Tree structures in PostgreSQL

About me

Kostas Maistrelis (Κώστας Μαϊστρέλης)

Developer @ altsol · reasonablegraph.org

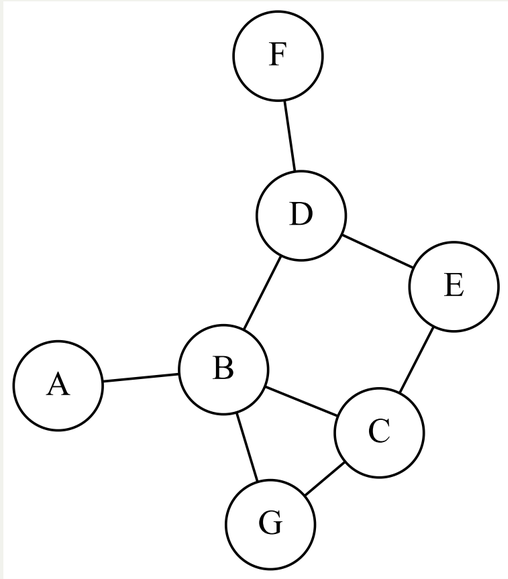
~30 years with PostgreSQL

Started with **Informix** — never looked back

Greece PostgreSQL Users Group Meetup #2

Graph

- **Abstract representation of relationships** between elements.
- The elements = **nodes** (or vertices).
- The relationships = **edges**.



More formally, a graph is a **pair of sets**:

```
G = (V, E)
```

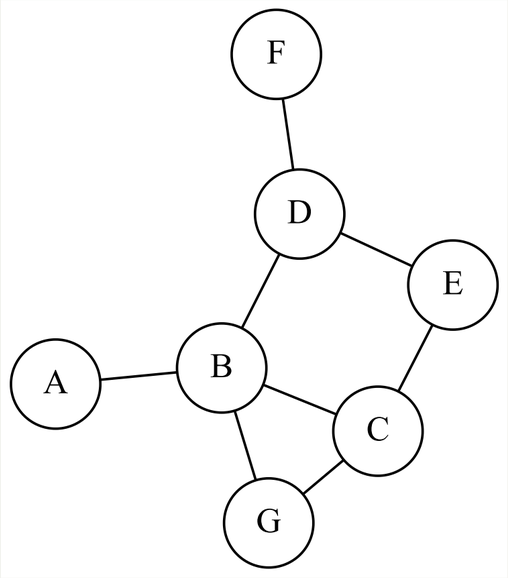
where:

```
V = set of vertices
E = set of edges
```

> That is: **objects and the relationships** between them.

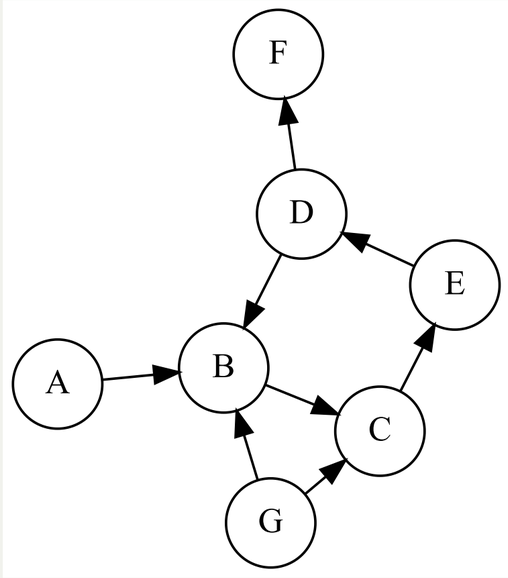
Two basic types of graphs

Undirected



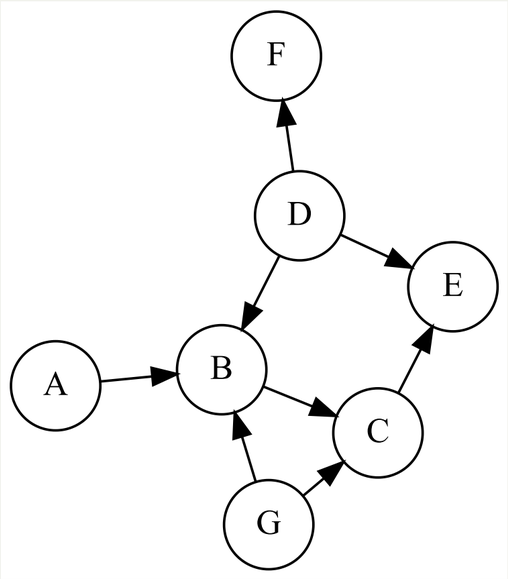
- The edges **have no direction**.
- The edge $\{u, v\}$ means: u is connected to v.
- It holds **both ways**.

Directed



- The edges **have direction**.
- The edge (u, v) = connection from u to v.
- **Direction matters**.

DAG — Directed Acyclic Graph



Directed graph WITHOUT cycles.

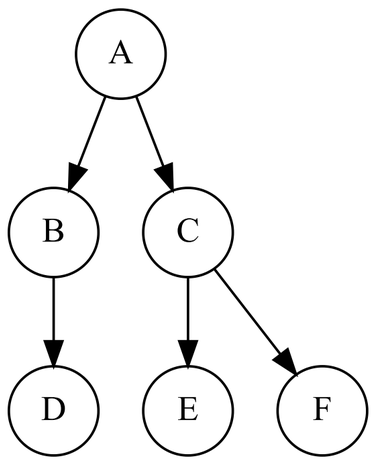
- There is no path that, following the direction of the edges, **returns to the starting node**.
- A node can have **multiple parents**.

Example:

E has **two parents** (C and D).

