1 What Makes Data Hierarchical?

Hierarchical data represents entities arranged in levels, where some entities are above, below, or related to other entities. A hierarchy is not merely a table with a `parent_id`; it is a way of describing structure and the questions we want to ask about that structure.

Typical questions include:

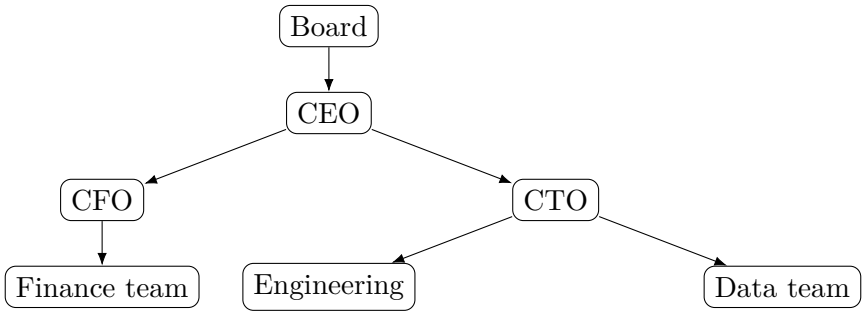
- Who reports, directly or indirectly, to a particular manager?
- Which tasks must finish before a given task can begin?
- Which categories are descendants of a product category?
- What is the path from a node to the root?
- How many levels are below a node, and what is the nearest common ancestor of two nodes?
- Is a node reachable from another node, and are there cycles?

The first design decision is conceptual: *what relationship does an edge represent?* A reporting relationship, a dependency, a containment relationship, and an *is-a* relationship may all look like parent-child links, but they have different meanings and constraints.

2 Recognizing the Right Shape

2.1 A rooted tree: one top-level node

A rooted tree has exactly one root. Every other node has exactly one parent, and there is exactly one path from the root to any node. A corporate reporting structure is a familiar example: the board (or a chairperson) sits at the top, executives report upward, and employees report to managers.



Properties of a tree with n nodes:

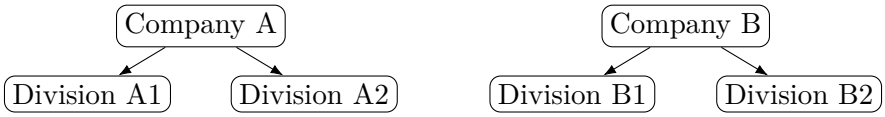
- There are exactly $n - 1$ parent-child edges.
- The root has no parent; every other node has one parent.
- There are no cycles.

- There is exactly one path between any two nodes when edge direction is ignored.

A tree is appropriate when each item belongs in exactly one place and has one authoritative parent. Examples include a file-system directory tree, a single-manager reporting chain, and a nested product taxonomy where each category has one parent.

2.2 A forest: multiple roots

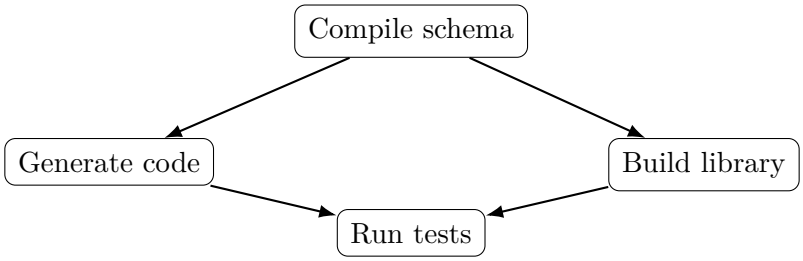
A forest is a collection of disjoint trees. It is appropriate when there are several independent top-level structures, or when the data contains several organizations, tenants, campuses, or product catalogs. A forest has no single global root, but each component has its own root.



In a relational representation, a forest usually needs no special storage mechanism: rows with a null parent are roots. A separate `tree_id` or `tenant_id` is often useful for scoping queries and enforcing that a parent belongs to the same tree.

2.3 A directed acyclic graph (DAG): multiple parents, no cycles

A directed acyclic graph permits a node to have more than one parent, but prohibits directed cycles. A scheduler or build system is a classic example. A task may depend on several prerequisite tasks, and one prerequisite may be shared by many downstream tasks.



The task *Run tests* has two prerequisites. This is not a tree, because there are two paths into the same node. It is still a DAG if all arrows point forward in the dependency order. DAGs are also used for workflow orchestration, package dependencies, data pipelines, and course prerequisites.

