SMVS
Page No.
Date : / /

# XML Query Evaluation

* XML Fragmentation:

=> XML Fragmentation refers to the process of breaking down large XML Document, into smaller.

Fragmentation is a Process of breaking or being broken into Fragment.

Fragmentation can improve the scalability of the XML Document because we can easily share and store the whole XML Document into the small Fragment.

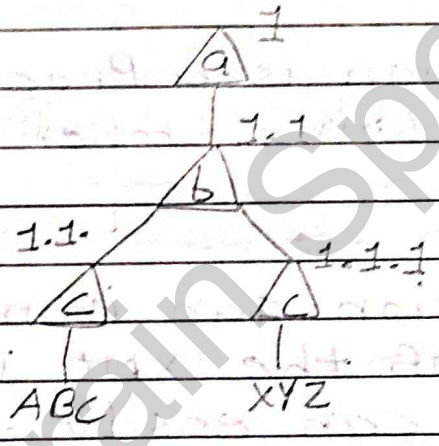
There are Two Types of Fragmentation

1) Horizontal : Fragmentation based on selection operators and predicates.

2) Vertical : Fragmentation based on a partitioning of the set of elements type in a Schema.

Ex.
```
<a>
   <b>
      <c> ABC </c>
      <d> XYZ </d>
   </b>
</a>
```
XML Data

=> XML Fragment tree with XFL.




=> XML Fragment with XFL

```
<Fragment FID="1" tsid="11">
  <a/></Fragment>

<Fragment FID="1.1" tsid="12">
  <b><c> ABC</c></b>
<Fragment>

<Fragment FID="1.1.1" tsid="13">
  <c> XYZ </c>
</Fragment>
```

SMVS
Page No.
Date : / /

* XML Identifire : Region-Based Identifire

=> XML Region-Based Identifire are a way to identify and manage the structure of XML document.

In XML Region-Based Identifire use offset or positional information to identify XML document.

This method use pairs of position to denote the start and end of the each element.

-> Basic of Region-Based Identifire or Steps of Region-Based Identifire.

1 (Begin Tag, End Tag) Counting :

Instead of counting character offsets, count only Opening and closing tag.

Count offset start and ~~End~~ End.

2 (Pre, Post) Identifiers :

Count nodes during a pre order traversal and assign these as <pre> value.

Page No.
Date : / /
SMVS

Count# nodes during a post-order traversal and assign this as 'post' value.

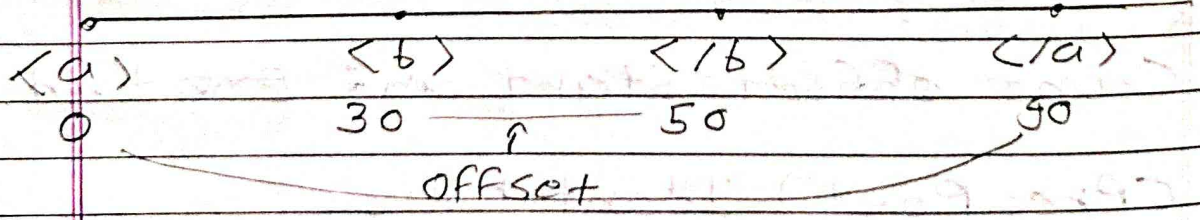
3 (pre, post, Depth) Identifiers:

Adds a 'depth' value, which representing the distance from the root.

=> If 'n1' is ancestor of 'n2' iff 'n1.pre <= n2.Pre and n2.post <= n1.post

=> Node 'n1' is parent of $n_2$ then n1.depth = n2.depth - 1

Ex. <a>
Some Text
 Text 
More
</a>




=> Here, For element

<a> offset = (0, 90)
 offset = (30, 50)

SMVS
Page No.
Date : / /

=> Here, <a> is a parent of b.
So, Starting Tag <a> and Ending Tag </a>
So, Tag count = (1, 4)

For, <b>, Tag Count = (2, 3)

=> Here, <a> is Root element and parent of <b>

(Exchange)

So, ~~a.pre <= b.pre~~
1 = 2
So, a.post = 2 and
b.post = 1

=> Here, <a> is Root element
So, distance from Root = Depth = 0

and For <b> = 1

So, For <a> = (pre, post, depth)
= (1, 2, 0)

<b> = (pre, post, depth)
= (2, 1, 1)

SMVS
Page No.
Date : / /

* XML Identifire : Dewey Classification

=> The Dewey classification schema, originally designed for libary organization and can be used for XML Document.

Dewey is provide hierachical structure or Identifire system to the XML Document.

This method creates idenifires by appending suffixes to the parent's ID.

=> Principle of Dewey IDs

1 Hierachical structure :

Each node's ID is derived by appending a unique suffix to its parent's ID.