> **Every tree is a DAG,**
> but **not every DAG is a tree.**

Tree



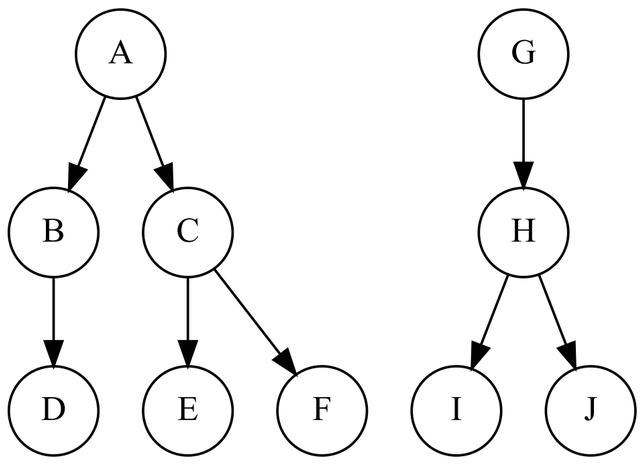
Connected graph without cycles.

- If it has `N` vertices → exactly `N - 1` edges.
- Between any two nodes there is **one and only one** path.

In databases → **rooted tree**:

- We pick one node as the **root**.
- Every other node has **exactly one parent**.
- The nodes below a node = its **subtree**.

Forest



A collection of trees.

- Each **connected component** is a separate tree.
- A single tree = special case of a forest with **one root**.

In databases this means:

- **multiple roots**,
- each root has its own subtree,
- each node has at most **one parent**.

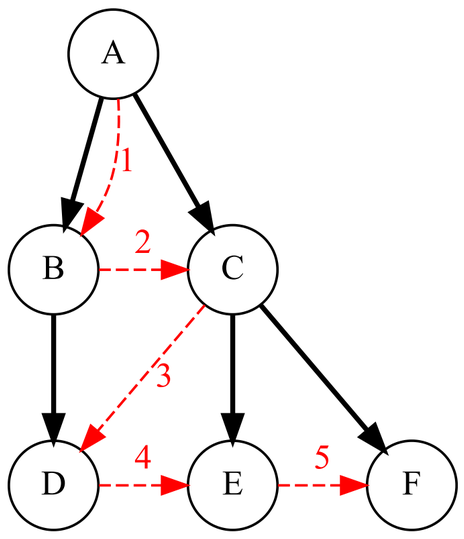
Tree Traversals

Traversal = the process by which we visit **all** nodes of a tree in a specific order.

Two basic strategies

Strategy	Idea
Depth-first (DFS)	We go **as deep as possible** down one branch before backtracking.
Breadth-first (BFS)	We visit **all nodes of the same level** first.

BFS — Breadth-first traversal



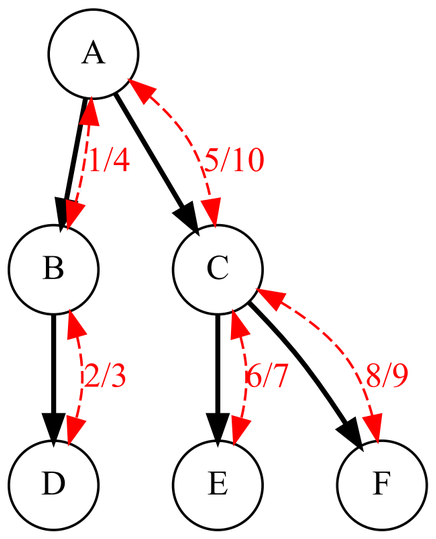
Level-by-level, from top to bottom.

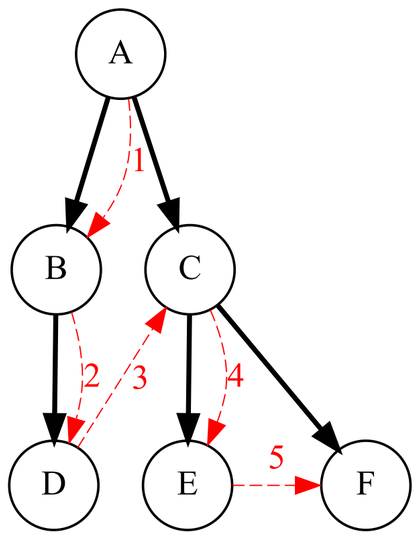
Visit order:

```
A,  B, C,  D, E, F
 ⌴   ⌴   ⌴⌴⌴
 0   1     2     ← levels
```

> First the whole level 0, then the whole level 1, then the whole level 2…

DFS — Depth-first traversal





As deep as possible down a branch before backtracking.

Visit order (pre-order):

```
A → B → D → C → E → F
```

> Go to A, then to the first child (B), then to its child (D), end of branch → backtrack, then to the next branch (C → E → F).

Bridging to PostgreSQL

> Up to here we have been talking **mathematics**.

In PostgreSQL the question becomes:

> How do we **store** these relationships so that we can **read quickly** subtrees, ancestors and paths?

That is where the **models** come in:

- **Adjacency list** + recursive CTEs
- `ltree` (materialized path)
- **Closure table**
- *(Nested Sets — only historically)*

Adjacency List

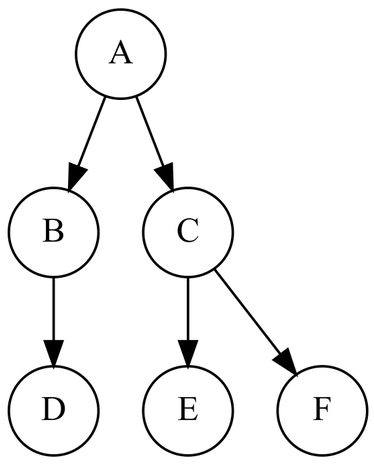
Adjacency list — the model

A table with **2 columns** for the hierarchy:

- `id` — primary key
- `parent_id` — pointer to the parent (self-FK)

```sql
CREATE TABLE node (
  id         BIGINT  PRIMARY KEY,
  parent_id  BIGINT  REFERENCES node(id),
  name       TEXT    NOT NULL
);
```

Our tree




```
 id | parent_id | name
----+-----------+------
  1 |      NULL | A
  2 |         1 | B
  3 |         1 | C
  4 |         2 | D
  5 |         3 | E
  6 |         3 | F
(6 rows)
```

Reads: Traversal By Level (before recursion)

Goal:

We want to traverse the tree level by level.

