Advanced topics in Databases

Hellenic Mediterranean University

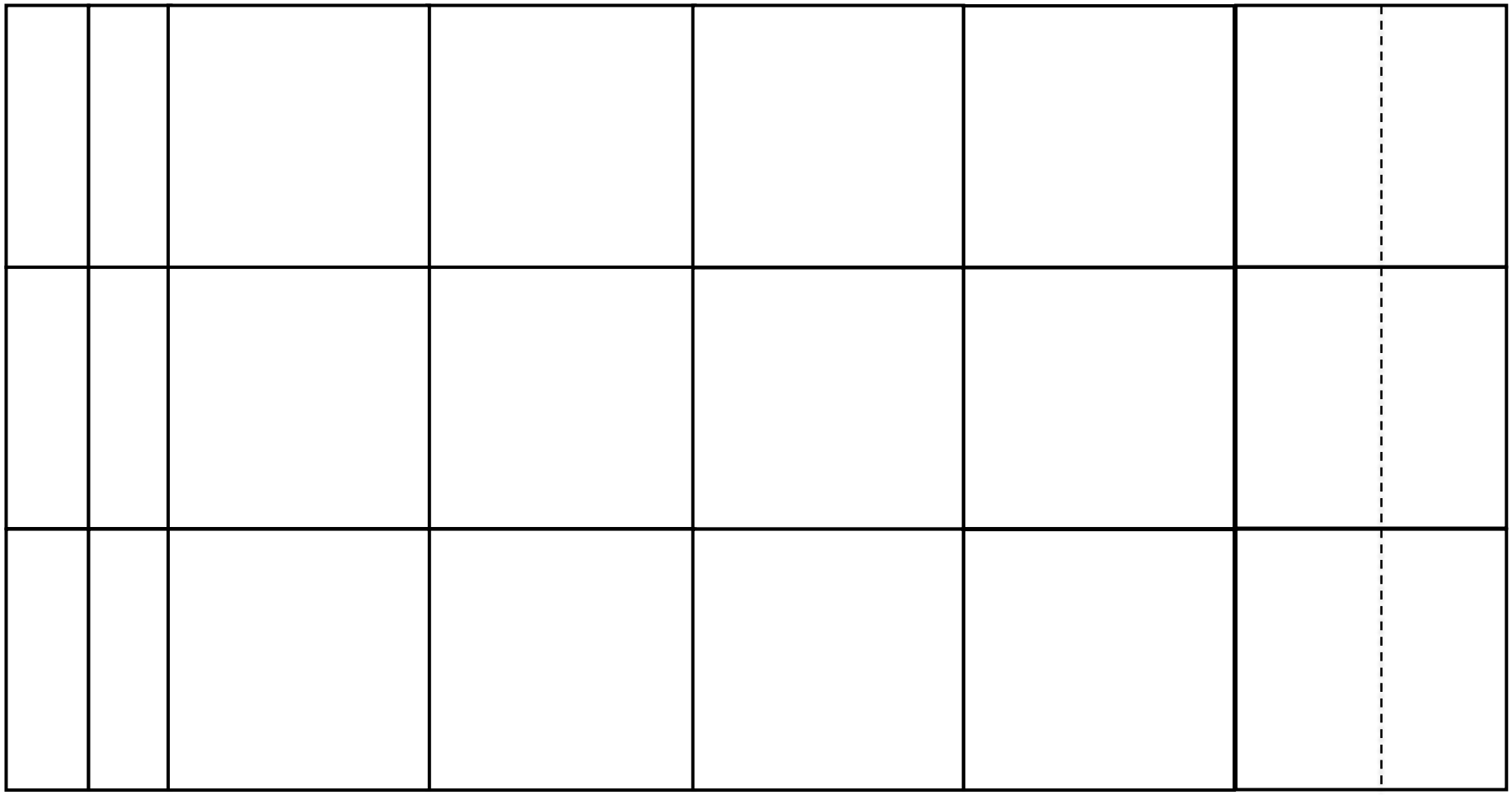
Prof. Demos Akoumianakis (da@hmu.gr)

Agenda

- ✓ *Object-oriented extensions and object-relational systems*
  - *Postgres but there are other approaches too, such as SQL3, Oracle*
  - *Focus on new data types*
    - *Enum, multivalue, composite types*
    - *Application in assignment 2 (due after Easter break)*
    - *XML/JSON data types*
    - *Specialization hierarchies*
- Today
  - Hierarchical data and trees
    - Comparison of adjacency list, closure table and nested set models
  - Recursive queries

So far – the relational view

- Table – a set of tuples



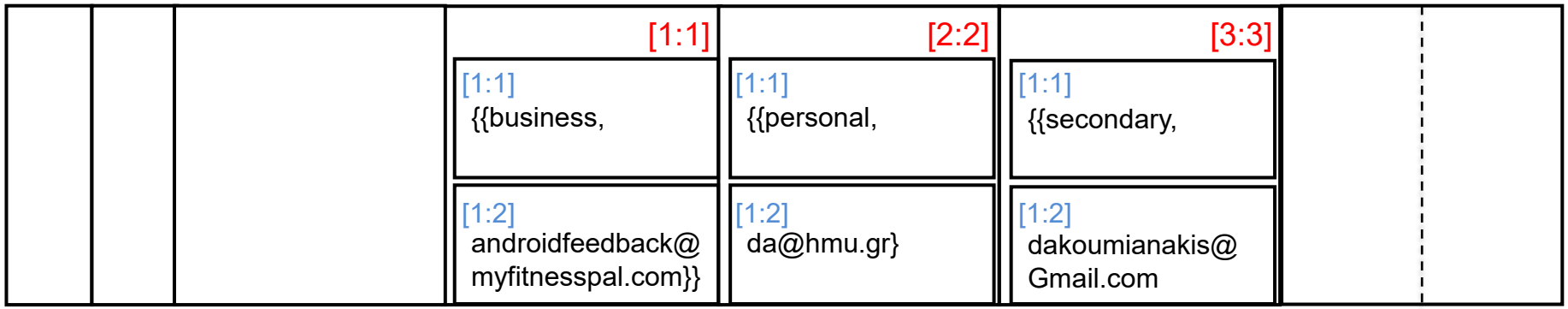
The object-relational view

- Tuples may have elements

			{{business,	{{personal,	{{secondary,		
			androidfeedback@ myfitnesspal.com}}	da@hmu.gr}	dakoumianakis@ Gmail.com		

The object-relational view (cont.)

- Need for indexing to identify elements

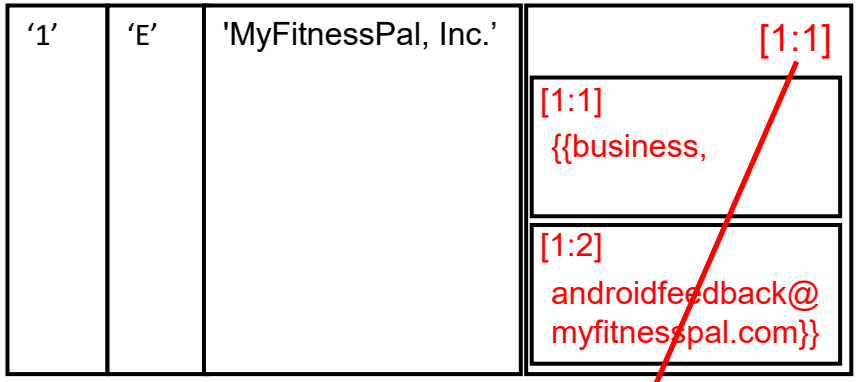


- Approach relying on complex data types

```
CREATE TYPE Address AS (
    Line VARCHAR(90),
    Zip_Code VARCHAR(30));
CREATE TABLE DEVELOPER (
    Dev_ID TEXT NOT NULL,
    Dev_Type Affiliation,
    Name TEXT NOT NULL UNIQUE,
    email TEXT [][] NOT NULL,
    Dev_Address Address NOT NULL,
    PRIMARY KEY(Dev_ID));
```

The result …

- Maintaining consistency with SQL




```
/* Εισαγωγή δεδομένων στον πίνακα DEVELOPER */
INSERT INTO DEVELOPER (Dev_ID, Dev_Type, Name, email, Dev_Address) VALUES (
    '1', 'E', 'MyFitnessPal, Inc.',
    '{  {business, androidfeedback@myfitnesspal.com},
```

The result …

- Maintaining consistency with SQL

‘1’	‘E’	'MyFitnessPal, Inc.’	[1:1]	[2:2]
			[1:1] {{business,	[1:1] {{personal,
			[1:2] androidfeedback@ myfitnesspal.com}}	[1:2] da@hmu.gr}

```
/* Εισαγωγή δεδομένων στον πίνακα DEVELOPER */
INSERT INTO DEVELOPER (Dev_ID, Dev_Type, Name, email, Dev_Address) VALUES (
    '1', 'E', 'MyFitnessPal, Inc.',
    '{  {business, androidfeedback@myfitnesspal.com},
        {personal, da@hmu.gr},
```

The result …

- Maintaining consistency with SQL

			[1:1]	[2:2]	[3:3]
‘1’	‘E’	'MyFitnessPal, Inc.’	[1:1] {{business,	[1:1] {{personal,	[1:1] {{secondary,
			[1:2] androidfeedback@ myfitnesspal.com}}	[1:2] da@hmu.gr}	[1:2] dakoumianakis@ Gmail.com

```
/* Εισαγωγή δεδομένων στον πίνακα DEVELOPER */
INSERT INTO DEVELOPER (Dev_ID, Dev_Type, Name, email, Dev_Address) VALUES (
    '1', 'E', 'MyFitnessPal, Inc.',
    '{  {business, androidfeedback@myfitnesspal.com},
        {personal, da@hmu.gr},
        {secondary, dakoumianakis@gmail.com}
```