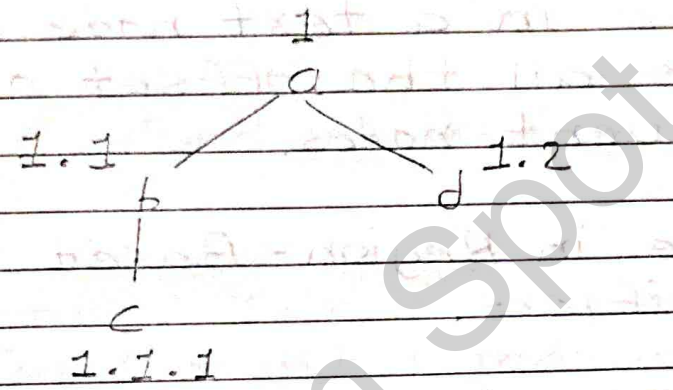
If Node n1 is parent of n2 than n1 is a prefix of n2

Ex. n1 = prefix = 1.1 than n2 = 1.1.1 (extends)

If n2 has sibling called n3 than n3 = 1.1.2 (same Parent Prefix)

SMVS
Page No.
Date : / /

Ex. <a>
<b>
<c> Text </c>
<b>
<d> More </d>
</a>




* Structural Identifires and Updates in XML Document.

=> When XML Document are updates or Change than structural identifire efficiently maintaining the XML Document's Integrity and performance.

Structural Identifires Manage the Changes of XML Document Structure

=) Impact :

1 Change in offset-Based Identifiers :

Even a Single character add or remove in a text node can change all the offset of all subsequent nodes.

2 Change in Region-Based Identifiers.

This Identifier are not impact by the changes to text node.

Inserting node require adjusting the identifier of all subsequent node in preorder.

Deleting node require to introduce gap, but not impact on structural join.

3 Dewey IDs :

Character change / remove can not impacted to change text nodes.

SMVS
Page No.
Date : / /

Inserting a new node require the change all subsequent nodes Dewey ID's

Deletion can Gives a gap, but not required Dewey ID changes.

* XML Evaluation Techniques : Structural Join.

=> Structural Join in XML databases are Fundamental operation that combine tuples from two input based on a structural relationship.

=> Let $P_1$ and $P_2$ be partial evaluation plans in XML Database, where

- $p_1.x$ is an attribute of output of $P_1$
- $p_2.y$ is an attribute of output $P_2$

Both, $P_1.x$ and $P_2.y$ has structural IDs.

-> Two Key binary relationship are used.

- is parent of (<) : One node is parent of another.

- isAncestorOf(<<): One node is an Ancestor of another.

Structural Join.

- IF $P_1.X$ is ancestor of $P_2.Y$

$$P_1 \bowtie X<<Y\ P_2 = \{(t_1, t_2) \mid t_1 \in P_1, t_2 \in P_2, t_1.X << t_2.Y\}$$

- IF $P_1.X$ is parent of $P_2.Y$

$$P_1 \bowtie X<Y\ P_2 = \{(t_1, t_2) \mid t_1 \in P_1, t_2 \in P_2, t_1.X < t_2.Y\}$$

=> Efficiency Consideration:

Goal is to reduce CPU, memory and I/O cost while we perform structural join on large XML Document.

1 Nested Loop Join: Iterates over the output of $P_1$ for each tuple, iterater over the output of $P_2$

Cost = $O(|P_1| \times |P_2|)$

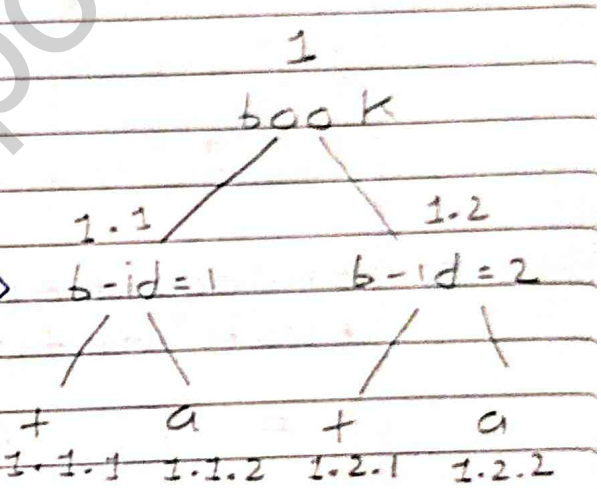
2. Hash Join: Similar to Hash Join in Relational Database

Cost = O(|P1| + |P2|)

3. Stack-Based Join: Used to evaluate scheme that have (start, end) ID.

Cost = n(|P1| + |P2|)

Ex.
```
<book>
    <book id="1">
        <t> XYZ </t>
        <a> A </a>
    </book>
    <book id="2">
        <t> ABC </t>
        <a> X </a>
    </book>
</book>
```




We want to use Stack to Join <book> and <t> element.

