Trees in Tables

How to Encode Semi-structured Data in RM?

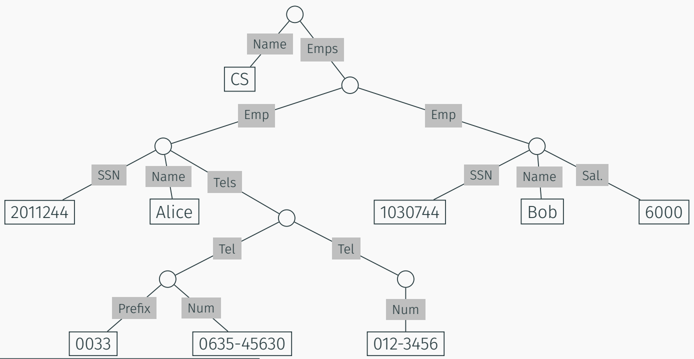
Guillaume Raschia — Nantes Université

Last update: October 14, 2025

Intro

Semi-structured Data Model

(Ordered[1]) Labeled Unranked Unbounded Tree



[1]True in XML, questionable in JSON...

Mapping Docs to Relational Databases

Requirements

- How to put semi-structured data into tables?
  preserve **tree structure, content, node id's, order**
- How to get it back efficiently?
  provide strict **round-tripping**
- How to run queries on them?
  navigation through **path expression** capabilities

Why?

Use as much of existing DB technology as possible

Large Object Blocks: a Dead End

Import serialized fragments of XML docs or JSON objects into tuple fields of type CLOB or BLOB:

uri	json	
“emp-a.json”	‘{“name”: “Alice”, “SSN”: 2011244, …}’	…

Cons

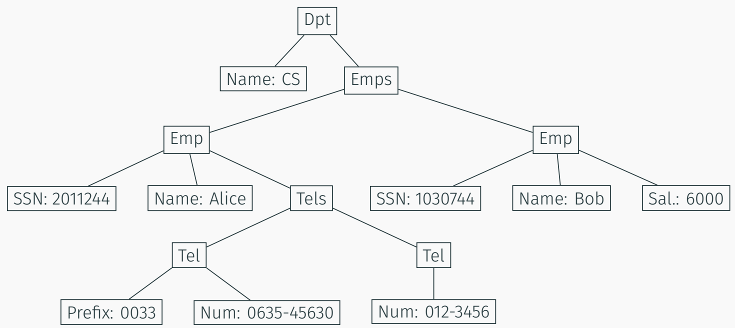
C/B-LOB column content is **monolithic and opaque** w.r.t. the relational query engine

Contents

Adjacency List

Shrink the Tree

A compact but lossless representation of XML-oriented docs



One Table to Fit Them All

node				
id	parent	label	value	order
1	NULL	dpt/NULL	NULL	1
2	1	name	CS	1
3	1	emps	NULL	2
4	3	emp/1	NULL	1
5	3	emp/2	NULL	2
6	4	ssn	2011244	1
7	4	name	Alice	2
8	4	tels	NULL	3
...	...	...	...	...

- `id`: node identity (1 record per node or per edge)
- `(id, parent)`: structural part
- `label` and `value`: content of intern and leaf nodes (/x stands for JSON alt.)
- `order`: keep track of sibling's order (optionnal)

Path Expressions

Querying the **node** table to retrieve:

- *root* node: `parent` is NULL
- *leaf* nodes: `value` is not NULL
- *children* of node x: `parent` = x
- *parent* of node x:

$$\pi_{n_1.*}\left(\sigma_{n_2.\mathtt{id}=x}\left(\text{node } n_1 \underset{n_1.\mathtt{id}=n_2.\mathtt{parent}}{\bowtie} \text{node } n_2\right)\right)$$

- *left/right siblings*: join predicate becomes
  n_1.`parent` = n_2.`parent` and n_1.`order` <> n_2.`order`

✎ *ancestors* ? *descendants* ? (to take away)

Reachability and Transitive Closure

Grand-parent of x:

$$\pi_{n_1.*}\left(\sigma_{n_3.\mathtt{id}=x}\left(\text{node } n_1 \underset{n_1.\mathtt{id}=n_2.\mathtt{parent}}{\bowtie} \text{node } n_2 \underset{n_2.\mathtt{id}=n_3.\mathtt{parent}}{\bowtie} \text{node } n_3\right)\right)$$

How to decide whether two nodes are connected or not?
How to compute the whole transitive closure of the tree?

$$\text{node} \bowtie \text{node} \bowtie \text{node} \bowtie \text{node} \bowtie \ldots$$

```
SELECT * FROM node n1
  LEFT JOIN node n2 ON n2.parent = n1.id
  LEFT JOIN node n3 ON n3.parent = n2.id
  LEFT JOIN node n4 ON n4.parent = n3.id
  LEFT JOIN node n5 ON n5.parent = n4.id
  ...
```


Recursive Queries

Limitation of the Relational Algebra

- cannot perform reachability queries
- cannot achieve the transitive closure of a graph

Both issues require **recursivity**

SQL can do it!