The result …

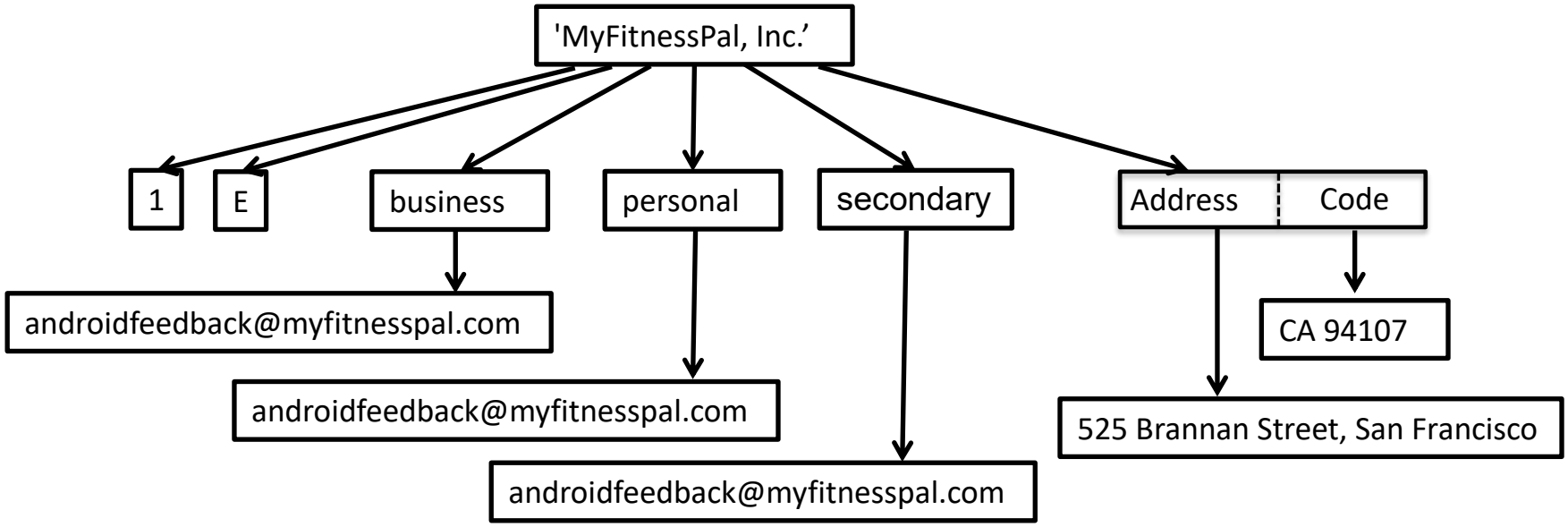
- Maintaining consistency with SQL

‘1’	‘E’	'MyFitnessPal, Inc.’	[1:1] [1:1] {{business, [1:2] androidfeedback@ myfitnesspal.com}}	[2:2] [1:1] {{personal, [1:2] da@hmu.gr}	[3:3] [1:1] {{secondary, [1:2] dakoumianakis@ Gmail.com	525 Brannan Street, San Francisco	CA 94107

```
/* Εισαγωγή δεδομένων στον πίνακα DEVELOPER */
INSERT INTO DEVELOPER (Dev_ID, Dev_Type, Name, email, Dev_Address) VALUES (
    '1', 'E', 'MyFitnessPal, Inc.',
    '{  {business, androidfeedback@myfitnesspal.com},
        {personal, da@hmu.gr},
        {secondary, dakoumianakis@gmail.com}
    }',
    ROW('525 Brannan Street San Francisco', 'CA 94107'));
```

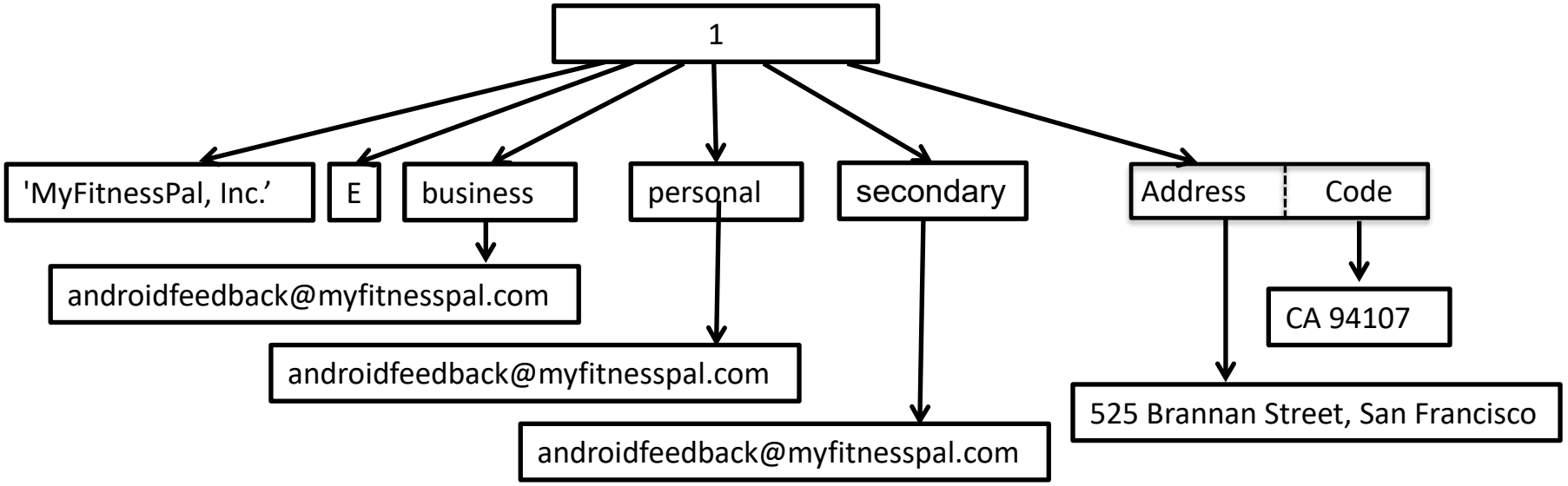
The hierarchical view

- Hierarchical data depict a structure whereby each data item / entity can have 0, 1 or many children but 0 or 1 parent



Choice of model

- Hierarchies vs. Complex Data Types



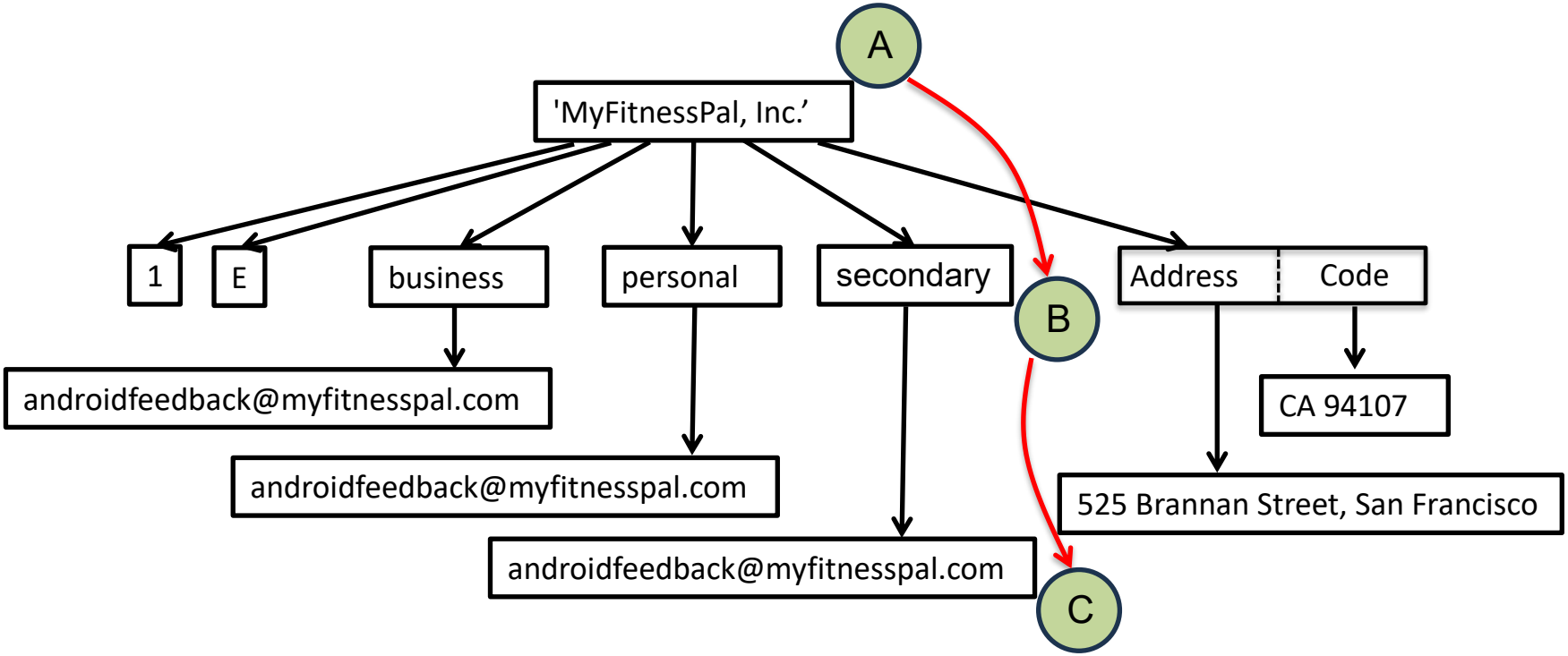
'1'	'E'	'MyFitnessPal, Inc.'	[1:1]	[2:2]	[3:3]	525 Brannan Street, San Francisco	CA 94107
			[1:1] {{business,	[1:1] {{personal,	[1:1] {{secondary,		
			[1:2] androidfeedback@myfitnesspal.com}}	[1:2] da@hmu.gr}	[1:2] dakoumianakis@Gmail.com		

Properties of hierarchies

- Properties (e.g. ancestor-descendant relation)
  - Transitive (μεταβατική)
    - If A is an ancestor of B and B is an ancestor of C, then A is an ancestor of C
  - Antisymmetric
    - If A is an ancestor of B, then B is never an ancestor of A
  - Irreflexive
    - A is never an ancestor to itself

