# XML Indexing II

CPS 216

Advanced Database Systems



# Announcements (April 14)

- Homework #3 will be graded by next Tuesday
- Reading assignment due next Monday
  - Selinger paper on query optimization



# XML indexing overview (review)

- It is a jungle out there
  - Different representation scheme lead to different indexes
  - Will we ever find the "One Tree" that rules them all?
- Building blocks: $B^+$-trees, inverted lists, tries, etc.
- Indexes for node/edge-based representations (graph)
- Indexes for interval-based representations (tree)
- ☞ Indexes for path-based representations (tree)
- ☞ Indexes for sequence-based representations (tree)
- ☞ Structural indexes (graph)



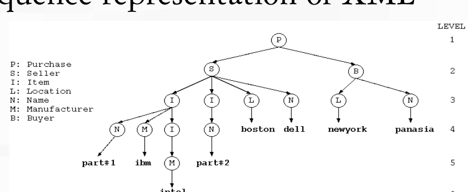
# ViST: a sequence-based index

Wang et al. "ViST: A Dynamic Index Method for Querying XML Data by Tree Structures." *SIGMOD* 2003

- Use a sequence-based encoding for XML
- Turn twig queries to subsequence matches
- Index sequences in a virtual trie using interval-based encoding



# Sequence representation of XML




- A sequence of (*symbol*, *prefix*) pairs, in depth-first order:
  - (P, $\varepsilon$), (S, P), (I, PS), (N, PSI), ($v_1$, PSIN), (M, PSI), ($v_2$, PSIM), (I, PSI), (M, PSII), ($v_3$, PSIIM), (I, PS), (N, PSI), ($v_4$, PSIN), (L, PS), ($v_5$, PSL), (N, PS), ($v_6$, PSN), (B, P), (L, PB), ($v_7$, PBL), (N, PB), ($v_8$, PBN)
- ☞ What is the worst-case storage requirement?
- ☞ Would listing symbols in depth-first order be sufficient?

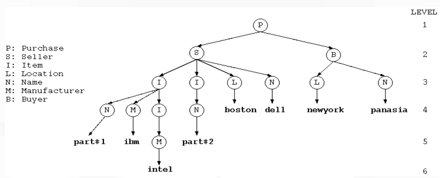


# Sequence representation of twigs

- Twigs can be represented sequences as well

| | Path Expression | Structure-Encoded Sequence |
|---|---|---|
| $Q_1:$ | $/Purchase/Seller/Item/Manufacturer$ | $(P,\epsilon)(S,P)(I,PS)(M,PSI)$ |
| $Q_2:$ | $/Purchase/[Seller[Loc=v_5]]/Buyer[Loc=v_7]$ | $(P,\epsilon)(S,P)(L,PS)(v_5,PSL)(B,P)(L,PB)(v_7,PBL)$ |
| $Q_3:$ | $/Purchase/*/[Loc=v_5]$ | $(P,\epsilon)(L,P*)(v_5,P*L)$ |
| $Q_4:$ | $/Purchase//[Manufacturer=v_3]$ | $(P,\epsilon)(M,P//)(v_3,P//M)$ |

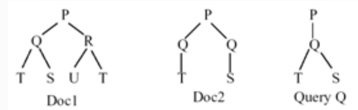
# Matching twigs as sequences




- Data: (P, $\varepsilon$), (S, P), (I, PS), (N, PSI), ($v_1$, PSIN), (M, PSI), ($v_2$, PSIM), (I, PSI), (M, PSII), ($v_3$, PSIIM), (I, PS), (N, PSI), ($v_4$, PSIN), (L, PS), ($v_5$, PSL), (N, PS), ($v_6$, PSN), (B, P), (L, PB), ($v_7$, PBL), (N, PB), ($v_8$, PBN)
- Query (Boston seller New York buyer): (P, $\varepsilon$), (S, P), (L, PS), ($v_5$, PSL), (B, P), (L, PB), ($v_7$, PBL)
- ☞ Find a (non-contiguous) subsequence of data that matches the query



# False alarms




$D_1$ = (P, e) (Q, P) (T, PQ) (S, PQ) (R, P) (U, PR) (T, PR)
$D_2$ = (P, e) (Q, P) (T, PQ) (Q, P) (S, PQ)
Q = (P, e) (Q, P) (T, PQ) (S, PQ)

- /P/Q[T]/S
  - Match sequences for /P/Q[T]/S and /P/[Q/T]/Q/S
  - Compute the difference between the answers
  - But what if a document exhibits both structures?



