# XML Indexing

CPS 296.1

Topics in Database Systems

---

# Roadmap

- Index fabric
  - Cooper et al. “A Fast Index for Semistructured Data.” *VLDB*, 2001
- DataGuide
  - Goldman and Widom. “DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases.” *VLDB*, 1997
- T-indexes
  - Milo and Suciu. “Index Structures for Path Expressions.” *ICDT*, 1997
- Some recent papers
  - Grust; Chung et al.; Kaushik et al., *SIGMOD*, 2002
  - Kaushik et al., *ICDE*, 2002

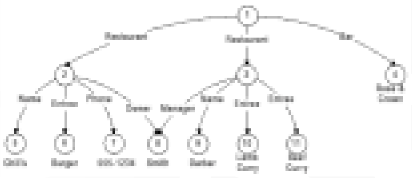
---

# DataGuides

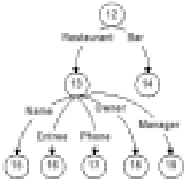
- Can handle graph data and arbitrary regular path expressions
- Given a semistructured/XML database instance *DB*, a DataGuide for *DB* is a graph *G* such that:
  - Every label path in *DB* also occurs in *G*
    - Complete coverage
  - Every label path in *G* also occurs in *DB*
    - Accurate coverage (no bogus path)
  - Every label path in *G* (starting from a particular object) is unique (i.e., *G* is a DFA)
    - Efficient search: to process a label path of length *n*, just examine *n* nodes in *G*

---

# DataGuide example



Database

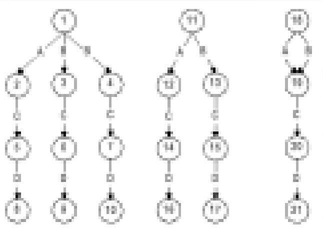


DataGuide

Each node in the DataGuide can point to a set of database nodes

---

# Multiple DataGuides for same data



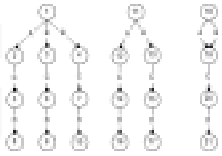
Database DataGuide $G_1$ DataGuide $G_2$

Which is better?

---

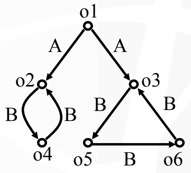
# Strong DataGuides

- 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*
- *G* is a strong DataGuide for a database *DB* if the equivalence relations $\equiv_G$ and $\equiv_{DB}$ are the same
- Example
  - $G_1$ is strong; $G_2$ is not
  - A.C(*DB*) = { 5 }, B.C(*DB*) = { 6, 7 }
    - Not equal
  - A.C($G_2$) = { 20 }, B.C($G_2$) = { 20 }
    - Equal



# Size of DataGuides

- If *DB* is a tree, then | *G* | ≤ | *DB* |
  - Linear construction time
- In the worst case, however, the size of a strong DataGuide may be exponential in | *DB* |



---

# T-indexes

- Can handle graph data and, in general, multiple path expressions chained in sequence
  - 1-index indexes all objects reachable through an arbitrary path expression *P* from a root
  - 2-index indexes all pairs of objects connected by an arbitrary path expression *P*
  - T-index indexes all sequences of objects connected by a sequence of path expressions

---

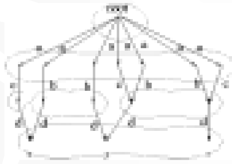
# A first attempt at 1-index (slide 1)

- Let $L_v$ be the set of words on paths from some root node to *v*
  - $L_v = \{ l_1 l_2 \ldots l_n \mid \text{root} \xrightarrow{l_1} v_1 \xrightarrow{l_2} \ldots \xrightarrow{l_n} v \}$
  - That is, all path queries that lead to *v*
- Define equivalence relation ≡ on the nodes in *DB*
  - $u \equiv v$ if $L_u = L_v$
  - That is, *u* and *v* are indistinguishable by path queries starting from the root
- Notation: [*u*] is the equivalent class containing *u*