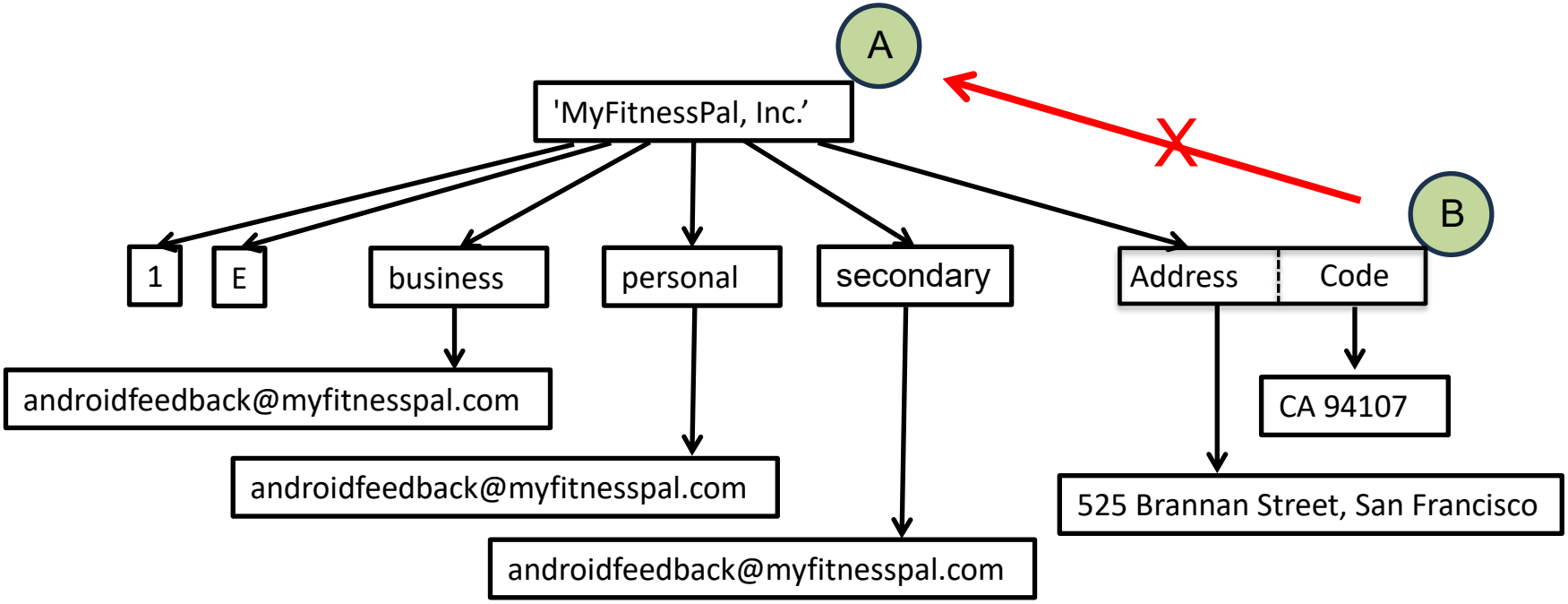
Properties of hierarchies (cont.)

- Properties (e.g. ancestor-descendant relation)
  - Transitive (μεταβατική)
    - If A is an ancestor of B and B is an ancestor of C, then A is an ancestor of C



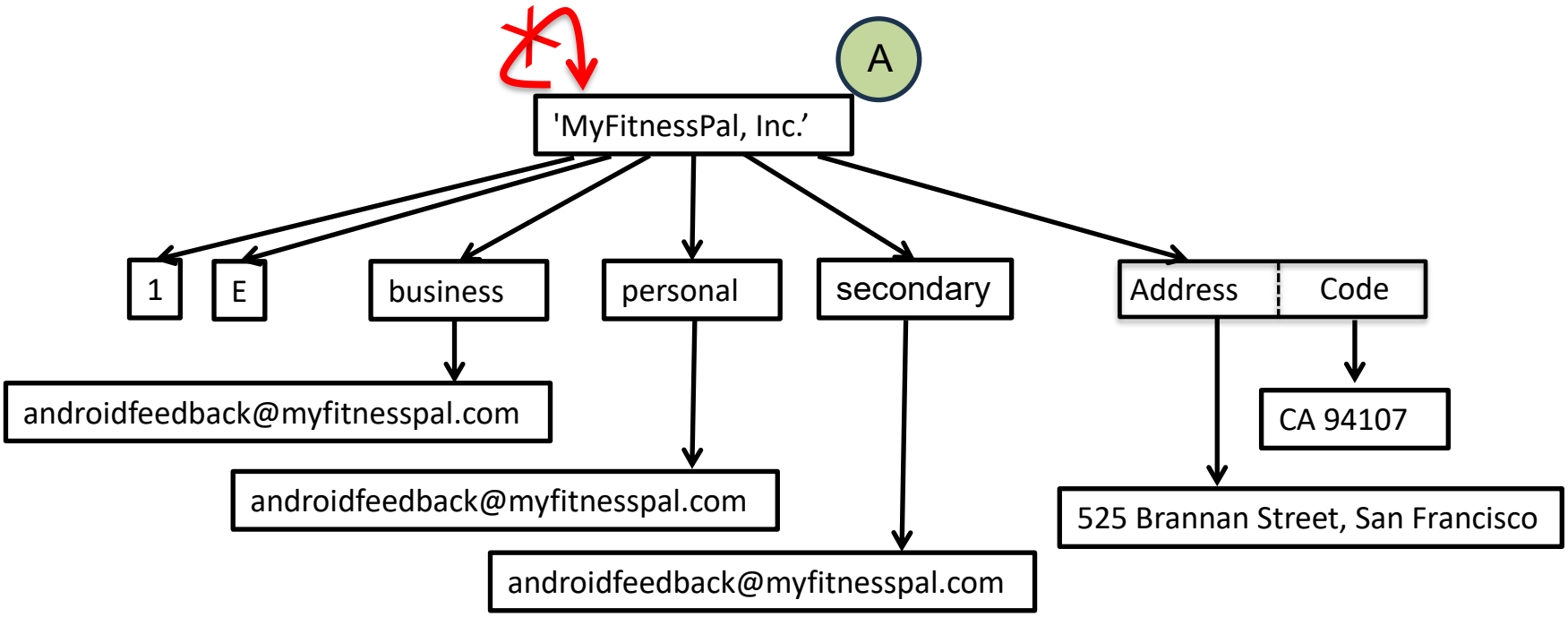
Properties of hierarchies (cont.)

- Properties (e.g. ancestor-descendant relation)
  - Antisymmetric (αντι-συμμετρική)
    - If A is an ancestor of B, then B is never an ancestor of A



Properties of hierarchies (cont.)

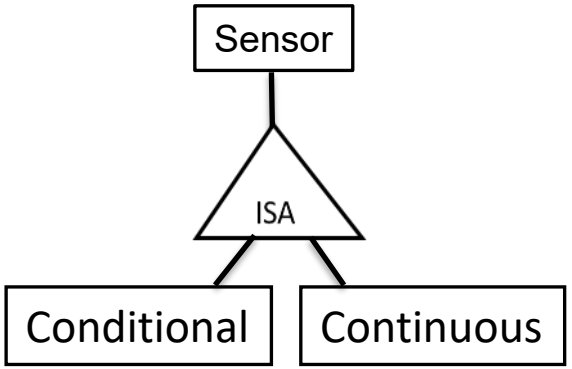
- Properties (e.g. ancestor-descendant relation)
  - Irreflexive (μη ανακλαστική)
    - A is never an ancestor to itself



Important point

- Hierarchical data *should not* be confused with (or interpretated as) *specialization hierarchies* (cf. Assignment 3)

```
CREATE TABLE sensor (
      id SERIAL PRIMARY KEY,
      sensorName VARCHAR(64));

CREATE TABLE Continuous (data json)
      INHERITS (sensor);
CREATE TABLE Periodic (data xml)
      INHERITS (sensor);
CREATE TABLE Conditional (  time timestamp,
       data measurement)   INHERITS (sensor);
```



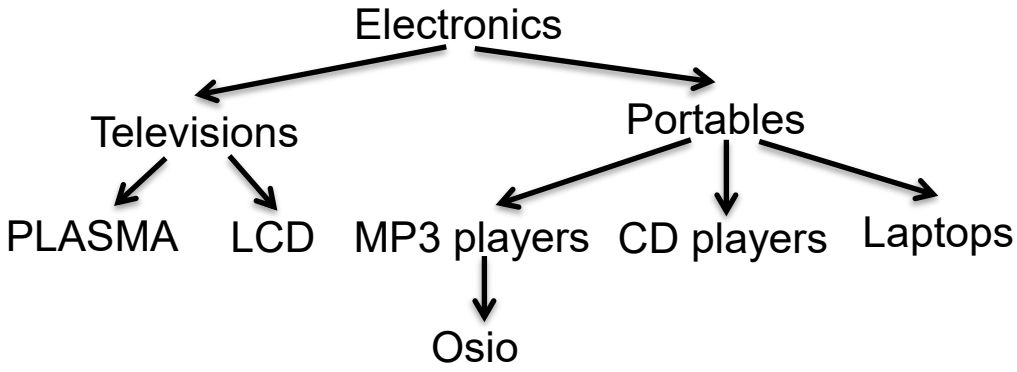
Data models and patterns for implementing hierarchies

Problem and solutions

- There have been several proposals for implementing hierarchies, each having pros and cons
  - Adjacency model
    - Single table solution
  - Closure table
    - Two table solution
  - Nested sets model
    - Single table solution

Adjacency model

- In this model, each node in the hierarchy is stored as a tuple in a relation with a pointer to its immediate parent
- An example

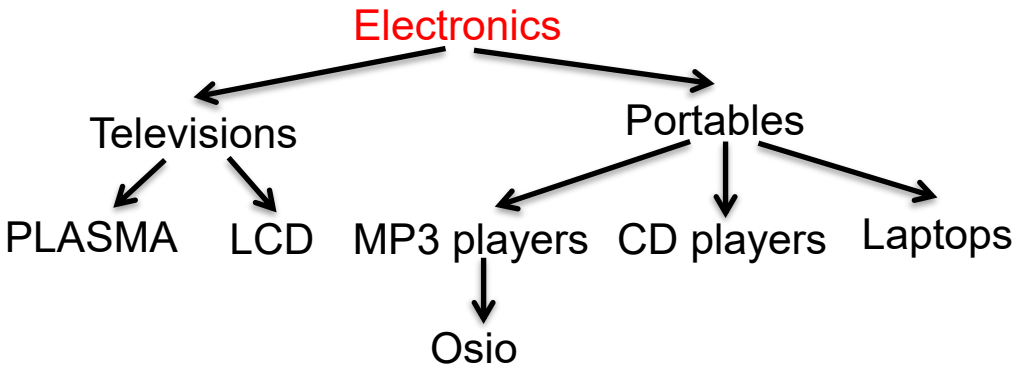


- One table with a foreign key referring to itself (parent_id)

```
CREATE TABLE categories (
        id serial PRIMARY KEY,
        title VARCHAR (50) UNIQUE
        parent_id INT DEFAULT NULL,
        description JSON,
        FOREIGN KEY (parent_id) REFERENCES categories (id));
```