We would have to write **one SELECT per level**

```
SELECT ... WHERE id = 1                          -- root
UNION ALL
SELECT ... WHERE parent_id = 1                   -- level 1
UNION ALL
SELECT ... WHERE parent_id IN (SELECT id ...)    -- level 2
UNION ALL
...
```

The problem:

- Needs **one SELECT per level**.
- We have to know the maximum depth **in advance**.
- 3 levels → 3 SELECTs. 10 levels → 10 SELECTs.
- If tomorrow a new level is added → the query changes.

The solution: `WITH RECURSIVE`

Instead of:

- root, children, children of children, …

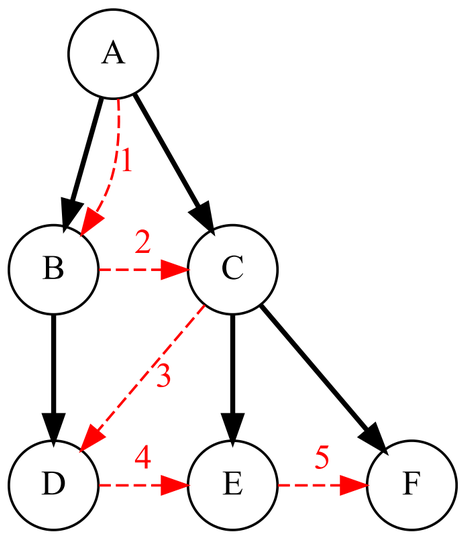
we write:

- start from the root, and **as long as you find children, keep going**

```
1  WITH RECURSIVE tree AS (
2    -- Anchor: where we start
3    SELECT id, parent_id, name, 0 AS depth
4    FROM node
5    WHERE id = 1
6
7    UNION ALL
8
9    -- Recursive: the next level
10   SELECT n.id, n.parent_id, n.name, tree.depth + 1
11   FROM node n
12   JOIN tree ON n.parent_id = tree.id
13 )
14 SELECT * FROM tree ORDER BY depth;
```

This query implements the chain of SELECTs from the previous slide:
— the anchor (lines 2-5) = the first SELECT (WHERE id=1)
— the recursive term (lines 9-12) = each subsequent SELECT
The difference: instead of writing n SELECTs, PG runs the recursive term until it produces no new rows.

Result




```
 id | parent_id | name | depth
----+-----------+------+-------
  1 |      NULL | A    |     0
  2 |         1 | B    |     1
  3 |         1 | C    |     1
  4 |         2 | D    |     2
  5 |         3 | E    |     2
  6 |         3 | F    |     2
(6 rows)
```

What does “recursive” mean?

A recursive function as a mental model

> **Explanatory** — not production code.

A recursive function has:

- **BASE (base case)** → handle the current node
- **RECURSIVE STEP** → call yourself for the children

These **two** pieces are exactly the two arms of a recursive CTE (**anchor + recursive term**).

Recursive function

```
1  CREATE FUNCTION traverse_tree(node_id BIGINT, depth INT DEFAULT 1)
2  RETURNS TABLE (id BIGINT, name TEXT, level INT)
3  LANGUAGE plpgsql AS $$
4  DECLARE child RECORD;
5  BEGIN
6    -- BASE: return the CURRENT node
7    RETURN QUERY
8      SELECT n.id, n.name, depth FROM node n WHERE n.id = node_id;
9
10   -- RECURSIVE: for each child, call YOURSELF
11   FOR child IN
12     SELECT n.id FROM node n  WHERE n.parent_id = node_id ORDER BY n.name
13   LOOP
14     RETURN QUERY
15       SELECT * FROM traverse_tree(child.id, depth + 1);
16   END LOOP;
17 END;
18 $$;
19
20 SELECT * FROM traverse_tree(1);
```

> RECURSION STEP: Calls itself on line 15: `traverse_tree(child.id, depth + 1)`

SEARCH clause (PG 14+) — what it does

Picking a traversal strategy

- `SEARCH DEPTH FIRST | BREADTH FIRST` — DFS or BFS
- `BY <columns>` — order **among siblings**
- `SET <name>` — opaque sortable column for the `ORDER BY`

SEARCH clause (PG 14+)

PostgreSQL ≥14 generates the sort column **automatically**:

```
 1 WITH RECURSIVE tree AS (
 2   SELECT id, parent_id, name, 0 AS depth
 3   FROM node
 4   WHERE parent_id IS NULL
 5
 6   UNION ALL
 7
 8   SELECT n.id, n.parent_id, n.name, tree.depth + 1
 9   FROM node n
10   JOIN tree ON n.parent_id = tree.id
11 )
12 SEARCH DEPTH FIRST BY name SET ordercol
13 SELECT id, parent_id, name, depth FROM tree
14 ORDER BY ordercol;
```

→ standard WITH RECURSIVE query with two extra lines:

12: traversal type defined via `ordercol`

14: ordering via `ORDER BY ordercol`

→ Cleaner, more declarative code — instead of the manual `path` array you had to carry yourself before PG14.

Problem: infinite recursion

PostgreSQL has **no** default recursion depth limit.

If the data has a cycle (broken `parent_id`, or actually a DAG instead of a tree), the recursive CTE runs **forever** — until `statement_timeout` or Out Of Memory.

Solution: CYCLE

- **Defense** in queries over “dirty” trees — instead of hanging the session, it terminates and shows you the problem.
- **Traversing DAGs/graphs** where a cycle is a legitimate possibility.