---

# A first attempt at 1-index (slide 2)

- Index is also a graph (no bigger than *DB*)
  - Each index node corresponds to an equivalent class; it points to the set of *DB* nodes in that equivalent class
  - There is an index edge labeled *e* from *s* to *s*' if there is a *DB* edge labeled *e* from a node in *s* to a node in *s*'



- ➢ Any accurate index should have at least this many nodes
- ➢ Expensive to construct (PSPACE-complete)

---

# 1-index

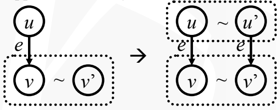
Idea: use simulation/bi-simulation instead of ≡

- Stronger conditions → finer equivalence classes → more index nodes
- Simulation and bi-simulation are much easier to compute (PTIME)
  - Details in other papers
  - To be practical, still need
    - External-memory construction algorithm
    - Incremental index update algorithm

---

# Simulation/bi-simulation (slide 1)

- A binary relation ~ on *DB* nodes is a (backward) bi-simulation if
  - If *v* ~ *v*' and *v* is a root, then so is *v*' (and vice versa)
    - Root nodes can be bi-similar only to root nodes
  - If *v* ~ *v*', then for any edge $u \xrightarrow{e} v$ there exists $u' \xrightarrow{e} v'$ such that *u* ~ *u*' (and vice versa)
    - Edges are mapped consistently



- Simulation: no "vice versa" (not symmetric in general)

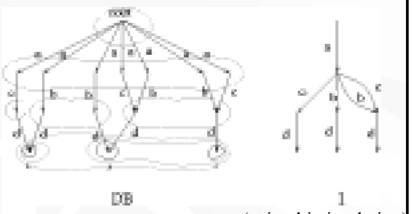
# Simulation/bi-simulation (slide 2)

- Two nodes *u* and *v* are bi-similar ($u \approx_b v$) if they are related in some bi-simulation
- Two nodes *u* and *v* are similar ($u \approx_s v$) if there are two simulations ~ and ~' s.t. $u \sim v$ and $v \sim' u$
- Fact: $u \approx_b v \Rightarrow u \approx_s v \Rightarrow u \equiv v$
  - Why?

---

# 1-index example

- $x \equiv y \equiv z$
- $x \not\approx_s y \approx_s z$
- $x \not\approx_b y \not\approx_b z$



(using bi-simulation)

---

# Analyzing 1-index

- For a tree-structured *DB*, 1-indexes using $\approx_b$, $\approx_s$, $\equiv$ are all identical to DataGuide
- Always: size(1-index) ≤ size(*DB*)
  - Unlike DataGuide
  - But we are back to NFS; is lookup time bounded?
- Always: can construct index in $O(|DB| \log|DB|)$
- Still need: external-memory construction algorithm and incremental update algorithm
- Designed to answer arbitrarily complex path expressions, but such expressions may not show up often in queries

---

# 2-index

- 1-index is for queries of the form: root $\xrightarrow{P} x$
  - Given *P*, find all *x*'s that satisfy the query
- 2-index is for queries of the form: root $\xrightarrow{*} x_1 \xrightarrow{P} x_2$
  - Given *P*, find all $(x_1, x_2)$ pairs that satisfy the query
- Again, index is a graph
  - What are the nodes?
  - What are the edges?

---

# Nodes of 2-index

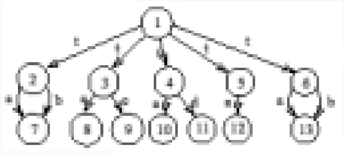
- Let $L_{(u, v)}$ be the set of words on the paths from *u* to *v*
  - $L_{(u, v)} = \{ l_1 l_2 \ldots l_n \mid u \xrightarrow{l_1} \ldots \xrightarrow{l_n} v \}$
  - That is, all path queries that return $(u, v)$ as one of its answers
- Define equivalence relation ≡ on pairs of nodes in *DB*
  - $(u, v) \equiv (u', v')$ if $L_{(u, v)} = L_{(u', v')}$
  - That is, they are indistinguishable by path queries of the form: root $\xrightarrow{*} x_1 \xrightarrow{P} x_2$
- Nodes in a 2-index correspond to equivalent classes defined by ≡; each 2-index node points to $[(u, v)]$, a set of pairs in the same equivalent class as $(u, v)$
  - Again, we can use a refinement of ≡ that is easier to compute