Adjacency model

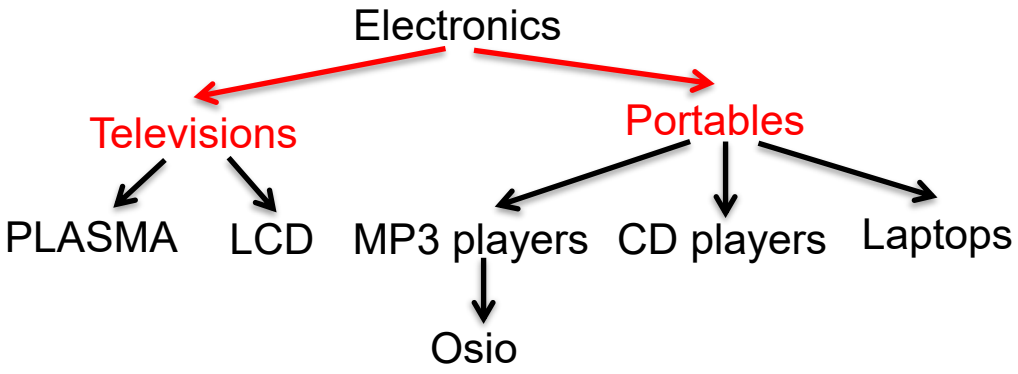
- An example



id [PK] integer	title character varying (50)	parent_id integer
1	Electronics	[null]

Adjacency model

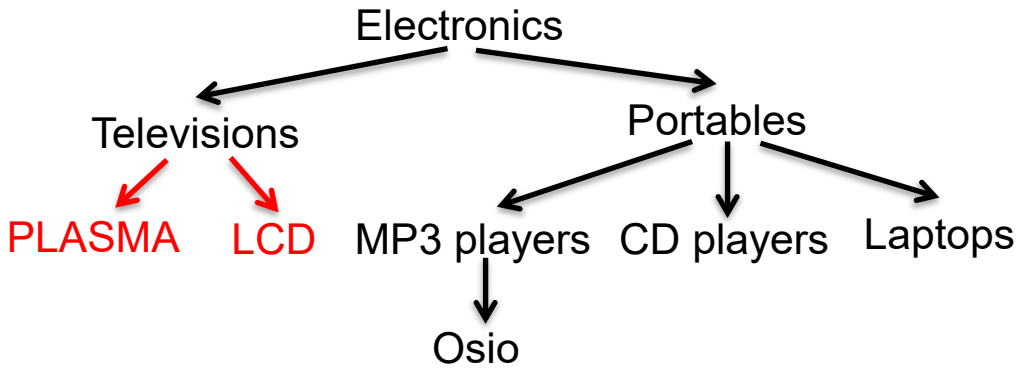
- An example



id [PK] integer	title character varying (50)	parent_id integer
1	Electronics	[null]
2	Televisions	1
3	Portables	1

Adjacency model

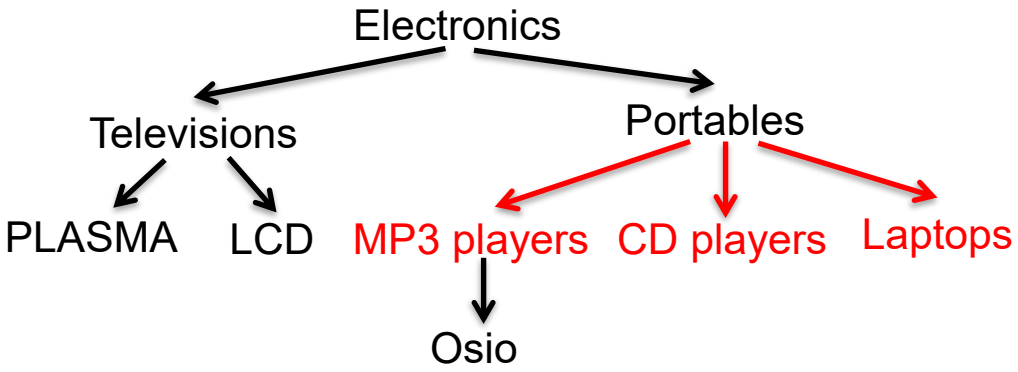
- An example



id [PK] integer	title character varying (50)	parent_id integer
1	Electronics	[null]
2	Televisions	1
3	Portables	1
4	Plasma	2
5	LCD	2

Adjacency model

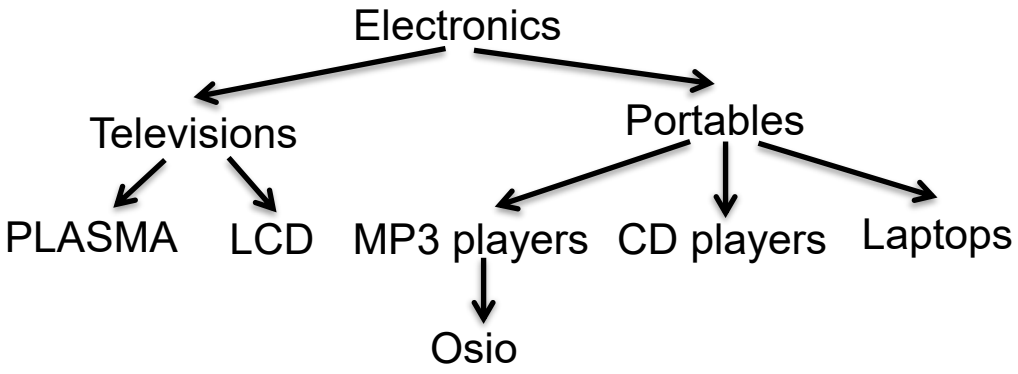
- An example



id [PK] integer	title character varying (50)	parent_id integer
1	Electronics	[null]
2	Televisions	1
3	Portables	1
4	Plasma	2
5	LCD	2
6	MP3	3
7	CD	3
8	Laptops	3

Adjacency model

- An example



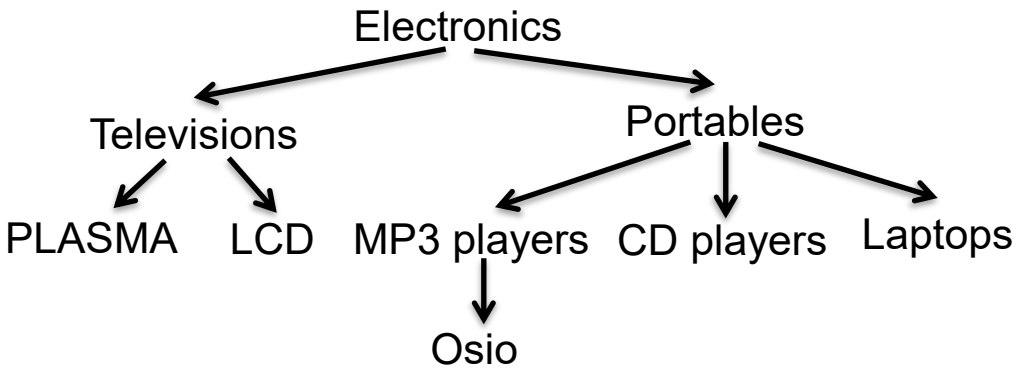
id [PK] integer	title character varying (50)	parent_id integer
1	Electronics	[null]
2	Televisions	1
3	Portables	1
4	Plasma	2
5	LCD	2
6	MP3	3
7	CD	3
8	Laptops	3
9	Osio	6

Adjacency list model

- In summary

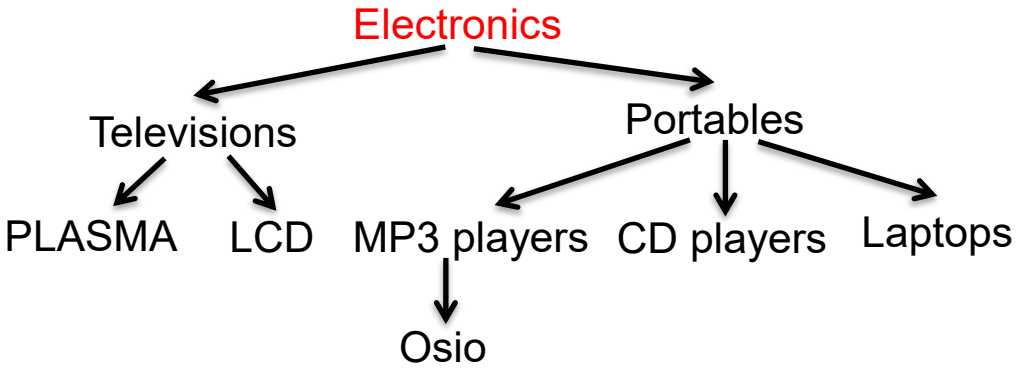
```
CREATE TABLE categories (
        id serial PRIMARY KEY,
        title VARCHAR (50) UNIQUE
        parent_id INT DEFAULT NULL,
        description JSON,
        FOREIGN KEY (parent_id)
                REFERENCES categories (id));
```

id [PK] integer	title character varying (50)	parent_id integer
1	Electronics	[null]
2	Televisions	1
3	Portables	1
4	Plasma	2
5	LCD	2
6	MP3	3
7	CD	3
8	Laptops	3
9	Osio	6



Querying the hierarchy

- Finding the root node

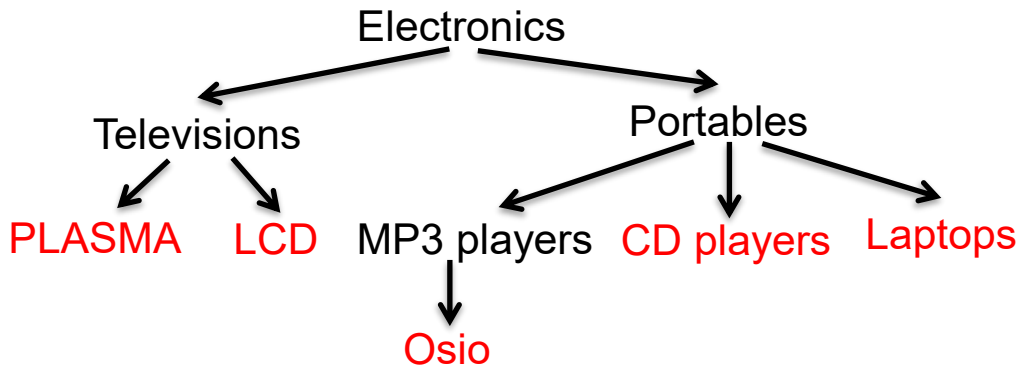



```
-- Finding the root node
SELECT *
FROM categories c
WHERE parent_id IS NULL;
```

id [PK] integer	title character varying (50)	parent_id integer
1	Electronics	[null]

Querying the hierarchy

- Finding the leaf nodes (i.e., all devices)