2.4 General graphs: cycles may be meaningful

Some relationships are not hierarchical at all. A social network, road network, set of web links, or account-transfer network can contain cycles. If A is connected to B, B to C, and C back to A, there is no meaningful top node. Such data is a general graph, not a tree.

A cycle can be a valid fact in a general graph, but it is usually an error in a reporting tree or a dependency DAG. The data model and validation rules must reflect that distinction.

2.5 Quick comparison

Structure	Parent count	Cycles?	Example / interpretation
Rooted tree	One root; each other node has one parent	No	One company reporting hierarchy
Forest	Several roots; each non-root has one parent	No	Several independent company trees
DAG	A node may have several parents	No	Scheduler tasks with prerequisites
General graph	Any number of incoming/outgoing edges	Yes, possibly	Social connections or road network

3 Two Different Meanings: Parent–Child and Is-A

These relationships are often confused because both can be drawn as a hierarchy.

3.1 Parent–child: containment, supervision, or dependency

A parent-child edge says that one particular entity is structurally above, contains, supervises, or is a prerequisite of another entity. Examples:

- A manager supervises an employee.
- A directory contains a subdirectory.
- A project contains a subtask.
- A task depends on another task.

The child is still the same kind of entity as the parent in many such models: a manager and an employee may both be rows in an `employee` table; a directory and a subdirectory may both be rows in a `directory` table.

3.2 Is-a: type specialization and inheritance

An is-a relationship describes a type hierarchy. For example, a *CheckingAccount is an Account*, and a *SavingsAccount is an Account*. The subtype inherits the identity and common attributes of the supertype, while adding subtype-specific attributes.

This is not the same as one checking account being the parent of another account. It is a statement about classification, not containment.

A common relational design uses one table per type:

```sql
CREATE TABLE account (
    account_id   bigint PRIMARY KEY,
    opened_on    date NOT NULL,
    balance      numeric(18,2) NOT NULL
);

CREATE TABLE checking_account (
    account_id   bigint PRIMARY KEY REFERENCES account(account_id),
    overdraft_limit numeric(18,2) NOT NULL
);

CREATE TABLE savings_account (
    account_id   bigint PRIMARY KEY REFERENCES account(account_id),
    interest_rate numeric(8,5) NOT NULL
);
```

The subtype's primary key is also a foreign key to the supertype. This enforces that each subtype row corresponds to an existing account. It does not, by itself, enforce that an account appears in exactly one subtype; that rule may need a trigger, a carefully designed type discriminator, or application-level validation.

Other common inheritance mappings are:

- **Single table:** one table contains all subtype columns; irrelevant columns are null. Easy to query, but can become very wide and requires checks to keep subtype-specific columns consistent.
- **Class-table inheritance:** one table for the supertype and one for each subtype. Normalized, but querying a concrete subtype requires joins.
- **Concrete-table inheritance:** each concrete subtype has its own table, repeating common columns. Simple subtype reads, but common queries and updates are more complicated and data can drift.

Keep the concepts separate: use an edge or parent key for a structural relationship; use subtype tables or a type discriminator for an is-a relationship.

4 Relational Storage Patterns

4.1 Adjacency list: store the parent on each row

The adjacency-list model is the simplest relational representation of a tree or forest. Each node has a foreign key to its parent. A null parent identifies a root.

```sql
CREATE TABLE org_node (
    node_id    bigint PRIMARY KEY,
    parent_id  bigint REFERENCES org_node(node_id),
    node_name  text NOT NULL,
```

```
    node_type  text NOT NULL
);

CREATE INDEX org_node_parent_idx ON org_node(parent_id);
```

Example rows:

node_id	parent_id	node_name	node_type
1	NULL	Board	board
2	1	CEO	executive
3	2	CFO	executive
4	2	CTO	executive
5	3	Finance	department
6	4	Engineering	department

Advantages:

- One row per entity; compact and intuitive.
- Inserting a node or moving a subtree is usually straightforward.
- Foreign keys can enforce that a non-null parent exists.
- Works naturally for trees and forests, and is a useful base representation for many applications.

Limitations:

- A direct-child lookup is easy, but finding all descendants requires traversal.
- A foreign key does not prevent a node from becoming its own ancestor.
- A single `parent_id` cannot represent a DAG node with multiple parents.

