather stable



Future Work

- **Construct a cost model to finely tune the algorithms' parameters**
- **Some problems of pre-order traversal encoding-based approaches**
  - **Pre-order traversal encoding cannot give the path between two elements**
  - **The cost increases rapidly when the length of a path expression increases**
- **Investigate new query processing techniques**
  - **The cost of the new technique would roughly remain constant no matter how much the length of an XML path expression increases, and**
  - **Be able to give the exact path instances between any pair of XML elements**
- **Explore new XML query types**