CYCLE clause (PG 14+)

```
1  BEGIN;
2  SET LOCAL statement_timeout = '5s'; -- fallback timeout
3
4  -- B (id=2) becomes child of D (id=4) → cycle B ⇄ D
5  UPDATE node SET parent_id = 4 WHERE id = 2;
6
7  WITH RECURSIVE descendants AS (
8    SELECT id, parent_id, name, 1 AS level FROM node WHERE id = 2
9    UNION ALL
10   SELECT n.id, n.parent_id, n.name, d.level + 1
11   FROM node n JOIN descendants d ON n.parent_id = d.id
12 )
13 CYCLE id SET is_cycle USING cycle_path
14 SELECT level, id, name, is_cycle, cycle_path FROM descendants;
15
16 ROLLBACK;
```

→ standard WITH RECURSIVE query with one extra line:

13: `CYCLE` clause

- `CYCLE id` — which column identifies the node
- `SET is_cycle` — `true` on the row that **closes** the cycle
- `USING cycle_path` — array with the path (the column is populated for every row, not just cycle-closing ones)

→ Cycle-safe by construction — no manual `visited[]` array to carry.

CYCLE — result

```
 level | id | name | is_cycle |   cycle_path
-------+----+------+----------+-----------------
     1 |  2 | B    | f        | {(2)}
     2 |  4 | D    | f        | {(2),(4)}
     3 |  2 | B    | t        | {(2),(4),(2)}
(3 rows)
```

- `is_cycle` — marks rows where the path closes a cycle
- `cycle_path` — records the visited path for every row (not just cycle-closing ones).

Writes — the easy side

All of the above is about **reads**. Writes are **trivial**:

```sql
-- INSERT: new node → one row
INSERT INTO node (id, parent_id, name) VALUES (7, 3, 'G');

-- MOVE subtree: change parent_id → the entire subtree "follows"
UPDATE node SET parent_id = 2 WHERE id = 3;

-- DELETE: one row (with ON DELETE CASCADE or re-parent)
DELETE FROM node WHERE id = 6;
```

→ No extra structure to keep in sync. `parent_id` is the **single source of truth** for the hierarchy.

Summary — trade-offs

✅ Pros

- Trivial writes
- Simple schema (one column)
- Natural representation + FK integrity

⚠️ Cons

- Reads require a recursive CTE
- No ready-made level/path
- No default cycle safety

> In the next models (closure / ltree / nested sets) the trade-off is **reversed**: cheaper reads, more expensive writes.

Closure Table

Closure — the model

1st table — `node` (the nodes)

Column	Description
`id`	unique identifier of the node

`node` can have additional payload columns (`name`, JSON, timestamps, …) — the **model** only cares about `id`.

2nd table — `node_closure` (all ancestor–descendant pairs)

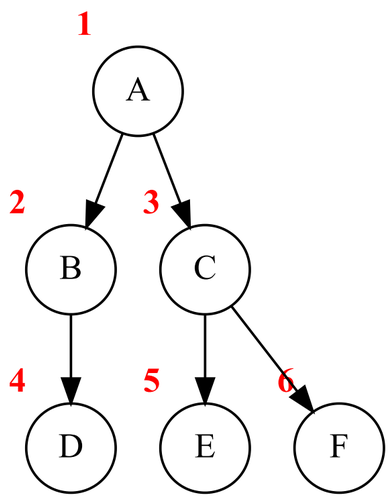
Column	Description
`ancestor_id`	ancestor node (or the node itself)
`descendant_id`	descendant node (or the node itself)
`depth`	distance in edges: 0 = self, 1 = child, 2 = grandchild…

> The table contains **all** (ancestor, descendant) pairs of the tree along with the **self-pairs** (depth = 0).
> In this model the fact that a tree is at the same time a graph is highlighted.

Closure — Tables

```
1
2  CREATE TABLE node (
3       id          BIGSERIAL PRIMARY KEY,
4       name        TEXT NOT NULL
5  );
6
7  CREATE TABLE node_closure (
8       ancestor_id    BIGINT NOT NULL REFERENCES node(id) ON DELETE CASCADE,
9       descendant_id  BIGINT NOT NULL REFERENCES node(id) ON DELETE CASCADE,
10      depth          INT    NOT NULL CHECK (depth >= 0),
11      PRIMARY KEY (ancestor_id, descendant_id)
12 );
13
```

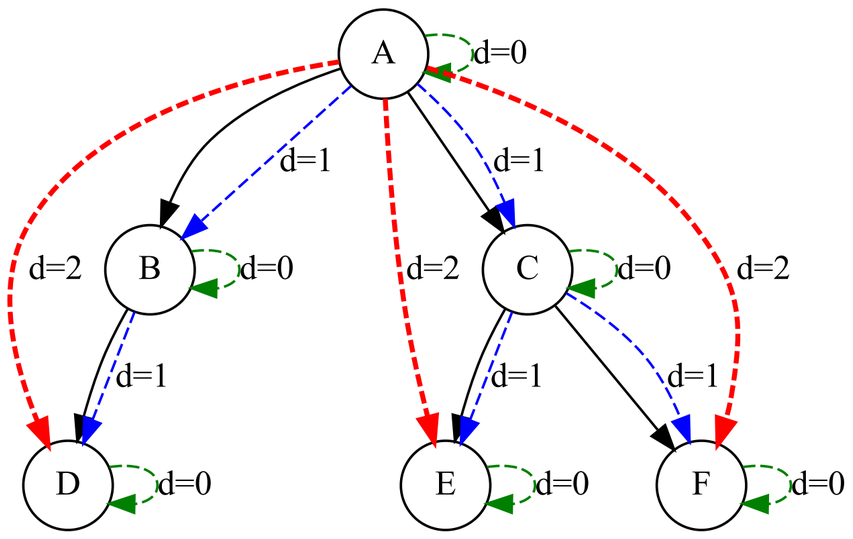
Our tree (A)




```
closure:
 id | name
----+------
  1 | A
  2 | B
  3 | C
  4 | D
  5 | E
  6 | F
(6 rows)
```

```
node_closure:
 ancestor_id | descendant_id | depth
-------------+---------------+-------
           1 |             1 |     0
           2 |             2 |     0
           1 |             2 |     1
           3 |             3 |     0
           1 |             3 |     1
           4 |             4 |     0
           2 |             4 |     1
           1 |             4 |     2
           5 |             5 |     0
           3 |             5 |     1
           1 |             5 |     2
           6 |             6 |     0
           3 |             6 |     1
           1 |             6 |     2
(14 rows)
```