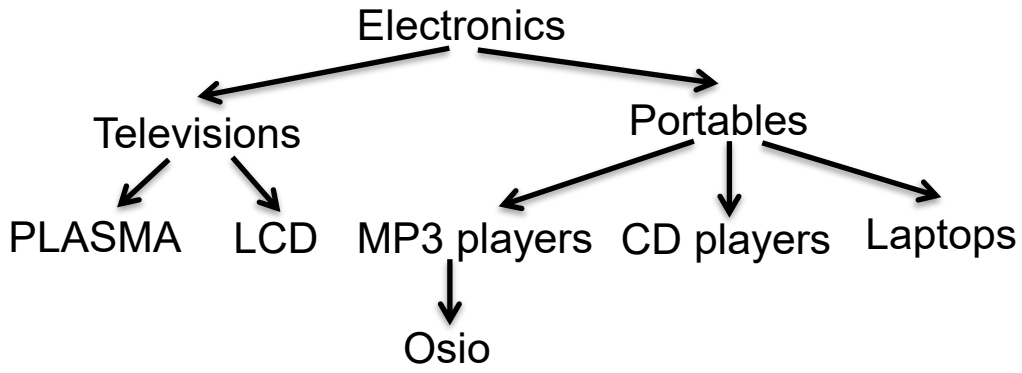


id [PK] integer	title character varying (50)
4	Plasma
5	LCD
7	CD
8	Laptops
9	Osio

```
SELECT c.id, c.title
FROM categories c
WHERE c.id NOT IN (
    SELECT parent_id FROM categories c2
    WHERE parent_id IS NOT NULL);
```

Querying the hierarchy

- Find all parent-child siblings




```
SELECT c1.parent_id AS shared_parent_id, c2.title AS parent_name,
       STRING_AGG(c1.title, ', ' ORDER BY c1.id) AS sibling_list
FROM categories c1
INNER JOIN categories c2
   ON c1.parent_id = c2.id
GROUP BY c1.parent_id, c2.title
HAVING COUNT(c1.id) > 1
ORDER BY c1.parent_id;
```

shared_parent_id integer	parent_name character varying (50)	sibling_list text
1	Electronics	Televisions, Portables
2	Televisions	Plasma, LCD
3	Portables	MP3, CD, Laptops

Querying the hierarchy – Full retrieval

- To retrieve the full hierarchy we need to traverse all hierarchical parent-child relationships
  - Standard SQL cannot do this in a single flat query without knowing the maximum depth, so specialized techniques are required

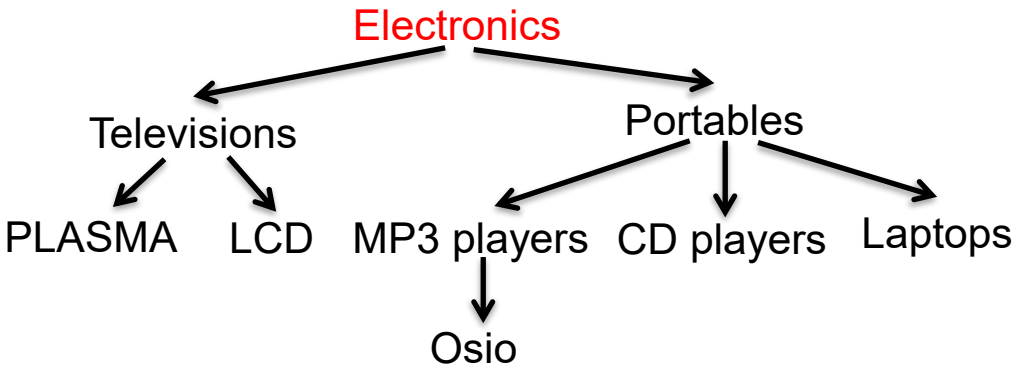
Closure table

- Unlike adjacency model (which only stores a parent_id), a closure table is a separate bridge table that typically contains three columns
  - Ancestor
    - The ID of a node higher in the tree.
  - Descendant
    - The ID of a node lower in the tree.
  - Depth (optional)
    - The number of levels between the ancestor and descendant (e.g., 0 for self-reference, 1 for direct child)

Electronics	Electronics	0

Closure table

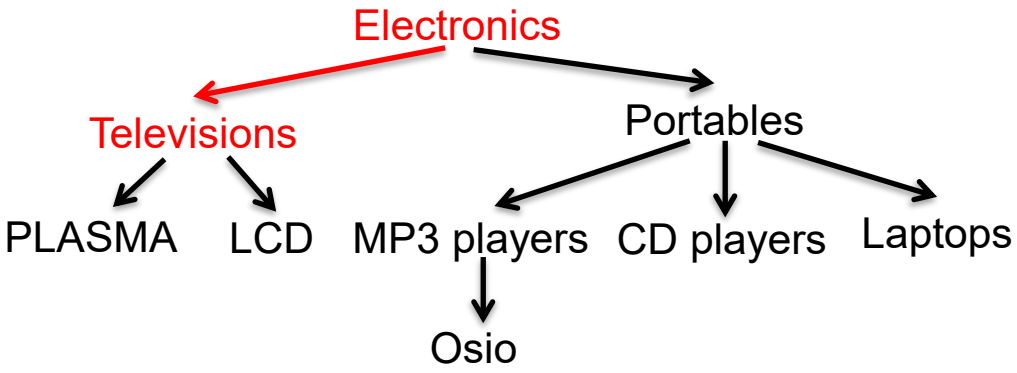
- For the same data set



Closure table

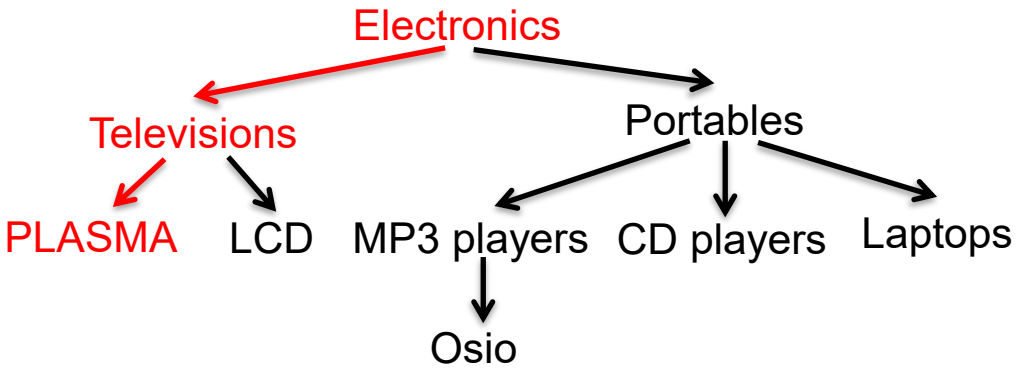
Electronics	Electronics	0
Electronics	Televisions	1
Televisions	Televisions	0

- For the same data set



Closure table

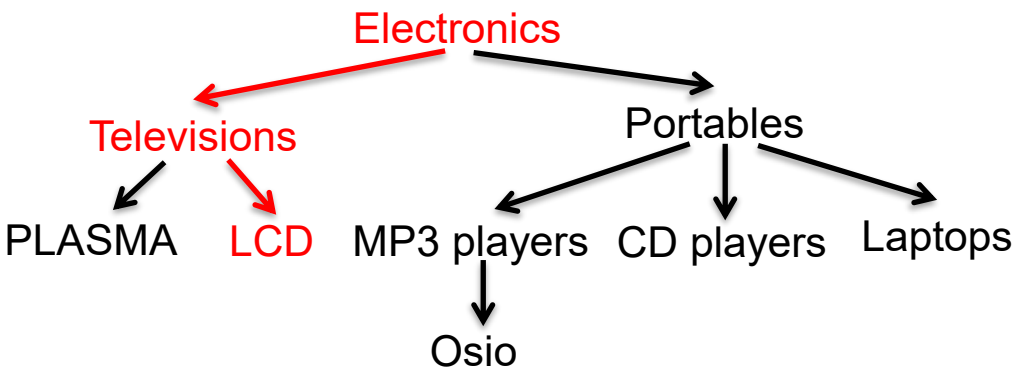
- For the same data set



Electronics	Electronics	0
Electronics	Televisions	1
Televisions	Televisions	0
Televisions	Plasma	1
Electronics	Plasma	2
Plasma	Plasma	0

Closure table

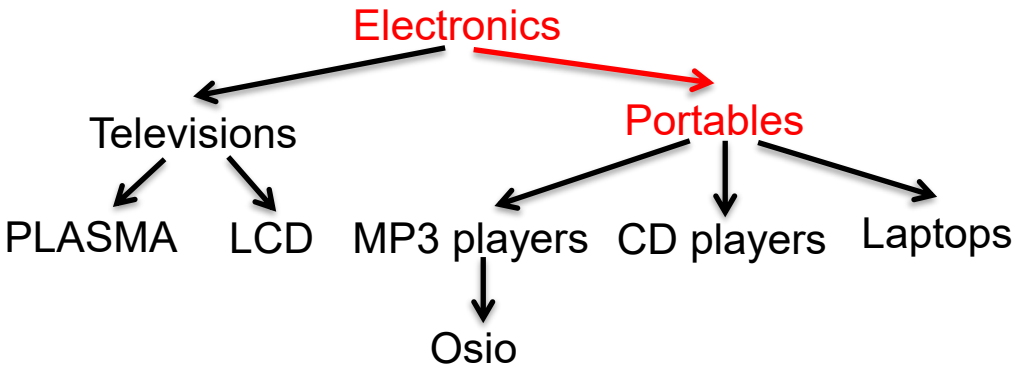
- For the same data set



Electronics	Electronics	0
Electronics	Televisions	1
Televisions	Televisions	0
Televisions	Plasma	1
Electronics	Plasma	2
Plasma	Plasma	0
Televisions	LCD	1
Electronics	LCD	2
LCD	LCD	0

Closure table

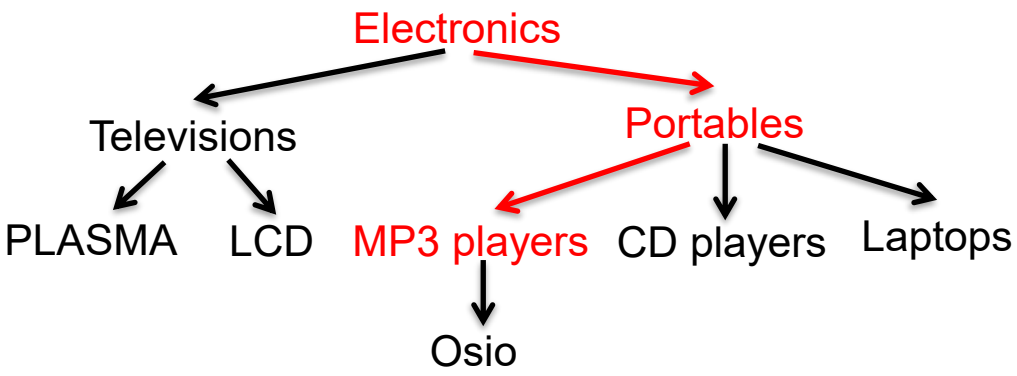
- For the same data set



Electronics	Electronics	0
Electronics	Televisions	1
Televisions	Televisions	0
Televisions	Plasma	1
Electronics	Plasma	2
Plasma	Plasma	0
Televisions	LCD	1
Electronics	LCD	2
LCD	LCD	0
Electronics	Portables	1
Portables	Portables	0