Stack 1 (P1) = Hold <book> element
Stack 2 (P2) = Hold <t> element
Stack 3 (P3) = Output

-> Steps :

1 Initial state: Three stack are empty.

2 Encounter <book id = "1">

Push onto stack 1 : P1 (b-id=1)

| P1 | P2 | O/P |
|---|---|---|
| 1.1 | | |

3 Encounter <t> of b-id = 1
Push onto stack 2 : P2 (1.1.1)

| P1 | P2 | O/P |
|---|---|---|
| 1.1 | 1.1.1 | |

Condition Check :
P1.top() < P2.top()
1.1 is parent of 1.1.1
So, This is vaid Pair
So, Push into O/P stack

| P1 | P2 | O/P |
|---|---|---|
| 1.1 | 1.1.1 | (1.1),(1.1.1) |

SMVS
Page No.
Date : / /

4 Same For <book id = "2">

Push onto stack P1 (b-id = 2)
Push onto stack P2 @b-id = 2 -> <t>

Condition check:
P1.top() < P2.top()
1.2 is a parent of 1.2.1
So, Push into O/P stack.

| P1 | | P2 | | O/P |
|---|---|---|---|---|
| 1.2 | | 1.2.1 | | |
| 1.1 | | 1.1.1 | | (1.1), (1.1.1) |

↓ Vaild condition

| P1 | | P2 | | O/P |
|---|---|---|---|---|
| 1.2 | | 1.2.1 | | (1.2), (1.2.1) |
| 1.1 | | 1.1.1 | | (1.1), (1.1.1) |

* Optimizing Structural Join:

=> There are Two Algorithms are used for structural Join.

1) STD Algorithm: Combines two input based on matching ID's of node.

2) STA Algorithm: Similar to STD but may sort based on different ID's depending on join order.

For Optimizing the Structural Join We have Four Plan.

1 Plan (a): Logical Plan:

Illustrates a tree patern query and the initial logical plan evaluating it.

2 Plan (b): Implement Logical plan using STD Algorithm For First Join.

A Sort Operator is required to ensure the correct order of third Join.

3 Plan (c): Uses STA For First Plan and Required Two Sort operation.

One After First Join and another after second Join

SMVS
Page No.
Date : / /

4 Plan(d) : Use STA for both Join, does not require any sort operators.

5 Plan (e) : Shows a fully pipelined plan using TwigStack algorithm.

* Holistic Twig Joins :

=> Holistic Twing Joins represents an advanced method to process the entire query.

It is reducing both running time and space requirement.

-> Logical Defination :

For a Query Tree Pattern 'q' with 'n' nodes, where each node 'i' is labeled 'a-i' and 'a-1' is the Root.

The Logical Holistic Twing operator denoted as : $\Lambda_q(1\_p1, 1\_p2, \ldots, 1\_pn)$

Where, 1_P1, - - -, 1_Pn are Logical sub-plans the output structure ID.

The result is

$$\sigma(ap_2 << a_2 \wedge ap_3 << a_3 \wedge \ldots \wedge ap_n << a_n)(lp1 \times lp2 \times \ldots \times lpn)$$

-> Algorithm PathStack

1 Initialization:

Maintain one stack ('S_i') for each query node labeled 'q_i'

Push node ID's onto their respective stack.

2 ~~Id~~ Execution:

Identify the smallest document order ID.

Push the corresponding ID onto its stack and update pointer between stacks to reflect parent-child relationship

3 PoP ID's from stack that cannot contribute to result tuple.

SMVS
Page No.
Date : / /

3 Result Construction :

When an entry is pushed onto the stack of a leaf node, build result from entires in all stacks.

* Automata on Ranked Tree :

=> In a Rank tree, the number of a childrean node can have is Fixed based on the node's label

Eacth type of node is associated with a specific arity or a fixed number of child node.

In a binary tree, every internal node has exactly two children.

Automata designed for ranked trees rely on his fixed structure to define transitition and states.

Each node has a fixed, predefined number of children and gives a uniform and predictable structure.

=> Bottom-Up Tree Automata:

Bottom-Up Tree binary tree start the process from the leaves towards the root.

- Leaf Alphabet: A Finite set of symbols for the leaves
- Internal Alphabet(Σ): A Finite set of symbols for the internal.
- Start Set(Q): A Finite set of states
- Accepting States(F): A subset of states considered as accepting.
- Transition Function(δ): Describes how states are assigned to nodes based on their labels and the states of their children.

=> Top-Down Tree Automata:

Top-Down Tree Automata process the tree from the root towards the leaves.

SMVS
Page No.
Date : / /

* Unranked Tree:

In a Unranked tree, there is no fixed limit on the number of children a node can have.

A node can have any number of chidren.

In an XML structure, elements can have varying numbers of chid elements.

=> Automata on Unranked Tree:

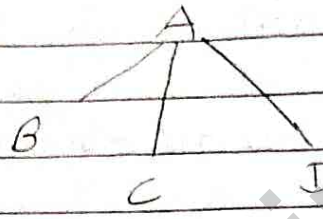
Defining automata for unranked tree is more complex, because the automata needs to handle nodes with varying numbers of children.

Nodes can have a variable number of children, allowing for a more flexible and adaptable tree structure.

Page No.
Date : / /

Ex. Ranked Tree :

Ex. UnRanked Tree :




=> Ranked Tree :