Our tree (B)



The dashed arrows in the tree correspond to the rows of the `node_closure` table.

```
node_closure:
 ancestor_id | descendant_id | depth
-------------+---------------+-------
 A           | A             |     0
 B           | B             |     0
 A           | B             |     1
 C           | C             |     0
 A           | C             |     1
 D           | D             |     0
 B           | D             |     1
 A           | D             |     2
 E           | E             |     0
 C           | E             |     1
 A           | E             |     2
 F           | F             |     0
 C           | F             |     1
 A           | F             |     2
(14 rows)
```

Reads: BFS — by level

```sql
SELECT n.name, c.depth
FROM node_closure c
JOIN node n ON n.id = c.descendant_id
WHERE c.ancestor_id = 1   -- root (A)
ORDER BY c.depth, n.name;
```

> **No recursion** — simple `JOIN + ORDER BY`.
> The `depth` is **ready** in the table.

```
 name | depth
------+-------
 A    |     0
 B    |     1
 C    |     1
 D    |     2
 E    |     2
 F    |     2
(6 rows)
```

Reads: DFS pre-order

```sql
SELECT n.name, c.depth
FROM node_closure c
JOIN node n ON n.id = c.descendant_id
WHERE c.ancestor_id = 1   -- root (A)
ORDER BY (
  SELECT array_agg(a.ancestor_id ORDER BY a.depth DESC)
  FROM node_closure a
  WHERE a.descendant_id = c.descendant_id
);
```

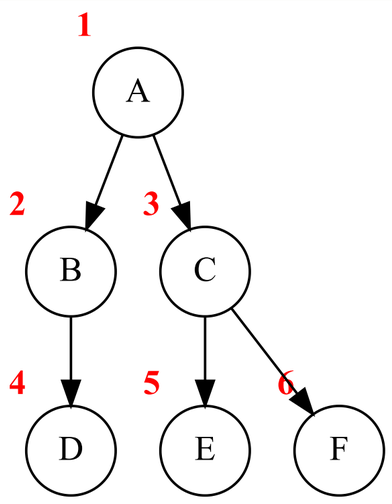
```
 name | depth
------+-------
 A    |     0
 B    |     1
 D    |     2
 C    |     1
 E    |     2
 F    |     2
(6 rows)
```

> For each node we build a **path from the root** as an array
> (all the ancestors, by `depth DESC`)

DFS — the paths up close

```sql
SELECT n.name,
       array_agg(c.ancestor_id ORDER BY c.depth DESC) AS path
FROM node_closure c
JOIN node n ON n.id = c.descendant_id
GROUP BY n.name, c.descendant_id
ORDER BY path;
```

The **same** array that goes into the `ORDER BY` of the previous slide — here we see it as a column.



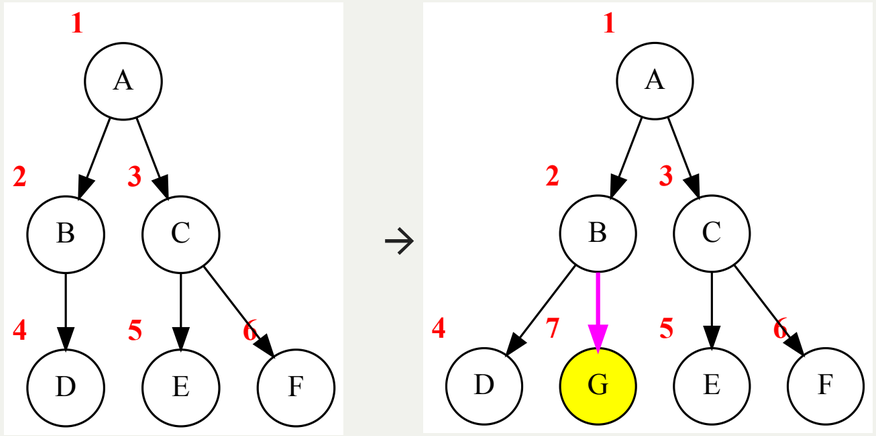

```
paths with id:

 name |  path
------+---------
 A    | {1}
 B    | {1,2}
 D    | {1,2,4}
 C    | {1,3}
 E    | {1,3,5}
 F    | {1,3,6}
(6 rows)
```

```
paths join with label:

 name |  path
------+---------
 A    | {A}
 B    | {A,B}
 D    | {A,B,D}
 C    | {A,C}
 E    | {A,C,E}
 F    | {A,C,F}
(6 rows)
```

Write — adding node G under B

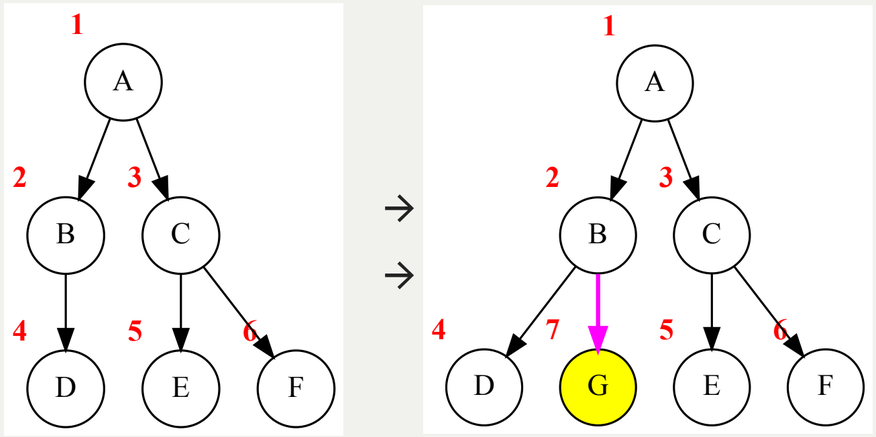



```sql
-- 1) The node itself
INSERT INTO node (id, name) VALUES (7, 'G');

-- 2) Self-row: every node is ancestor of itself (depth=0)
INSERT INTO node_closure VALUES (7, 7, 0);

-- 3) Inherit ALL the ancestors of the parent, with depth+1
INSERT INTO node_closure (ancestor_id, descendant_id, depth)
SELECT c.ancestor_id, 7, c.depth + 1
FROM node_closure c
WHERE c.descendant_id = 2;   -- the parent (B)
```