4.2 Edge table: represent relationships as rows

For DAGs and general graphs, store nodes separately from edges. Each edge is a relationship from one node to another.

```
CREATE TABLE graph_node (
    node_id bigint PRIMARY KEY,
    label   text NOT NULL
);

CREATE TABLE graph_edge (
    from_id bigint NOT NULL REFERENCES graph_node(node_id),
    to_id   bigint NOT NULL REFERENCES graph_node(node_id),
    edge_type text NOT NULL DEFAULT 'depends_on',
    PRIMARY KEY (from_id, to_id, edge_type)
);
```

```sql
CREATE INDEX graph_edge_to_idx ON graph_edge(to_id);
```

For a scheduler, define the edge direction precisely. In this example, `from_id` is a prerequisite and `to_id` is the task that depends on it. Reversing that convention is also valid, but inconsistent interpretation is a common source of bugs.

An edge table supports multiple parents and multiple children. A uniqueness constraint can prevent duplicate edges. A check constraint can prevent self-edges, but preventing longer cycles requires a traversal-based validation step or a trigger with appropriate concurrency control.

4.3 Closure table: store reachability explicitly

A closure table stores every ancestor-descendant pair, not just direct parent-child edges. It is useful when descendant and ancestor queries are frequent and the hierarchy changes less often.

```sql
CREATE TABLE org_closure (
    ancestor_id   bigint NOT NULL REFERENCES org_node(node_id),
    descendant_id bigint NOT NULL REFERENCES org_node(node_id),
    depth         integer NOT NULL CHECK (depth >= 0),
    PRIMARY KEY (ancestor_id, descendant_id)
);

CREATE INDEX org_closure_desc_idx
    ON org_closure(descendant_id, ancestor_id);
```

Every node has a self-row with depth zero. A direct parent-child relationship has depth one. If node 1 is an ancestor of node 6 at depth three, the table stores `(1, 6, 3)`.

The closure table makes reachability a normal indexed lookup, but it can be much larger than the node table. In a chain of n nodes, the number of ancestor-descendant pairs is $n(n+1)/2$, which is $O(n^2)$. In a balanced tree, the number of pairs is typically closer to $O(n \log n)$, but the exact count depends on shape. Moving a subtree requires updating many closure rows.

4.4 Materialized path: store the path on each node

A materialized path stores the sequence of ancestor identifiers (or a path-like label) with each node. A simple text illustration might be `/1/2/4/6/`. Descendants share the ancestor's prefix.

```sql
CREATE TABLE path_node (
    node_id bigint PRIMARY KEY,
    path    text NOT NULL UNIQUE,
    label   text NOT NULL
);

-- Example path values:
-- /1/
-- /1/2/
-- /1/2/4/
```

```
-- /1/2/4/6/
```

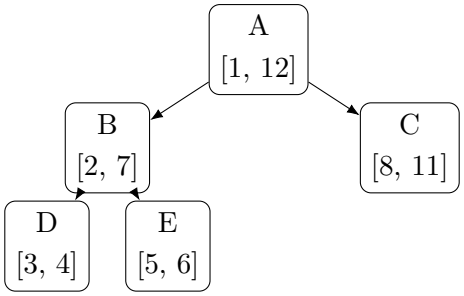
A prefix search can find descendants, but ordinary B-tree prefix behavior depends on collation, operator class, and pattern. In PostgreSQL, a `text_pattern_ops` index can help with anchored prefix searches under suitable conditions; alternatively use `ltree`, a PostgreSQL extension designed for label paths.

Materialized paths make path and subtree reads convenient, but moving a node requires rewriting the path of every descendant. Long paths can also become expensive for very deep hierarchies.

4.5 Nested sets: store subtree intervals

Nested sets assign each node a left and right number such that every descendant's interval is contained within its ancestor's interval. A node a is an ancestor of node b when:

$$lft(a) < lft(b) \quad \text{and} \quad rgt(b) < rgt(a).$$



A descendant query is a range query:

```sql
SELECT child.*
FROM nested_node AS parent
JOIN nested_node AS child
  ON child.lft > parent.lft
 AND child.rgt < parent.rgt
WHERE parent.node_id = 2
ORDER BY child.lft;
```

Nested sets can make subtree reads fast, but inserts and moves may require renumbering many rows. This is often a poor fit for frequently changing structures or very large trees with frequent edits. It is also less intuitive to maintain correctly than an adjacency list.

5 One Wide Table or Many Child Tables?

A second, independent design choice concerns the entity attributes, not just the edges. Suppose an organization contains boards, executives, departments, and employees.