- (Recursive) Common Table Expression
- In the SQL-99 spec
- supported in IBM DB2, Oracle 11gr2+ (2009), PostgreSQL 8.4+, MariaDB 10.2+, MySQL 8.0.1+, SQLite 3.8.3+, MS SQL Server 2008 R2, Informix 11.50+, Firebird 2.1+, SAP Sybase (?) ...

CTE by Example

Retrieve all the ancestors of node 7 (name=Alice)

```
WITH RECURSIVE closure(nid, anc, length) AS
  -- stop condition: all pairs (id, id) are connected
  (SELECT id, id, 0 as length FROM node)
    UNION ALL
  -- recursive step:
  -- (x,y) in closure and (y,z) in node -> (x,z) in closure
  (SELECT c.nid, n.par, c.length + 1 FROM closure c
     JOIN node n ON c.anc = n.id)
  -- the actual query below
  SELECT anc FROM closure WHERE nid = 7 ;
```

- temporary `closure` table that recursively connects node 7 with all its ancestors: fix point semantics
- regular SFW query against the `closure` table

Adjacency List + CTE: a Fully-Featured Tree Encoding

- easy to grasp: one single binary relation `(id,parent)`
- can handle *ancestor* and *descendant* queries
- must enforce semantics with constraints and triggers (otherwise, diy in the app!):
  - prevent self-loops (x, x) and cycles (x, y) and (y, x)
  - prevent multiple connexions: (x, y) and (x, y)
  - ensure a connected graph: #edges = #nodes - 1
  - ensure one root only
  - add-move-remove a **tree node** is not tied to insert-update-delete a **node tuple**: must define Tx and triggers ✎

Closure Table

Materialize the Transitive Closure

Database realizes a trade-off between storage and computation costs

node

id	label	value	order
1	dpt	NULL	1
2	name	CS	1
3	emps	NULL	2
4	emp	NULL	1
5	emp	NULL	2
6	ssn	2011244	1
7	name	Alice	2
8	tels	NULL	3
...	...	...	...

closure

node	descendant	depth
1	1	0
1	2	1
1	3	1
1	4	2
1	5	2
...	...	...
2	2	0
3	3	0
3	4	1
...	...	...

Closure Table

- `node` table has no `parent` column: structure is in the `closure` table
- *ancestors* and *descendants* turn to be basic selections on the `closure` table
- Size is $\mathcal{O}(n^2)$ but actually much lower
- Overhead cost to maintain (add-move-remove)

Path Enumeration

Path Enumeration Table

Materialize paths from the root to each node

node

id	path_id	label	value	order
1	1	dpt	NULL	1
2	2	name	CS	1
3	2	emps	NULL	2
4	3	emp	NULL	1
5	3	emp	NULL	2
6	4	ssn	2011244	1
7	4	name	Alice	2
8	4	tels	NULL	3
...	...	...	...	...

path

id	key
1	/
2	/1
3	/1/3
4	/1/3/4
...	...

- separate paths from nodes to prevent from duplicate paths
- sep. char "/" in the `path.key` column
- lots of string processing in queries: substring matching

Querying the Path Enumeration Table

- *depth*:

```
SELECT LEN(p.key) - LEN(REPLACE(p.key, '/', ''))
FROM path p JOIN node n ON p.id = n.path_id
WHERE n.id = :x
```

- *descendants*:

```
SELECT * FROM node n JOIN path p ON n.path_id = p.id
WHERE p.key LIKE '%/' || :x || '%' ;
```

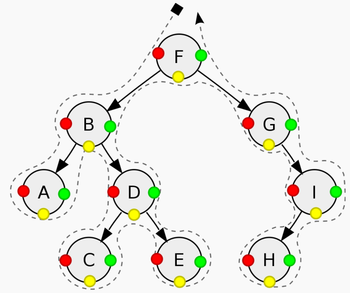
- *ancestors*:

```
SELECT n2.* FROM node n1 JOIN path p1 ON n1.path_id = p1.id
CROSS JOIN node n2 JOIN path p2 ON n2.path_id = p2.id
WHERE n1.id = :x AND LOCATE(p2.key, p1.key) = 1 ;
```

✎ *children*? add-move-remove?