Write — result



New rows in node:

```
 id | name
----+------
  7 | G
```

New rows in node_closure:

```
 ancestor_id | descendant_id | depth
-------------+---------------+-------
           7 |             7 |     0   -- self        (G)
           2 |             7 |     1   -- parent      (B)
           1 |             7 |     2   -- grandparent (A)
```

> **1** row in node
> **3** rows in node_closure
> (= depth(parent) + 1)

Summary — trade-offs

✅ Pros

- Reads **without recursion** (BFS, ancestors, descendants → flat joins)
- The `depth` is **ready** in the table
- "Is X an ancestor of Y?" → 1 row lookup
- Friendly to the query planner (simple joins + indexed PK)
- Works for **DAGs** too, not only trees

⚠️ Cons

- Writes are **multiple rows**: 1 node → `depth(parent)+1` closure rows
- **Move subtree** expensive: re-compute all the pairs
- **Storage overhead** that grows with depth
- **Two tables** to keep in sync (triggers/functions)
- DFS pre-order requires a correlated subquery (not as natural as BFS)

> In the **next model** (`ltree` — materialized path) the hierarchy is stored as **one column** on the same table — a different reads/writes balance.

ltree

ltree — the model

One table — `node` (the whole hierarchy lives in the `path` column)

Column	Description
`id`	unique identifier of the node
`path`	**type** `LTREE` — the path root → node as labels with `.`

`node` can have additional payload columns (`name`, JSON, timestamps, …)
— the **model** only cares about `path` (+ a stable label per node).

What `LTREE` is

- Built-in type of PostgreSQL (extension `ltree`, in contrib).
- Value: dot-separated labels — e.g. `'1.2.4'`, `'shop.electronics.phones'`.
- Labels: alphanumeric + `_` (no spaces, no special characters).
- Operators: `@>` (is ancestor of), `<@` (is descendant of), `~` (lquery match), `@` (ltxtquery).
- Comes with a **GiST index** → indexed ancestor/descendant lookups.

> No extra structure — the hierarchy is **one column** on the same table.

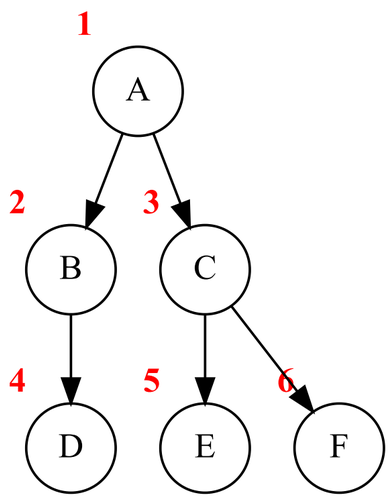
ltree — Table

```sql
CREATE EXTENSION IF NOT EXISTS ltree;

CREATE TABLE node (
     id    BIGINT PRIMARY KEY,
     name  TEXT   NOT NULL,
     path  LTREE  NOT NULL UNIQUE
);

-- GiST: critical for ancestor/descendant queries (<@, @>, ~)
CREATE INDEX idx_node_path_gist ON node USING GIST (path);
```

Our tree




```
ltree.node:
 id | name | path
----+------+--------
  1 | A    | 1
  2 | B    | 1.2
  3 | C    | 1.3
  4 | D    | 1.2.4
  5 | E    | 1.3.5
  6 | F    | 1.3.6
(6 rows)
```

```
ltree.node:
 id | name | path
----+------+--------
  1 | A    | A
  2 | B    | A.B
  3 | C    | A.C
  4 | D    | A.B.D
  5 | E    | A.C.E
  6 | F    | A.C.F
(6 rows)
```

Reads: BFS — by level

```sql
SELECT name, nlevel(path) - 1 AS level
FROM node
WHERE path <@ '1'::ltree   -- root (A)
ORDER BY nlevel(path), name;
```

```
 name | level
------+-------
 A    |     0
 B    |     1
 C    |     1
 D    |     2
 E    |     2
 F    |     2
(6 rows)
```

> **No recursion** — `<@` (“descendant of”) is served by the GiST index.
> The level comes out of `nlevel(path)` — number of labels.

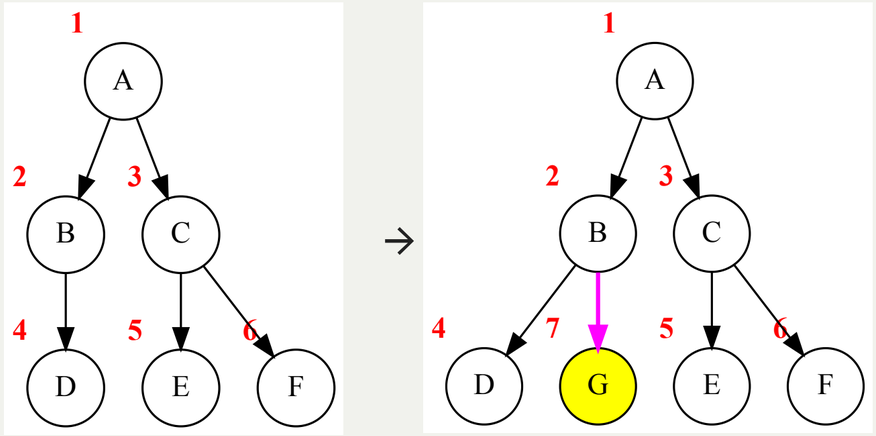
Reads: DFS pre-order

```sql
SELECT name, nlevel(path) - 1 AS level
FROM node
WHERE path <@ '1'::ltree   -- root (A)
ORDER BY path;
```

```
 name | level
------+-------
 A    |     0
 B    |     1
 D    |     2
 C    |     1
 E    |     2
 F    |     2
(6 rows)
```

> Pre-order is **free**: the path of the parent is a **prefix** of the child
> → a single `ORDER BY path` is enough.