5.1 One wide table with a type column

```
CREATE TABLE org_item (
    item_id bigint PRIMARY KEY,
    parent_id bigint REFERENCES org_item(item_id),
    item_type text NOT NULL,
    name text NOT NULL,
    -- attributes used by only some types
    board_term_end date,
    executive_title text,
    department_cost_center text,
    employee_salary numeric(12,2),
    employee_start_date date
);
```

This is sometimes called a single-table or table-per-hierarchy design. Every kind of node occupies the same table, and type-specific columns are null for rows where they do not apply.

Benefits

- One parent key and one recursive query work across the entire structure.
- A single table is convenient for generic tree traversal, search, reporting, and user interfaces.
- A node has one globally consistent identity and can be referenced by one foreign key.
- Fewer joins are needed to retrieve a mixed hierarchy.

Costs and risks

- The table can become very wide, with many null columns.
- Type-specific constraints become complicated: a salary should exist for employees but not for boards.
- Different node types may have very different lifecycle, security, and update rules.
- Schema changes for one subtype affect the shared table and can become operationally disruptive.
- A generic row can become a collection of weakly related attributes rather than a coherent entity.

A discriminator and check constraints can improve correctness:

```
ALTER TABLE org_item ADD CONSTRAINT org_item_type_check
CHECK (
  (item_type = 'employee'
     AND employee_salary IS NOT NULL
     AND employee_start_date IS NOT NULL)
  OR
  (item_type <> 'employee'
     AND employee_salary IS NULL
     AND employee_start_date IS NULL)
```

```
);
```

In a real schema, similar conditions may be needed for every subtype. Keep such constraints readable; a very large collection of subtype-specific columns and checks is a signal to consider another design.

5.2 Separate tables for different entity types

An alternative is to create separate tables for boards, executives, departments, and employees. This is a good fit when each type has substantially different attributes and behavior.

The difficulty is that a single foreign key cannot ordinarily reference one of several unrelated tables. A shared supertype table solves that problem:

```
CREATE TABLE org_entity (
    entity_id bigint PRIMARY KEY,
    parent_id bigint REFERENCES org_entity(entity_id),
    entity_kind text NOT NULL,
    display_name text NOT NULL
);

CREATE TABLE employee_detail (
    entity_id bigint PRIMARY KEY REFERENCES org_entity(entity_id),
    salary numeric(12,2) NOT NULL,
    start_date date NOT NULL
);

CREATE TABLE department_detail (
    entity_id bigint PRIMARY KEY REFERENCES org_entity(entity_id),
    cost_center text NOT NULL
);
```

This is class-table inheritance applied to the node types. The generic hierarchy lives in `org_entity`; subtype-specific data lives in detail tables.

Benefits: strong separation of attributes, fewer irrelevant nulls, and clearer type-specific constraints. **Costs**: retrieving a full entity may require joins, and the database must ensure that the subtype rows agree with `entity_kind`. The design can also require more complex insert and delete transactions.

5.3 JSON / JSONB for flexible attributes or embedded trees

JSON is useful when attributes vary substantially by node type, when the source data is naturally document-shaped, or when the hierarchy is usually retrieved and updated as a whole. PostgreSQL's `jsonb` supports indexing and operators, but it does not automatically provide the same referential integrity as normalized relational tables.

```
CREATE TABLE org_document (
    document_id bigint PRIMARY KEY,
```

```
    document jsonb NOT NULL
);

INSERT INTO org_document(document_id, document)
VALUES (1, '{
  "name": "Board",
  "kind": "board",
  "children": [
    {"name": "CEO", "kind": "executive",
     "children": [
       {"name": "CTO", "kind": "executive", "children": []}
     ]}
  ]
}'::jsonb);

-- Top-level fields
SELECT document ->> 'name' AS root_name
FROM org_document
WHERE document_id = 1;

-- Expand the immediate children of the root
SELECT child.value ->> 'name' AS child_name
FROM org_document AS d
CROSS JOIN LATERAL
     jsonb_array_elements(d.document -> 'children') AS child(value)
WHERE d.document_id = 1;
```

For a nested JSON document, each level can be expanded with another `jsonb_array_elements` call. The number of expansions is fixed in the SQL text unless the application constructs a query or a recursive function is used. A recursive CTE can also be used after first converting the document into relational edges, but JSON itself is not a magic recursive-query engine.

JSON advantages: flexible schema, natural representation of nested API payloads, convenient whole-document retrieval, and fewer joins when the application consumes the entire document.

JSON disadvantages: foreign keys cannot directly enforce references between arbitrary embedded nodes; updating one deeply nested item can require replacing or reconstructing a large value; indexing individual nested attributes takes planning; and querying across millions of independently managed nodes is often easier with normalized rows.