Closure table

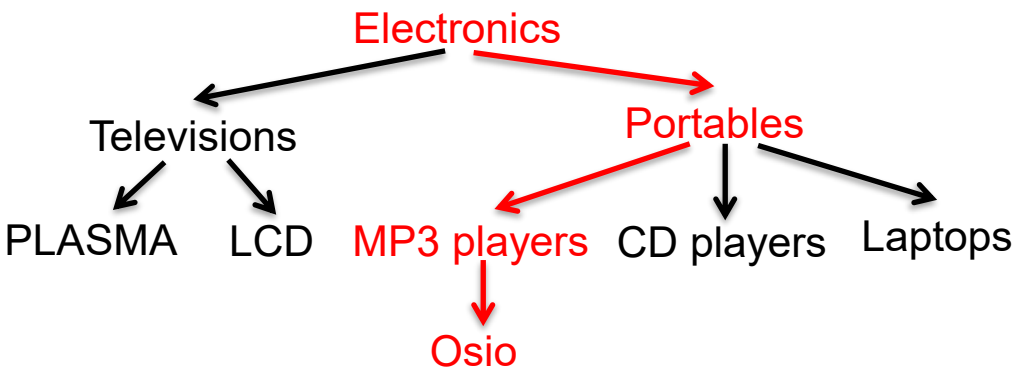
- For the same data set



Electronics	Electronics	0
Electronics	Televisions	1
Televisions	Televisions	0
Televisions	Plasma	1
Electronics	Plasma	2
Plasma	Plasma	0
Televisions	LCD	1
Electronics	LCD	2
LCD	LCD	0
Electronics	Portables	1
Portables	Portables	0
Portables	MP3	1
Electronics	MP3	2
MP3	MP3	0

Closure table

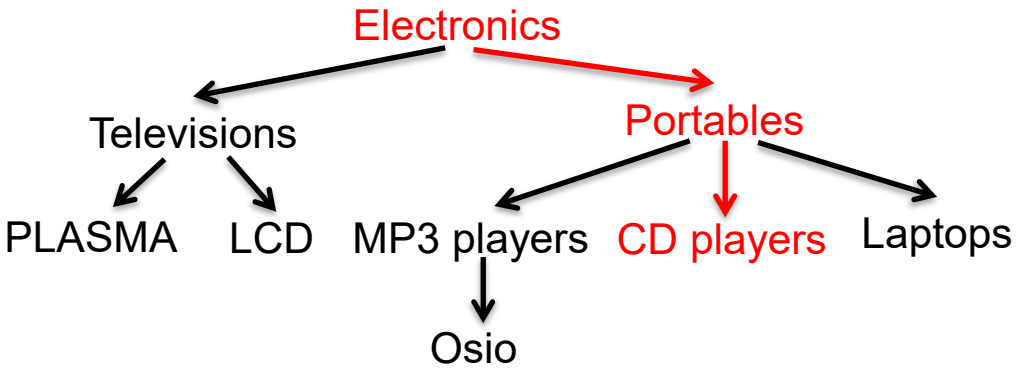
- For the same data set



Electronics	Electronics	0
Electronics	Televisions	1
Televisions	Televisions	0
Televisions	Plasma	1
Electronics	Plasma	2
Plasma	Plasma	0
Televisions	LCD	1
Electronics	LCD	2
LCD	LCD	0
Electronics	Portables	1
Portables	Portables	0
Portables	MP3	1
Electronics	MP3	2
MP3	MP3	0
MP3	Osio	1
Portables	Osio	2
Electronics	Osio	3
Osio	Osio	0

Closure table

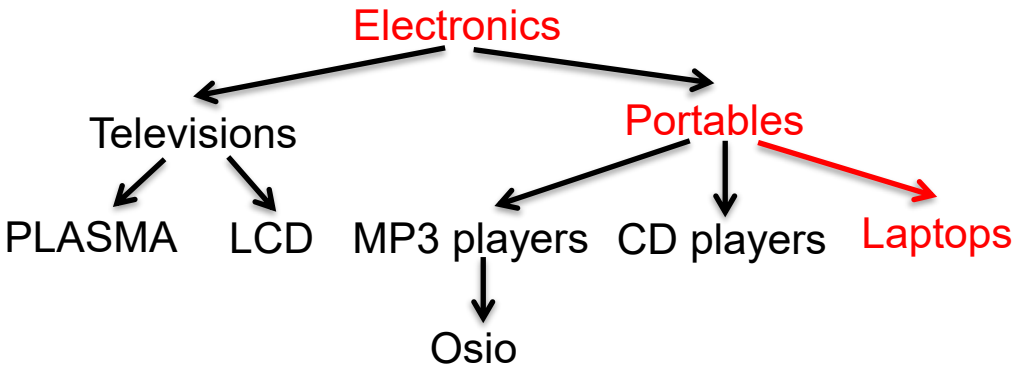
- For the same data set



Electronics	Electronics	0
Electronics	Televisions	1
Televisions	Televisions	0
Televisions	Plasma	1
Electronics	Plasma	2
Plasma	Plasma	0
Televisions	LCD	1
Electronics	LCD	2
LCD	LCD	0
Electronics	Portables	1
Portables	Portables	0
Portables	MP3	1
Electronics	MP3	2
MP3	MP3	0
MP3	Osio	1
Portables	Osio	2
Electronics	Osio	3
Osio	Osio	0
Portables	CD	1
Electronics	CD	2
CD	CD	0

Closure table

- For the same data set



Electronics	Electronics	0
Electronics	Televisions	1
Televisions	Televisions	0
Televisions	Plasma	1
Electronics	Plasma	2
Plasma	Plasma	0
Televisions	LCD	1
Electronics	LCD	2
LCD	LCD	0
Electronics	Portables	1
Portables	Portables	0
Portables	MP3	1
Electronics	MP3	2
MP3	MP3	0
MP3	Osio	1
Portables	Osio	2
Electronics	Osio	3
Osio	Osio	0
Portables	CD	1
Electronics	CD	2
CD	CD	0
Portables	Laptops	1
Electronics	Laptops	2
Laptops	Laptops	0

Implementing the closure

- Two table solution

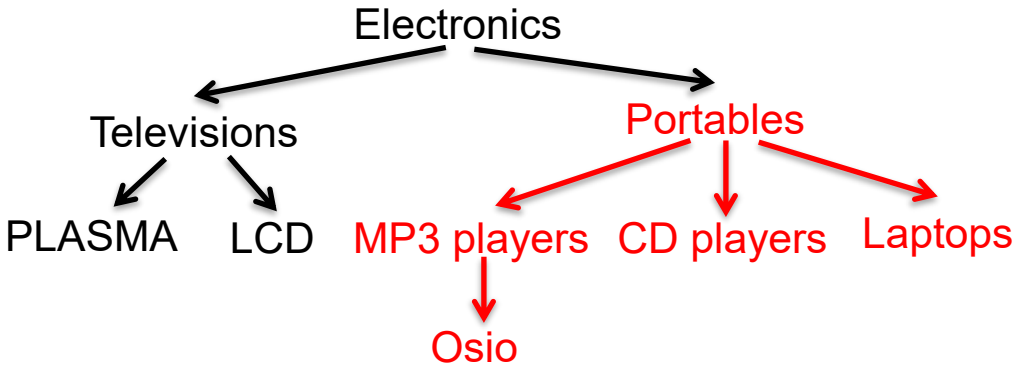
```
CREATE TABLE categories (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL);

CREATE TABLE categories_closure (
    ancestor_id INT REFERENCES categories(id),
    descendant_id INT REFERENCES categories(id),
    depth INT NOT NULL,
    PRIMARY KEY (ancestor_id, descendant_id));
```

Electronics	1
Televisions	2
Portables	3
Plasma	4
LCD	5
MP3	6
CD	7
Laptops	8
Osio	9

Electronics	Electronics	0
Electronics	Televisions	1
Televisions	Televisions	0
Televisions	Plasma	1
Electronics	Plasma	2
Plasma	Plasma	0
Televisions	LCD	1
Electronics	LCD	2
LCD	LCD	0
Electronics	Portables	1
Portables	Portables	0
Portables	MP3	1
Electronics	MP3	2
MP3	MP3	0
MP3	Osio	1
Portables	Osio	2
Electronics	Osio	3
Osio	Osio	0
Portables	CD	1
Electronics	CD	2
CD	CD	0
Portables	Laptops	1
Electronics	Laptops	2
Laptops	Laptops	0

Querying using the closure table

- Finding all children (depth > 0) of node with id=3



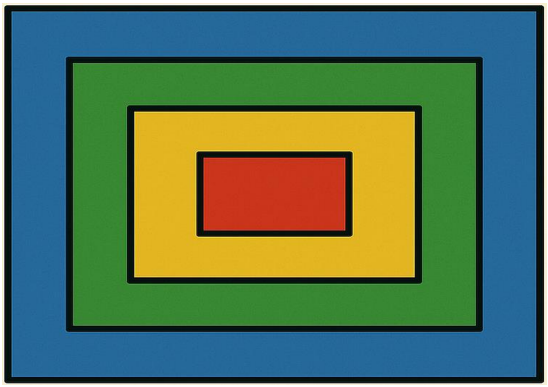
id [PK] integer	title character varying (50)	parent_id integer
6	MP3	3
7	CD	3
8	Laptops	3
9	Osio	6

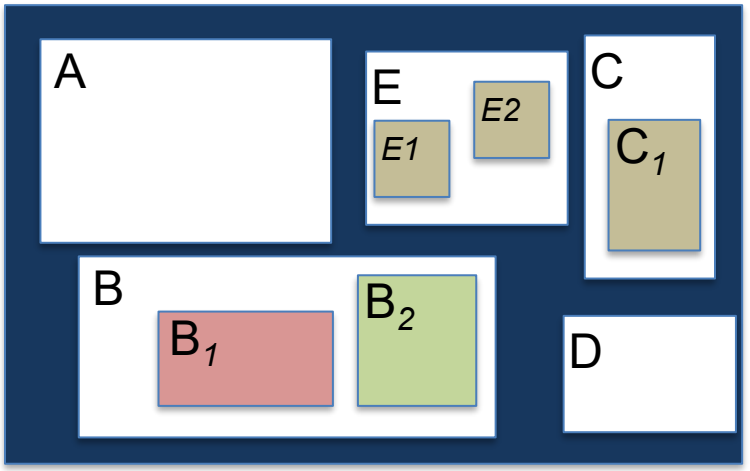
```
-- Finding all children (depth > 0) of node with id=3
SELECT n.*
FROM categories_closure nc
JOIN categories n ON n.id = nc.descendant_id
WHERE nc.ancestor_id = 3
  AND nc.depth > 0 ;
```