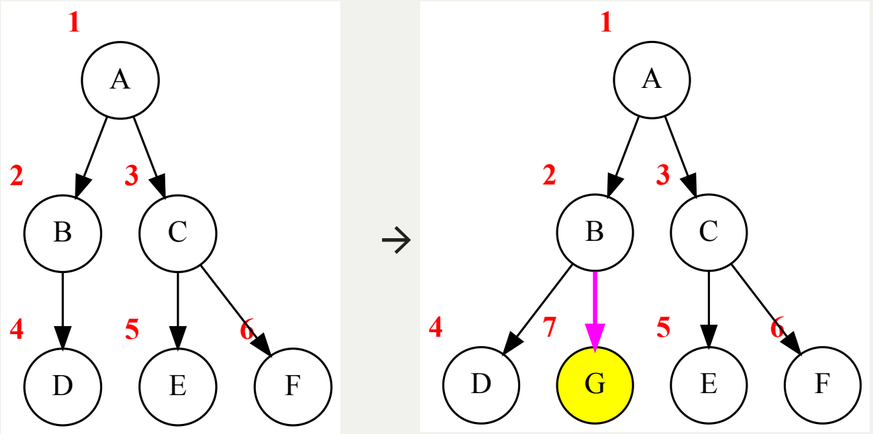
Write — adding node G under B




```sql
-- 1) Find the path of the parent (B = id 2)
SELECT path FROM node WHERE id = 2;   -- → '1.2'

-- 2) ONE row: path = parent_path || own_label
INSERT INTO node (id, name, path)
VALUES (7, 'G', '1.2' || '7'::ltree);
-- or equivalently:    text2ltree('1.2.7')
```

Write — result



New row in node:

```
 id | name | path
----+------+--------
  7 | G    | 1.2.7
```

> **1** row in node
> (= independent of depth)

Summary — trade-offs

✅ Pros

- **One column** for the whole hierarchy (not a second table)
- Pre-order = `ORDER BY path` (free)
- Ancestor / descendant with indexed `@>` / `<@` (GiST)
- `nlevel(path)` → level is ready
- Cycle prevention on move: O(1) with one `<@`
- INSERT = **one row**, independent of depth

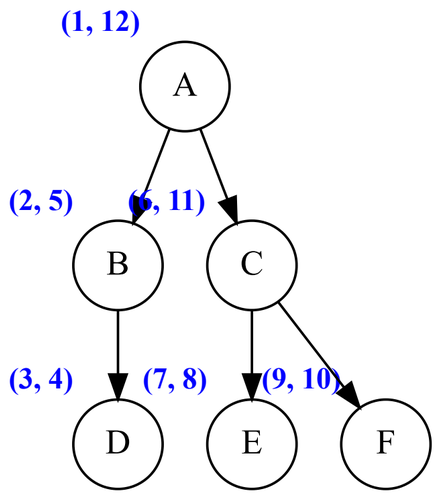
⚠️ Cons

- **PostgreSQL-specific** (extension `ltree`) — not portable
- **Move subtree** expensive: rewrites `path` on **all** descendants
- Path **denormalized** — duplication on every row
- Labels with restrictions (alphanumeric + `_`)
- Sibling order = **text comparison** on the labels (not numeric)

> Compared to **closure**: the insert/move trade-off is reversed (closure struggles on insert, ltree on move).
> Compared to **adjacency**: cheaper reads (no recursive CTE), but writes go beyond “one row” when the structure changes.

Nested Sets

Nested Sets — the idea



Each node gets **2 numbers** from a DFS traversal:

- **entering** → `lft`
- **leaving** → `rgt`

The subtree of a node = the nodes whose `lft` **is inside the interval** of the parent:

```
1 SELECT * FROM node child, node parent
2 WHERE parent.name = 'C'
3   AND child.lft BETWEEN parent.lft AND parent.rgt;
```

→ `C, E, F` with one **indexed range scan**. **Zero** recursion, **zero** joins (apart from the self-join for the range).

Nested Sets — why NOT today

⚠️ Catastrophic writes

- One insert/move → **renumbering**
- On average **half** of the `lft`/`rgt` values in the table shift
- Inserting a node = `UPDATE` of ~N/2 rows

Concurrency

- Every write touches half the table
- Practically **single-writer**

🧨 Fragility

- The `lft`/`rgt` invariant **breaks easily**
- The numbers have no meaning on their own

🚀 PG has better options

- `WITH RECURSIVE` → adjacency reads
- `ltree` → indexed subtree read **without** renumbering
- Closure → indexed joins, supports DAGs

`ltree` does **exactly** the job of Nested Sets, without any of the drawbacks.

> **Score on the read/write spectrum:**
> Adjacency = cheap writes, expensive reads ·
> **Nested Sets = the opposite extreme: maximum reads, catastrophic writes** ·
> `ltree` & closure sit in the middle.