A useful rule: store a document as JSON when the document is the unit of ownership and access. Store nodes and edges as rows when individual nodes and relationships must be independently queried, constrained, updated, joined, or referenced.

6 Navigating an Adjacency List with Recursive SQL

PostgreSQL supports recursive common table expressions using `WITH RECURSIVE`. A recursive CTE has an anchor query, which establishes the starting row or rows, and a recursive term, which

repeatedly joins the current result to the next level.

6.1 Find all descendants

Assume the `org_node` adjacency-list table defined earlier. Find every node below the CTO (node 4):

```
WITH RECURSIVE subtree AS (
    -- Anchor: start at the selected node
    SELECT node_id, parent_id, node_name, 0 AS depth
    FROM org_node
    WHERE node_id = 4

    UNION ALL

    -- Recursive term: find children of the current level
    SELECT child.node_id, child.parent_id,
           child.node_name, parent.depth + 1
    FROM org_node AS child
    JOIN subtree AS parent
      ON child.parent_id = parent.node_id
)
SELECT node_id, parent_id, node_name, depth
FROM subtree
ORDER BY depth, node_id;
```

The anchor returns the CTO at depth zero. The recursive term finds its children at depth one, then their children at depth two, and so on, until an iteration produces no rows. The result includes the starting node. Add `WHERE depth > 0` in the outer query if only strict descendants are wanted.

6.2 Find the ancestors of a node

To walk upward, join the current row to its parent instead of joining children to the current row:

```
WITH RECURSIVE ancestors AS (
    SELECT node_id, parent_id, node_name, 0 AS distance
    FROM org_node
    WHERE node_id = 6

    UNION ALL

    SELECT parent.node_id, parent.parent_id,
           parent.node_name, child.distance + 1
    FROM org_node AS parent
    JOIN ancestors AS child
      ON parent.node_id = child.parent_id
)
```

```
SELECT node_id, node_name, distance
FROM ancestors
ORDER BY distance;
```

Here the starting department is at distance zero, its manager is at distance one, and so forth. The final row, if one exists, is the root.

6.3 Build a readable path

A path can be accumulated as the recursive query proceeds:

```
WITH RECURSIVE subtree AS (
    SELECT node_id, parent_id, node_name,
           0 AS depth,
           ARRAY[node_id] AS id_path,
           node_name::text AS name_path
    FROM org_node
    WHERE node_id = 1

    UNION ALL

    SELECT child.node_id, child.parent_id, child.node_name,
           parent.depth + 1,
           parent.id_path || child.node_id,
           parent.name_path || ' / ' || child.node_name
    FROM org_node AS child
    JOIN subtree AS parent
      ON child.parent_id = parent.node_id
)
SELECT node_id, depth, name_path
FROM subtree
ORDER BY id_path;
```

The array path is useful for deterministic ordering and cycle checks. For a graph traversal, where a node may be reached by several paths, the query may return the same node more than once. Decide whether the desired result is paths, reachable nodes, or shortest paths; those are different questions.

6.4 Cycle protection

A valid tree should not contain a cycle, but the database's self-referencing foreign key does not prevent one. For example, changing node 2's parent to node 6 could create a loop. A defensive traversal can avoid revisiting a node on the current path:

```
WITH RECURSIVE walk AS (
    SELECT node_id, parent_id, ARRAY[node_id] AS visited,
           false AS cycle_found
    FROM org_node
```

```
    WHERE node_id = 1

    UNION ALL

    SELECT child.node_id, child.parent_id,
           walk.visited || child.node_id,
           child.node_id = ANY(walk.visited)
    FROM org_node AS child
    JOIN walk ON child.parent_id = walk.node_id
    WHERE NOT walk.cycle_found
)
SELECT * FROM walk;
```

This illustrates path-local cycle detection. In a production tree, it is better to prevent invalid moves as part of the write transaction, rather than relying only on readers to detect bad data. Under concurrent updates, validation must be designed so two transactions cannot each pass a check and jointly create a cycle. Locking the relevant tree or using a serialization strategy may be necessary.

PostgreSQL also supports a SQL-standard-style `CYCLE` clause in recursive CTEs in current releases. The explicit visited-array technique above is more portable across PostgreSQL versions and makes the logic visible to students.