---

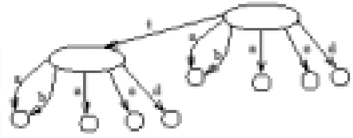
# Edges of 2-index

- Define 2-index edges in a way such that:
  A path query *P* on the 2-index returns a set of 2-index nodes that point to the answer to the query root $\xrightarrow{*} x_1 \xrightarrow{P} x_2$ in *DB*
- If $u \xrightarrow{e} u'$ in *DB*, then for each node *v* in *DB*, $[(v, u)] \xrightarrow{e} [(v, u')]$ in the 2-index
  - Intuitively, if *v* and *u* are connected via *P*, then *v* and *u*' are connected via *P.e*
- A root of a 2-index has the form $[(u, u)]$ because $L_{(u, u)}$ contains the empty word

# 2-index example



- In general, size of the 2-index may be quadratic in | *DB* |



---

# T-index

- T-index handles template: root $\xrightarrow{T_1} x_1 \xrightarrow{T_2} \dots \xrightarrow{T_n} x_n$
  - Each $T_i$ can be
    - A constant path expression, or
    - An arbitrary path expression
    - ➢Example template: Restaurant $x_1$, $x_1.P$ $x_2$
    - ➢The paper also handles an arbitrary formula (single-step path), but we will not consider it here for simplicity
  - Given $T_1, \dots, T_n$, find $(x_1, \dots, x_n)$ tuples that satisfy the query
    - Queries matching the example template:
      Restaurant $x_1$, $x_1$.owner $x_2$
      Restaurant $x_1$, $x_1$.manager.lastname $x_2$

---

# Nodes of T-index

- Query template: root $\xrightarrow{T_1} x_1 \xrightarrow{T_2} \dots \xrightarrow{T_n} x_n$
- Let $T_{(v_1, \dots, v_i)}$ be the language generated by regular expression $R_1$ \$ $R_2$ \$ … \$ $R_i$, where \$ is a special symbol, and
  - If $T_j$ represents an arbitrary path expression, then $R_j = L_{(v_{j-1}, v_j)}$
  - If $T_j$ represents a constant path expression, and if there is such a path from $v_{j-1}$ to $v_j$, then $R_j = S_j$ (a special symbol); otherwise $R_j = \emptyset$
- $(v_1, \dots, v_i) \equiv (u_1, \dots, u_i)$ if $T_{(v_1, \dots, v_i)} = T_{(u_1, \dots, u_i)}$
- Nodes of the T-index include
  - Equivalence classes of the form $[(v_1, \dots, v_i)]$, where $i \leq n$
  - For each $[(v_1, \dots, v_i)]$ a new node $[(v_1, \dots, v_i)]^\$$