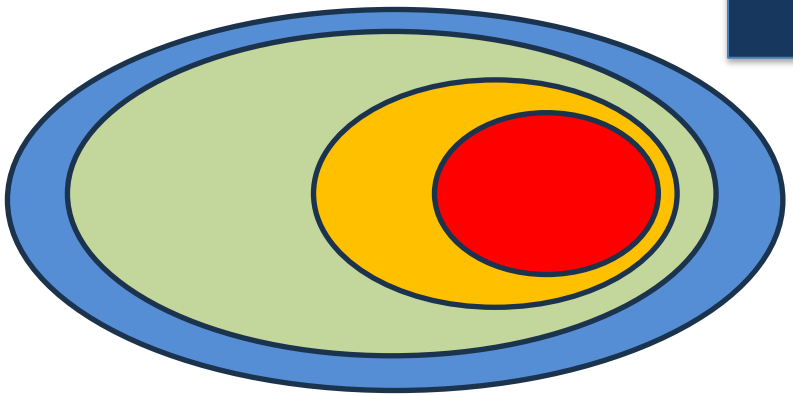
Nested sets

Nested sets model

- A nested set collection or nested set family is a collection of sets that consists of *chains of subsets* forming a *hierarchical* structure

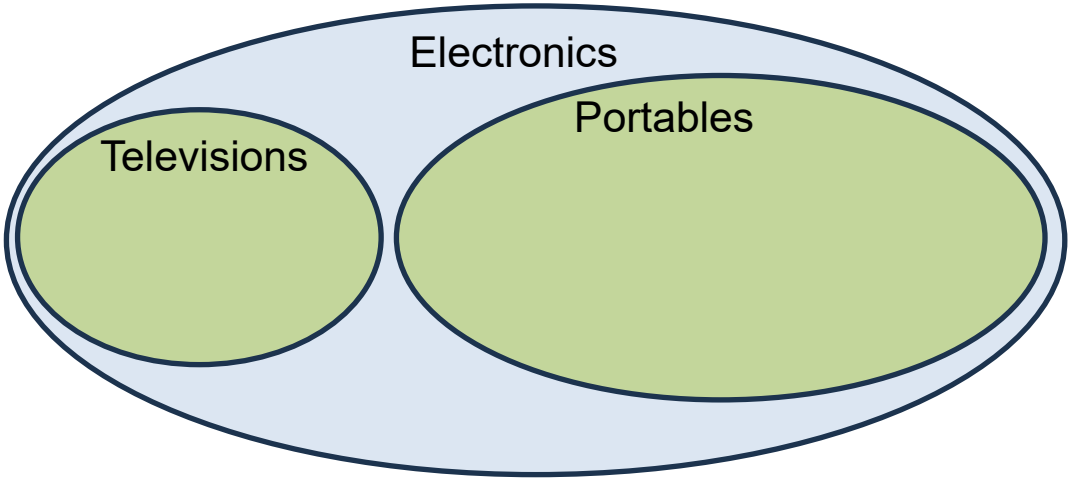


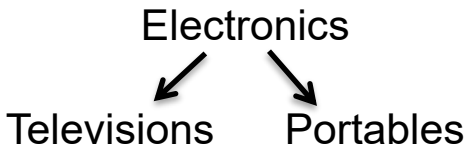




Nested sets model (cont.)

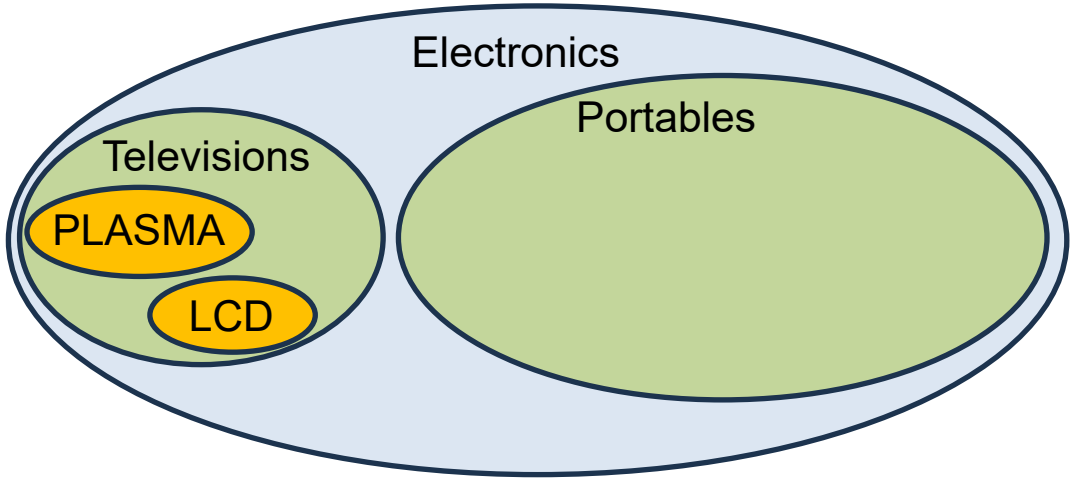
- The property featured is the mathematical set containment ($A \subseteq B$ if every element in A is in B)
  - Electronics include Televisions and Portables

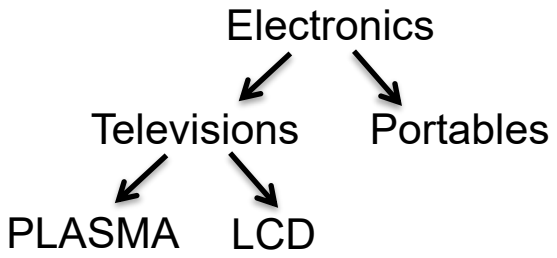




Nested sets model (cont.)

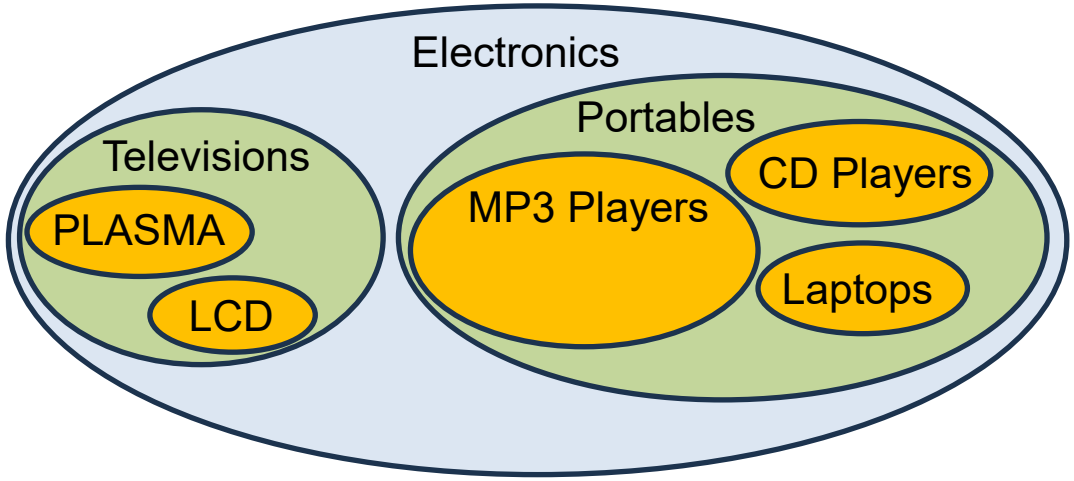
- Set containment may applies to any sub-category
  - Electronics include Televisions and Portables
    - Televisions include PLASMA and LCD

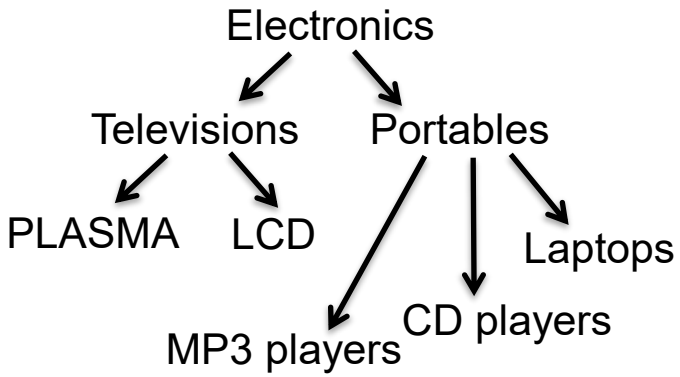




Nested sets model (cont.)

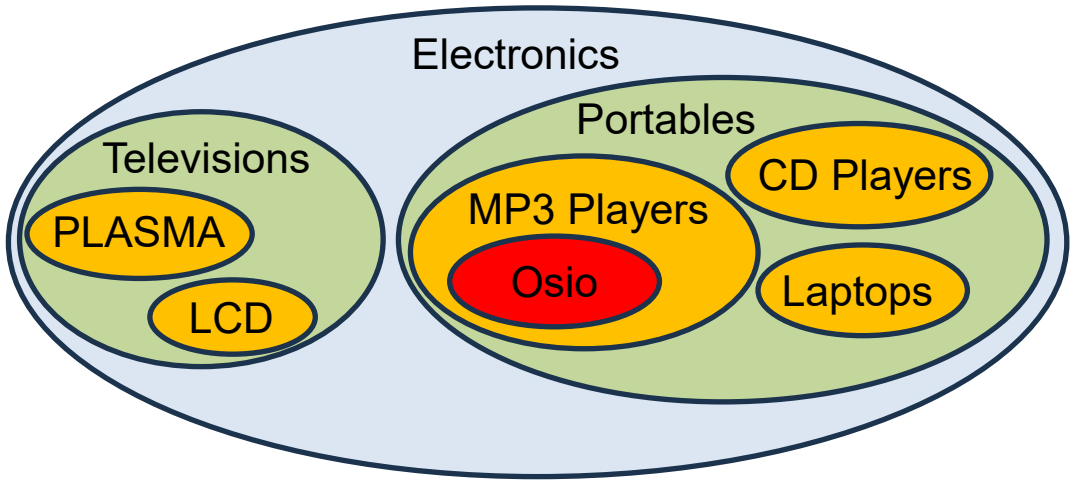
- Set containment may applies to any sub-category
  - Electronics include Televisions and Portables
    - Televisions include PLASMA and LCD
    - Portables include MP3 players, CD players and Laptops

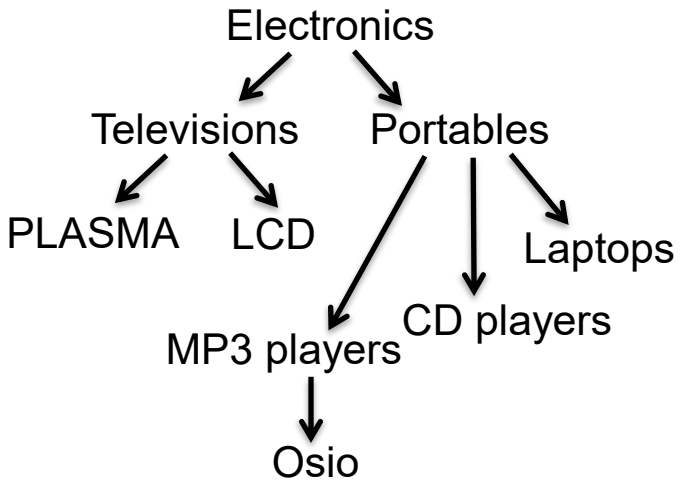




Nested sets model (cont.)