6.5 Recursive query over a DAG

Using the scheduler edge convention where `from_id` is a prerequisite and `to_id` is a dependent task, find all tasks that are downstream of task 10:

```
WITH RECURSIVE downstream AS (
    SELECT 10::bigint AS node_id,
           ARRAY[10::bigint] AS path,
           0 AS depth

    UNION ALL

    SELECT e.to_id,
           d.path || e.to_id,
           d.depth + 1
    FROM downstream AS d
    JOIN graph_edge AS e
      ON e.from_id = d.node_id
    WHERE NOT e.to_id = ANY(d.path)
)
SELECT DISTINCT node_id
FROM downstream
WHERE depth > 0;
```

The path test prevents an individual traversal path from looping forever if bad cyclic data exists. Because a DAG can have multiple paths to the same node, the same node can still appear several

times before the outer `DISTINCT`. If only reachability is required, duplicates can be removed; if path enumeration is required, do not remove them.

For scheduler execution, reachability is not the only concern. A task is ready only when all its prerequisites are complete. A common approach is to find tasks with no unfinished incoming prerequisites:

```
SELECT t.node_id, t.label
FROM graph_node AS t
WHERE NOT EXISTS (
    SELECT 1
    FROM graph_edge AS e
    JOIN task_status AS prerequisite
      ON prerequisite.node_id = e.from_id
    WHERE e.to_id = t.node_id
      AND prerequisite.status <> 'complete'
);
```

This assumes a task-status table and the stated edge direction. Real schedulers also need concurrency control so that two workers do not claim the same ready task.

7 Traversal Without Recursive SQL: Iteration

Recursion is expressive, but it is not the only way to navigate a hierarchy. An application can issue repeated SQL statements, keeping the current frontier (the nodes at the current level) in a temporary table. This is breadth-first traversal.

7.1 Level-by-level traversal with a frontier

The following is PostgreSQL-flavored pseudocode. It uses temporary tables and a loop in a stored procedure or application program:

```
CREATE TEMP TABLE frontier (
    node_id bigint PRIMARY KEY,
    depth integer NOT NULL
) ON COMMIT DROP;

CREATE TEMP TABLE visited (
    node_id bigint PRIMARY KEY,
    depth integer NOT NULL
) ON COMMIT DROP;

INSERT INTO frontier VALUES (1, 0);
INSERT INTO visited  VALUES (1, 0);

-- Repeat until frontier is empty:
INSERT INTO visited(node_id, depth)
SELECT child.node_id, min(frontier.depth + 1)
```

```
FROM frontier
JOIN org_node AS child
  ON child.parent_id = frontier.node_id
LEFT JOIN visited AS seen
  ON seen.node_id = child.node_id
WHERE seen.node_id IS NULL
GROUP BY child.node_id;

TRUNCATE frontier;
INSERT INTO frontier(node_id, depth)
SELECT node_id, depth
FROM visited
WHERE depth = (SELECT max(depth) FROM visited);
```

The code communicates the idea, but the frontier replacement should be implemented carefully in a real loop: preserve the old frontier in a separate table or use a second temporary table, then swap or truncate/insert. The key operation is always the same: expand the current frontier by one edge, exclude already visited nodes, and make the newly found nodes the next frontier.

A clearer loop outline is:

```
frontier := {start_node}
visited  := {start_node}
depth := 0
WHILE frontier is not empty:
    next_frontier := children(frontier) MINUS visited
    add next_frontier to visited with depth + 1
    frontier := next_frontier
    depth := depth + 1
END WHILE
```

For a tree, no node is reached by two paths, so the visited set is mostly defensive. For a DAG or general graph, it prevents repeated expansion and guarantees termination when the reachable graph is finite. If shortest path distances are needed in an unweighted graph, breadth-first traversal gives the minimum number of edges from the start node.

7.2 Pointer jumping: double the distance each round

A more specialized technique is pointer jumping (also called repeated squaring in some contexts). It is especially easy to understand for a rooted tree where every node has at most one parent and we want to find ancestors.

Initially, each node knows its parent, which is its 2^0-th ancestor. In each round, replace the pointer with the pointer's pointer:

$$up_{k+1}(v) = up_k(up_k(v)).$$

After round 1, each node knows its ancestor two edges above. After round 2, it knows the ancestor four edges above. After round 3, eight edges above, and so on. After k rounds, a node can jump up to 2^k levels.

Example table after successive rounds:

Node	Initial parent (2^0)	Round 1 (2^1)	Round 2 (2^2)	Round 3 (2^3)
8	7	6	4	1
7	6	5	2	NULL
6	5	4	1	NULL
5	4	3	NULL	NULL

The exact values depend on the sample tree; the important property is that the maximum covered distance doubles each round.