---

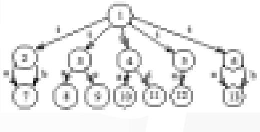
# Edges of T-index

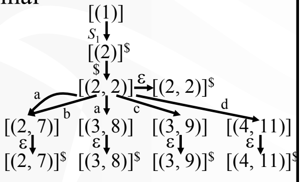
- For each $[(v_1, \dots, v_{i-1}, v_i)]^\$$, there is an edge in T-index $[(v_1, \dots, v_{i-1}, v_i)]^\$ \xrightarrow{\$} [(v_1, \dots, v_{i-1}, v_i, v_i)]$
  - Intuition: after binding $x_i$ to $v_i$, start matching $T_{i+1}$ from $v_i$
- If $T_i$ represents an arbitrary path expression
  - If $v_i \xrightarrow{e} v_i$' in *DB*, then $[(v_1, \dots, v_{i-1}, v_i)] \xrightarrow{e} [(v_1, \dots, v_{i-1}, v_i')]$
    - Intuition: $e$ can be part of $T_i$
  - $[(v_1, \dots, v_{i-1}, v_i)] \xrightarrow{\varepsilon} [(v_1, \dots, v_{i-1}, v_i)]^\$$
    - Intuition: $T_i$ can be of any length and terminated right here
- If $T_i$ represents a constant path expression
  - If $v_i \xrightarrow{T_i} v_i$' in *DB*, then $[(v_1, \dots, v_{i-1}, v_i)] \xrightarrow{S_i} [(v_1, \dots, v_{i-1}, v_i')]^\$$
    - Intuition: special symbol $S_i$ represents a complete match of $T_i$

---

# Roots, terminals, and an example

- Roots have the form $[(v)]$, where $v$ is a root of *DB*
- Terminals have the form $[(v_1, \dots, v_{n-1}, v_n)]^\$$
- Remove all nodes not reachable from root or not having any path to terminal
- Example: t $x_1$, $x_1$.* $x_2$





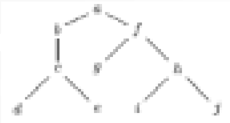
---

# Indexing XPath axes

- Most indexing work so far concentrates on speeding up parent-child traversals
- What about other types of XPath axes such as following, preceding, etc.?
  - Example: "preceding" axis contains all nodes that are before the context node in document order, excluding any ancestors
    //event[name="end"]/preceding::event[name="begin"]

➢ Grust. "Accelerating XPath Location Steps." *SIGMOD*, 2002

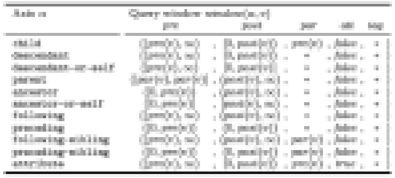
# Pre- and post-order traversal



- Pre-order traversal (self; left subtree; right subtree)
  - *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*
  - Pre-order ranks of nodes: pre(*a*) = 0, pre(*b*) = 1, pre(*c*) = 2, …
- Post-order traversal (left subtree; right subtree; self)
  - *d*, *e*, *c*, *b*, *g*, *i*, *j*, *h*, *f*, *a*
  - Post-order ranks of nodes: post(*d*) = 0, post(*e*) = 1, …
- Idea: use these ranks to determine node relationship

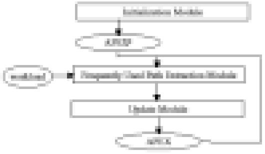
# Node descriptor indexing

- Descriptor of a node *v*: desc(*v*) = ⟨ pre(*v*), post(*v*), par(*v*), att(*v*), tag(*v*) ⟩
  - par(*v*): the pre-order rank of *v*'s parent
  - att(v): true if node is attribute; false otherwise
  - tag(v): element tag or attribute name of v
- Use R-tree or B-tree on node descriptor table



# Adaptive path indexing

- Most indexing work indexes all possible paths in the data, but few paths actually come up in queries
- Index only the frequently used paths (mined from a query workload)



➢ Chung et al. "APEX: An Adaptive Path Index for XML Data." *SIGMOD*, 2002

# More XML indexing work

➢ Kaushik et al. "Exploiting Local Similarity to Efficiently Index Paths in Graph-Structured Data." *ICDE*, 2002
  - Instead of (bi-)similarity, consider (bi-)similarity w.r.t. paths of up to length *k* (may get false positives)
  - Consider index updates

➢ Kaushik et al. "Covering Indexes for Branching Path Queries." *SIGMOD*, 2002
  - Consider branching path queries such as //part[bolt AND nut]
  - Index each edge both forward and backward
  - Reduce the size of the index by ignoring unimportant tags, limiting *k*, and limiting the tree depth of branching queries