Nested Sets

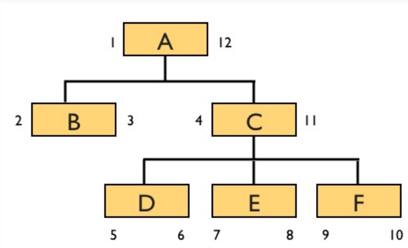
Depth-First Traversal

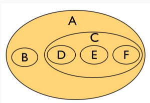


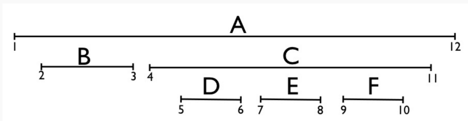
- **pre**-order (red): F, B, A, D, C, E, G, I, H;
- **in**-order (yellow): A, B, C, D, E, F, G, H, I;
- **post**-order (green): A, C, E, D, B, H, I, G, F.

Source: Tree Traversal entry from Wikipedia

Structural Node Identifiers



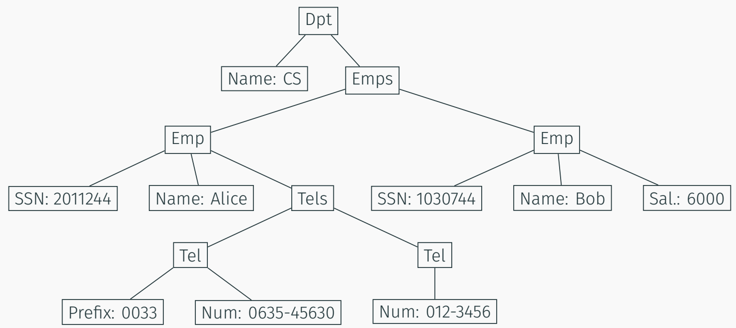




Source: L. Alberton. Trees in Databases - Advanced Data Structures (2009)

Annotate the Tree Nodes

✎ One single counter: mark **first** (pre-order) and **last** (post-order) visits only

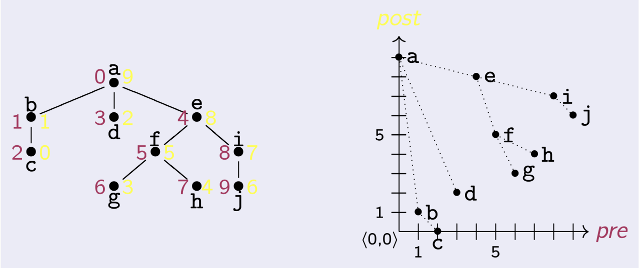


Pre-Post—aka. Left-Right—Encoding

node

id	left	right	label	value	order
1	1	32	dpt	NULL	1
2	2	3	name	CS	1
3	4	31	emps	NULL	2
4	5	20	emp	NULL	1
5	21	22	emp	NULL	2
6	6	7	ssn	2011244	1
7	8	9	name	Alice	2
8	10	21	tels	NULL	3
…	…	…	…	…	…

Pre-Post Plan



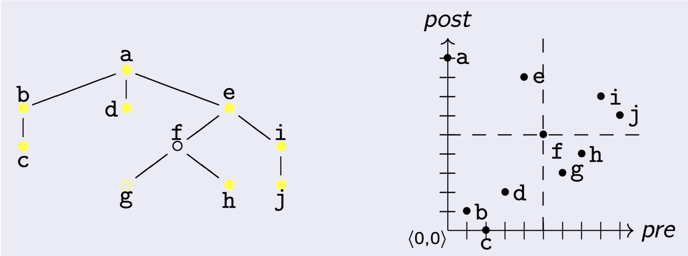
Warning

Two-counters alternative breaks the nested set property.

Only a matter of "compacting" the tree representation.

Source: M.Scholl. DBIS - Univ. of Konstanz

Pre-Post Quadrants



Source: M.Scholl. DBIS - Univ. of Konstanz

Querying the Nested Set Model

pre-post is left-right

- *root*: `left` = 1
- *leaves*: `left` = `right` -1
- *ancestors*: `left` < n.`left` and `right` > n.`right`
- *descendants*: `left` > n.`left` and `right` < n.`right`
- *parent*: ancestors and `depth` = n.`depth` - 1
- *children*: descendants and `depth` = n.`depth` + 1

✎ How to deal with *parent* and *children* without the `depth` column?