A SQL implementation can materialize the current jump pointers in a temporary table. Assume `org_node(node_id,parent_id)` and a table with one row per node:

```
CREATE TEMP TABLE jump (
    node_id bigint PRIMARY KEY,
    up_id   bigint
) ON COMMIT DROP;

INSERT INTO jump(node_id, up_id)
SELECT node_id, parent_id
FROM org_node;
```

One round of pointer jumping is:

```
CREATE TEMP TABLE next_jump (
    node_id bigint PRIMARY KEY,
    up_id bigint
) ON COMMIT DROP;

INSERT INTO next_jump(node_id, up_id)
SELECT j.node_id, p.up_id
FROM jump AS j
LEFT JOIN jump AS p
  ON p.node_id = j.up_id;

TRUNCATE jump;
INSERT INTO jump SELECT * FROM next_jump;
TRUNCATE next_jump;
```

Repeat that round $\lceil \log_2(H+1) \rceil$ times, where H is the maximum tree height, or simply repeat until all pointers become null. After k rounds, `up_id` represents the ancestor 2^k edges above the node (or null if the root is reached first).

This table alone does not return every ancestor. It provides fast powers-of-two jumps. To find the ancestor at an arbitrary distance d, use the binary representation of d and apply the corresponding jump levels. A practical implementation stores `up[node, k]` for each node and each jump level, then answers a d-ancestor query in $O(\log d)$ pointer applications.

Important trade-off: building a full jump table requires roughly $O(n \log H)$ stored pointers and $O(n \log H)$ work. This can be attractive for a large, mostly static tree with many ancestor

queries. It may be wasteful if the tree is small, changes constantly, or only needs occasional traversal. Updating a parent can invalidate jump information for many descendants.

7.3 Pointer jumping is not the same as enumerating descendants

Pointer jumping is naturally suited to moving upward through a single-parent structure. It does not magically solve arbitrary DAG reachability. In a DAG, a node can have multiple parents, so one pointer per node is insufficient. A general transitive-closure computation can require many more pairs and can approach quadratic size. Choose the algorithm based on the question: ancestor-at-distance, reachability, shortest path, or enumeration of all paths.

8 Choosing a Representation and Algorithm at Scale

No representation is best for every workload. The shape of the data and the frequency of reads versus writes matter as much as the total row count.

8.1 Tree shape matters: wide versus deep

A **wide, shallow tree** has many children at each level but few levels. For example, a catalog with one root, thousands of categories, and millions of products at the leaves. A recursive traversal has few iterations, but one iteration may produce a very large frontier. Indexing `parent_id` is important.

A **deep, narrow tree** has few children per node but many levels. A linked list is the extreme case. Recursive traversal may require many iterations and may build long path values. A closure table or jump pointers can make repeated ancestor queries faster, but both add storage and update complexity.

8.2 Approximate cost intuition

Let n be the number of nodes, h the height, and m the number of edges.

Method	Storage overhead	Typical strength	Main weakness
Adjacency list	$O(n)$ nodes plus $O(n)$ tree edges	Simple writes; flexible traversal	Descendant queries require walking edges
Edge table	$O(n+m)$	DAGs and general graphs	Traversal and cycle checks can be expensive
Closure table	Up to $O(n^2)$ pairs	Fast ancestor/descendant membership	Large storage and costly subtree moves
Materialized path	Path length per node; up to $O(nh)$ characters/labels	Prefix and path queries	Moving a subtree rewrites descendant paths
Nested sets	$O(n)$ interval values	Fast subtree range reads	Inserts and moves can renumber many rows
Jump pointers	$O(n \log h)$ pointers	Fast ancestor-at-distance queries	Extra storage; maintenance after moves
JSON document	One document per aggregate	Whole-document read/write	Weak node-level relational integrity; large-document updates

These are asymptotic intuitions, not universal performance guarantees. Index size, row width, cache behavior, data skew, query plans, and write concurrency all matter. Benchmark with representative data.

8.3 What happens at a billion nodes?

At a billion nodes, an elegant algorithm with quadratic intermediate results is not an option. Even $O(n \log n)$ auxiliary storage may be substantial. A design that works beautifully for a few thousand nodes may become operationally difficult at hundreds of millions.

Practical principles:

- Keep the base representation compact. An indexed adjacency list or edge table is often the best starting point.
- Index the traversal direction. For downward traversal, index the child lookup key (`parent_id` or `from_id`); for upward traversal, index the reverse edge key (`to_id`) when needed.
- Scope every query. Include tenant, organization, or graph identifiers in keys and indexes when data contains independent graphs. Avoid traversing across unrelated tenants.
- Do not materialize all paths in a graph unless the workload truly needs them. The number of paths can be exponential even when the graph itself is modest.
- Avoid storing a full transitive closure for a huge, frequently changing graph without estimating its row count and update cost first.
- Consider partitioning very large edge tables by tenant, graph, or another workload-aligned key. Partitioning is not automatic speed: queries must be able to prune irrelevant partitions.