# Indexing sequences with a trie

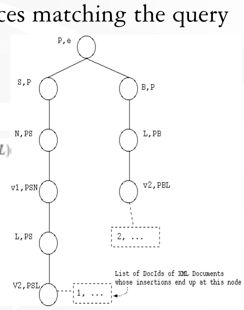
- Just insert sequences into a trie
- Search the trie for subsequences matching the query
  - Expensive because subsequences do not need to be contiguous

$Doc_1 : (P,\epsilon)(S,P)(N,PS)(v_1,PSN)(L,PS)(v_2,PSL)$

$Doc_2 : (P,\epsilon)(B,P)(L,PB)(v_2,PBL)$

$Q_1 : (P,\epsilon)(B,P)(L,PB)(v_2,PBL)$

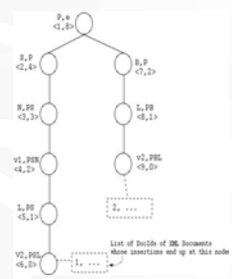
$Q_2 : (P,\epsilon)(L,P*)(v_2,P*L)$






# "Virtual trie" idea

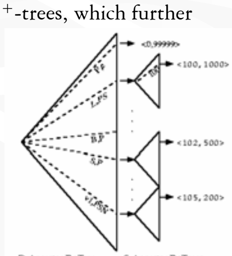
- Use (*left*, *size*) to encode trie nodes
  - *size* = *right* – *left*
  - Supports efficient "skipping"
- Index in a regular B$^+$-tree
- No need to store the trie itself





# ViST structures

- D-Ancestor B$^+$-tree indexes trie nodes by (*symbol*, *prefix*)
  - Facilitates prefix matching (checking for ancestor-descendent relationships in documents)
- Leaf nodes point to S-Ancestor B$^+$-trees, which further index nodes by (*left*, *size*)
  - Facilitates skipping in the trie (checking for ancestor-descendent relationships in the trie)
- Subsequence matching → repeated index lookups





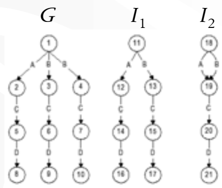

# Lore's DataGuide: a structural index

Goldman & Widom. "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases." *VLDB*, 1997

- Given an XML data graph $G$, a DataGuide is an index graph $I$ with the following properties
  - Every label path in $G$ also occurs in $I$
    - Complete coverage
  - Every label path in $I$ also occurs in $G$
    - Accurate coverage
  - Every label path in $I$ (starting from a particular object) is unique (i.e., $I$ is a DFA)
    - Efficient search: a label path of length $n$ traverses $n$ edges and ends at one node
  - Each index node in $I$ points to its extent: a set of data nodes in $G$
    - Label path query on $G$ → label path query on $I$

# Strong DataGuide

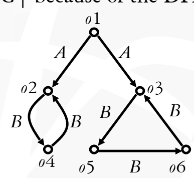
- Let $p$, $p'$ be two label path expressions and $G$ a graph; define $p \equiv_G p'$ if $p(G) = p'(G)$
  - That is, $p$ and $p'$ are indistinguishable on $G$
- $I$ is a strong DataGuide for a database $G$ if the equivalence relations $\equiv_I$ and $\equiv_G$ are the same
- Example
  - $I_1$ is strong; $I_2$ is not
  - A.C($G$) = { 5 }, B.C($G$) = { 6, 7 }
    - Not equal
  - A.C($I_2$) = { 20 }, B.C($I_2$) = { 20 }
    - Equal






# Size of DataGuides

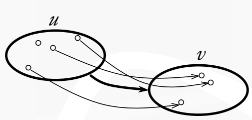
- If $G$ is a tree, then $|I| \leq |G|$
  - Linear construction time
- In the worst case, the size of a strong DataGuide may be exponential in $|G|$ because of the DFA requirement




☞ Relax the DFA requirement?



# NFA-based structural indexes




- Defined using an equivalence relation (based on the graph structure)
  - Each index node $v$ corresponds to an equivalence class of data nodes in $G$ (denoted *v.extent*)
  - There is a edge from $u$ to $v$ in $I$ iff there exists a edge from a node in *u.extent* to a node in *v.extent*

☞ $|I| \leq |G|$ by definition because extents do not overlap; however, the structure is no longer a DFA



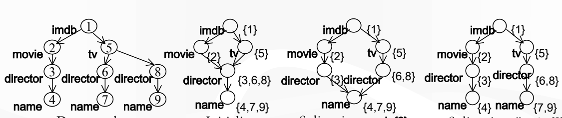
# 1-index

Milo & Suciu, "Index Structures for Path Expressions." *ICDT*, 1997

- "Perfect" equivalence relation: two data nodes are equivalent iff they are not distinguishable by label path expressions
  - That is, the sets of label path expressions that can reach them are the same
  - Too expensive to compute in practice
- 1-index uses a less perfect equivalent relation, bisimilarity, which is easier to compute
  - If two nodes are bisimilar, then they are not distinguishable by label path expressions
  - The converse is not necessary true
  - ☞ May result in larger indexes



# 1-index construction




Data graph    Initialize    Split using **movie{2}**    Split using **director{3}**

- Initialize the index
  - Data nodes with the same label go into the same index node
- Pick an index node $u$ to apply a *split* operation
  - For each index node $v$, split it into $v_1$ and $v_2$ (if both have non-empty extents)
    - $v_1$.*extent* contains data nodes in *v.extent* that are children of *u.extent*
    - $v_2$.*extent* contains the rest of *v.extent*
- Repeat *split* until there is no more change to the index