Comparison

Summary table

Criterion	Adjacency	ltree	Closure
Subtree read	recursive CTE (N iter.)	1 indexed predicate (`<@`)	1 indexed join
Leaf insert	1 row	1 row	`depth+1` rows
Subtree move	1 UPDATE (1 row)	1 UPDATE (N paths)	N × depth edges
Cycle check on move	recursive CTE	O(1) with `<@`	O(1) PK lookup
Storage	O(N)	O(N)	O(N × depth)
DAG / multiple parents	✗	✗	✓
Integrity from the DB	✓ (1 FK)	✗ (denormalized string)	✓ (FK edges)
Extension	—	`ltree` (contrib)	—
Query ergonomics	medium	high	medium

Read benchmark — pre-order on 6,000 nodes

Model	Time	Buffers (shared hit)	What the read path does
ltree	~13 ms	**372**	Seq scan + hash joins; plain `ORDER BY path`
closure	~35 ms	868	2× aggregation over 46,812 `node_closure` rows
adjacency	~39 ms	1545	2× **Recursive Union** — scans `node` per level

- **ltree** ~3× faster, ~4× fewer buffers — it doesn't compute, it reads columns.
- **adjacency** pays the most buffers (recursion per level).
- **closure** in between: one scan instead of recursion, but over a table 7.8× larger.

> The **buffers** (8KB pages the query touched) are deterministic and predict scaling — time is noisy.

Write benchmark — move subtree of 1,793 nodes

Model	Rows written	Buffers	What the write path does
adjacency	**1**	~26	one `UPDATE` of `parent_id` — the subtree follows
ltree	1,793	~22,300	rewrites `path` of every node + GiST entries
closure	14,344	~97,700	DELETE 5,379 + INSERT 8,965 edges + ~17,930 FK trg

- **adjacency** almost free — the hierarchy is one `parent_id`.
- **ltree** pays proportionally to the **size of the subtree**.
- **closure** worst: writes `subtree × depth` edges + triggers.

The read ↔ write symmetry

Model	Read (pre-order)	Write (`move_subtree`)
adjacency	worst (1545 buffers)	**best** (1 row)
ltree	**best** (372 buffers)	medium (1,793 rows)
closure	medium (868 buffers)	**worst** (14,344 rows)

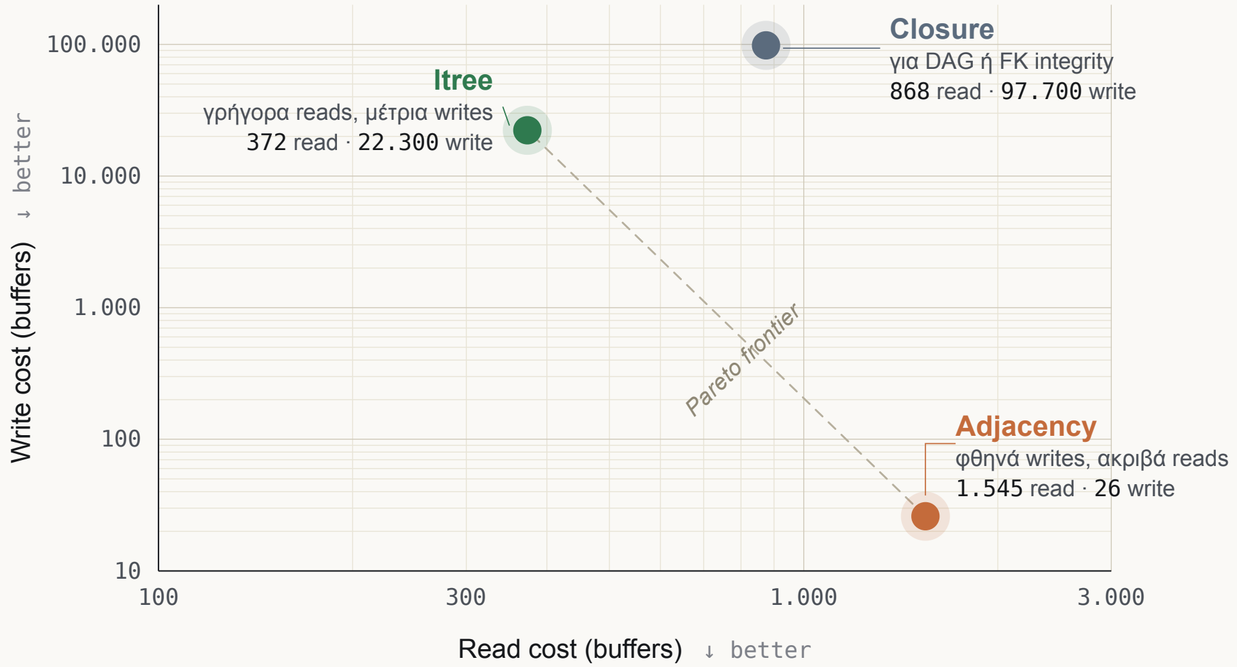
> **No model wins everywhere** — the cost simply moves around.

The **central message** of the whole presentation: you choose where to pay.

Read/Write trade-off — picture

Read/Write tradeoff — δενδρικές δομές στην PostgreSQL

Κανένα μοντέλο δεν κερδίζει παντού — το κόστος μετακινείται



Read = pre-order traversal · 6.000 κόμβοι · **Write** = move_subtree 1.793 κόμβων σε δέντρο 16.000 · 1 buffer = 8KB (EXPLAIN BUFFERS)

Match the model to the workload

You have…	Pick
Strict tree, simple keys, read-heavy	`ltree` — the good default in PG
DAG / multiple parents	**closure** — `ltree` cannot
Integrity from the DB (FKs on edges)	**closure**
Edge data (weights, dates, “primary parent”)	**closure** — `path` doesn’t fit it
Keys outside `[A-Za-z0-9_]` (UUID, Unicode)	**closure** — `ltree` labels are restricted
No extension allowed	**closure** or **adjacency**
Write-heavy, simple modeling	**adjacency** — 1 row writes
None of the above — it just runs	**adjacency** — the simplest, often enough

> The comparison is **not just** ltree-vs-closure. For many applications, the adjacency list with a recursive CTE is simply “enough” — zero extension, one FK, cheapest writes.

About this talk

View these slides online

https://maistrelis.com/postgresql/meetup-2/

Join us

Greece PostgreSQL Users Group — open to anyone.

Discord

discord.gg/xepUAKTAAu

Meetup

meetup.com/greece-postgresql-users-group