- Use batch processing or asynchronous maintenance for derived structures such as closure tables, materialized paths, and jump tables when immediate consistency is not required.
- Measure the largest and most skewed trees, not just the average tree. A single extremely deep or high-degree node can dominate resource use.

For a billion-node graph, a recursive CTE may still be useful for a small, bounded neighborhood or a carefully scoped traversal. It is not a promise that the database can cheaply enumerate the entire graph. A query that returns 300 million rows is expensive regardless of syntax.

8.4 Read-heavy versus write-heavy workloads

- **Write-heavy, frequently reorganized tree:** adjacency list is usually easiest to maintain. Traverse when needed; cache selectively.
- **Read-heavy tree with frequent descendant queries:** consider closure tables, materialized paths, or nested sets, after estimating update costs.
- **Mostly static tree with many ancestor-at-distance queries:** jump pointers can be effective.
- **DAG workflow:** edge table is the natural representation. Use indexed incoming and outgoing edges; validate acyclicity when dependencies are changed.
- **Document owned and read as a unit:** JSONB may be appropriate, especially at an application boundary. Normalize when nodes need independent identity, references, constraints, or high-volume relational queries.

9 Integrity Rules and Operational Concerns

A schema is not complete merely because its foreign keys are valid. Hierarchical data often has rules that span multiple rows.

9.1 Rules to define explicitly

- Can a node have zero, one, or many parents?
- Are multiple roots allowed?
- Are cycles forbidden, allowed, or conditionally allowed?
- Can a node have more than one edge type to the same neighbor?
- Must parent and child belong to the same tenant or organization?
- What happens when a parent is deleted: reject, cascade, detach children, or re-parent them?
- Can a node be moved? If so, what derived structures must be updated?

9.2 A simple same-tree constraint

For a multi-tenant forest, include the tenant identifier in the parent relationship. One pattern is a composite foreign key:

```
CREATE TABLE tenant_node (
    tenant_id bigint NOT NULL,
    node_id bigint NOT NULL,
    parent_id bigint,
    label text NOT NULL,
    PRIMARY KEY (tenant_id, node_id),
    FOREIGN KEY (tenant_id, parent_id)
      REFERENCES tenant_node(tenant_id, node_id)
);

CREATE INDEX tenant_node_parent_idx
    ON tenant_node(tenant_id, parent_id);
```

This prevents a node in one tenant from pointing to a parent in another tenant. It still does not prevent cycles.

9.3 Moving a subtree safely

To move node x below new parent p in a tree, check that p is not x and is not already a descendant of x. Otherwise the move would create a cycle. Conceptually:

```
-- Before changing x.parent_id, find whether p is below x.
WITH RECURSIVE descendants(node_id) AS (
    SELECT node_id
    FROM org_node
    WHERE node_id = :x

    UNION ALL

    SELECT c.node_id
    FROM org_node AS c
    JOIN descendants AS d ON c.parent_id = d.node_id
)
SELECT EXISTS (
    SELECT 1 FROM descendants WHERE node_id = :p
) AS would_create_cycle;
```

If the result is true, reject the move. In production, perform validation and update in one transaction with a concurrency strategy that prevents conflicting simultaneous moves. For closure tables, nested sets, materialized paths, or jump pointers, the move must also update the derived representation.

10 A Practical Decision Framework

When designing a hierarchical database, proceed in this order:

1. **Name the relationship.** Is it containment, supervision, dependency, classification, or a general connection?
2. **Identify the graph shape.** One root, a forest, a DAG, or a cyclic graph? Can a node have multiple parents?
3. **Define the operations.** Do users mostly read children, descendants, ancestors, paths, reachability, or shortest paths? How often are nodes moved or edges changed?
4. **Choose the base storage.** Adjacency list for a simple tree; edge table for a DAG or graph; JSON for document-shaped ownership and access; subtype tables for is-a modeling.
5. **Add derived structures only for measured needs.** Closure tables, nested sets, materialized paths, and jump pointers trade write complexity and storage for faster reads.
6. **Plan for integrity and scale.** Define cycle rules, indexes, tenant boundaries, deletion behavior, transaction isolation, and realistic maximum size/depth/degree.