Add-Move-Remove Nodes of the Tree

Drawback

- Update all the following numbering!
- Propagate to:
  - subtree
  - all right nodes (including siblings) and their subtrees
  - ancestors up to the root node

Patch #1

Avoid renumbering on every insertion:

- long ranges: $[\![1, 2]\!]$ becomes $[\![10, 20]\!]$
- big gaps: $[\![10, 20]\!]$ and next $[\![30, 40]\!]$

Overcome the “Insert” Limitation

- Nested intervals with **rational numbers**
- Split the interval into three parts to define an inner interval

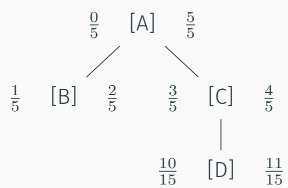
Source: E. Hildebrandt. Trees and Hierarchies in SQL (2011)

Adding a node is **always possible** (w/o reorganizing the all numbering)!

Nested Intervals Encoding

Add D node as a child of C:

split $[\frac{3}{5}, \frac{4}{5}]$ in three parts, such that the middle interval is the D range

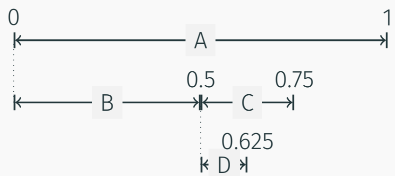


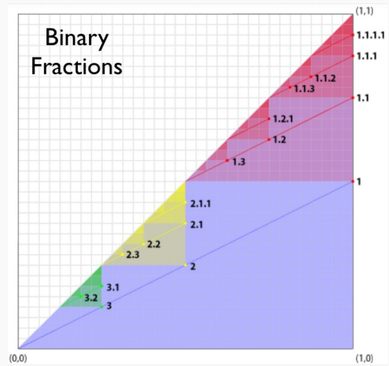
id	left_n	left_d	right_n	right_d
A	0	5	5	5
B	1	5	2	5
c	3	5	4	5
D	10	15	11	15

A Rational Schema

Recursively split ranges of node coordinates (y, x) in 2^{-k}

Binary arithmetic. Order doesn't matter.





To Sum Up

encoding	size	?child	?subtree	upd	ref. integrity
Adj. list	+	+	–	+	yes
Path enum	–	–	+	+	no
Nested sets	+	–	++	–	no
Closure tab	––	+	+	–	yes

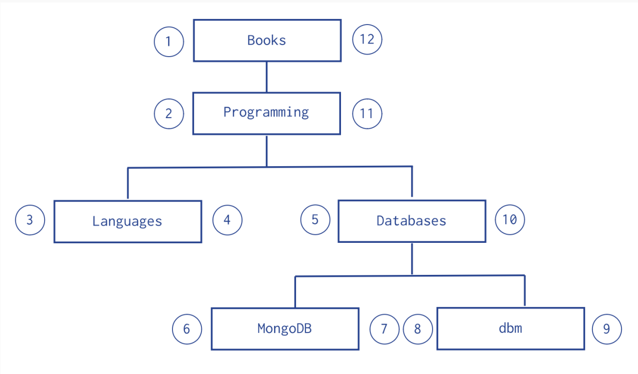
Those encodings apply to any hierarchy: org. chart, file system, phylogenetic tree, family tree, etc.

Trees in Document Stores?

Looks like a—kind of—native feature

- XML Stores actually manage trees, but
- J/BSON Document Stores fail to do so since:
  - Small docs only, then docs are hierarchy nodes rather than the entire tree
  - Require references in between nodes (docs)
  - Design tricks for tree modeling!

MongoDB Example



Source: official MongoDB documentation

Tree Encoding

Adjacency lists vs. nested sets

```
db.categories.insertMany( [
   { _id: "MongoDB", parent: "Databases" },
   { _id: "dbm", parent: "Databases" },
   { _id: "Databases", parent: "Programming" },
   { _id: "Languages", parent: "Programming" },
   { _id: "Programming", parent: "Books" },
   { _id: "Books", parent: null }
] )
```

```
db.categories.insertMany( [
   { _id: "Books", parent: 0, left: 1, right: 12 },
   { _id: "Programming", parent: "Books", left: 2, right: 11 },
   { _id: "Languages", parent: "Programming", left: 3, right: 4 },
   { _id: "Databases", parent: "Programming", left: 5, right: 10 },
   { _id: "MongoDB", parent: "Databases", left: 6, right: 7 },
   { _id: "dbm", parent: "Databases", left: 8, right: 9 }
] )
```

Inlining

Schema-based Encoding

- Inlining technique for DTD's
- Main idea: gather as many data fragments as possible in the same table
- Three modes: Basic, Shared, Hybrid
- No(t yet an) equivalent approach for JSON

✎ See J. Shanmugasundaram et al. *Relational Databases for Querying XML Documents: Limitations and Opportunities.* VLDB (1999)