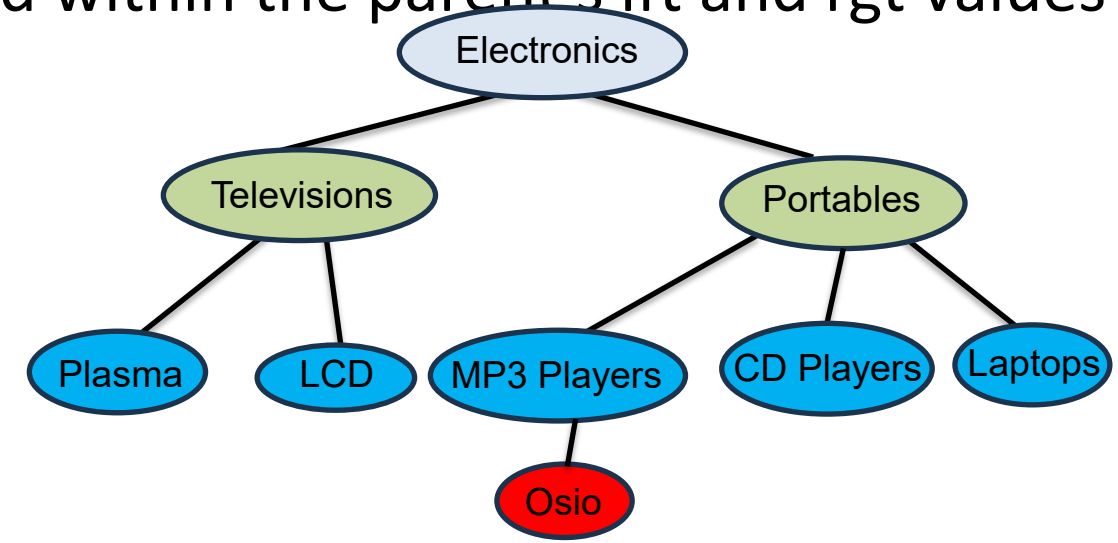
- Set containment may applies to any sub-category
  - Electronics include Televisions and Portables
    - Televisions include PLASMA and LCD
    - Portables include MP3 players, CD players and Laptops
      - MP3 players include Osio





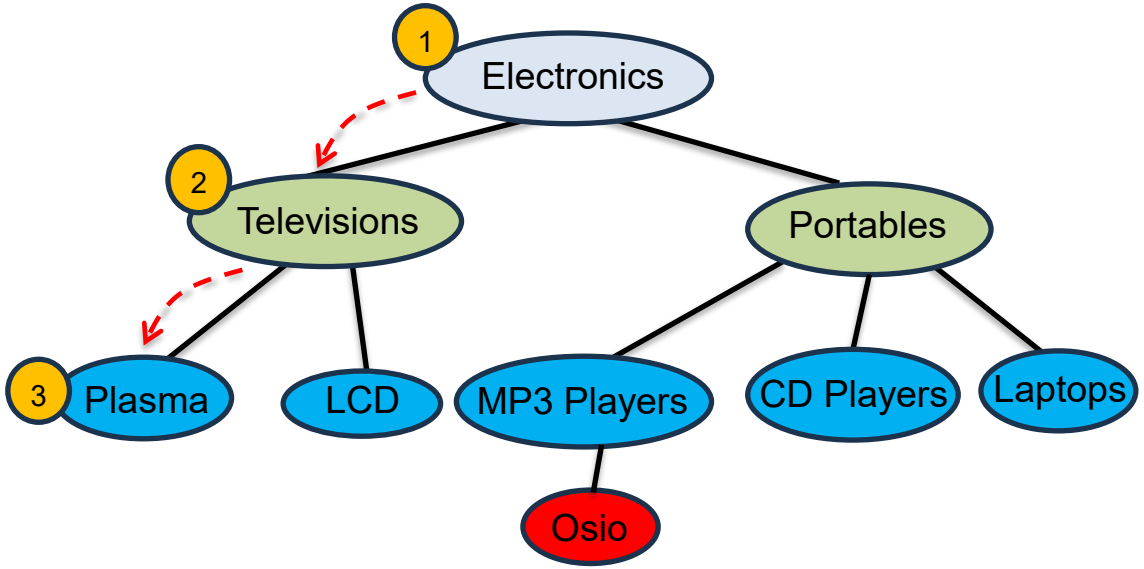
Relational representation of nested sets

- One relation with one tuple per node
- Assignment of two integer values—left (lft) and right (rgt)—to each node
  - Numbering occurs according to a tree *traversal*, which visits each node twice, assigning numbers in the order of visiting, and at both visits
- A node's descendants are found where lft and rgt values are entirely contained within the parent's lft and rgt values



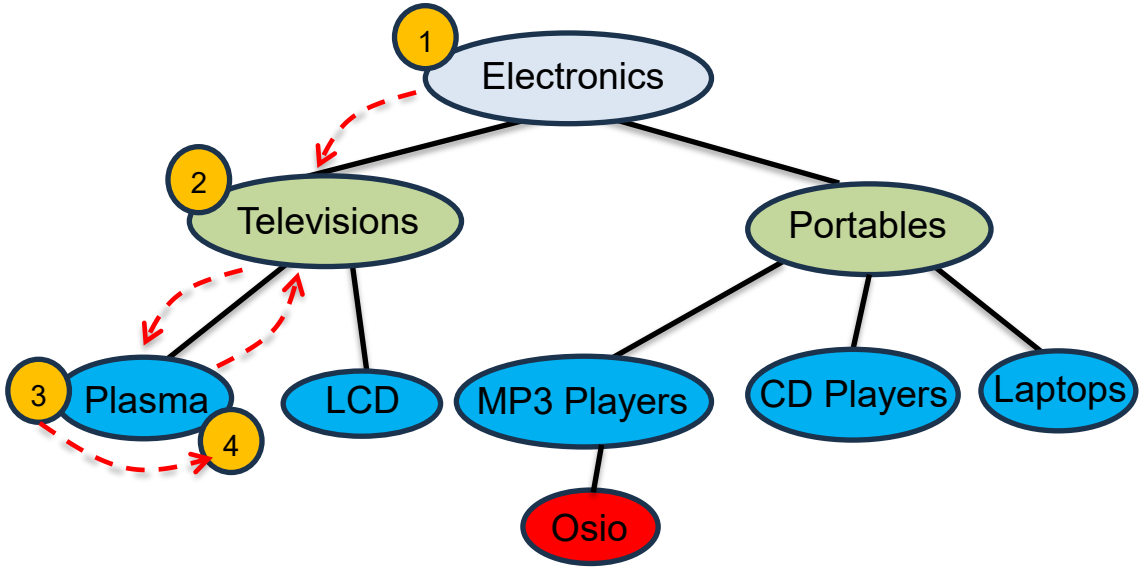
Traversing nested sets

- We start at the *root* node of the tree and go to the leftmost leaf node in order of visiting nodes



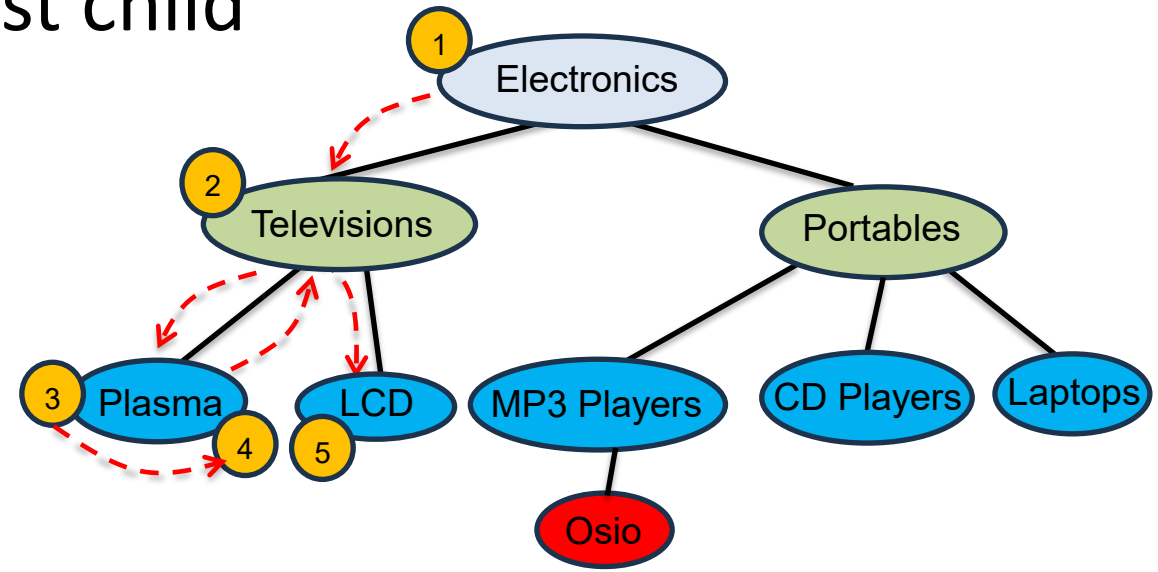
Traversing nested sets

- We start at the root node of the tree and go to the leftmost leaf node in order of visiting nodes
  - This done until we reach lead node
- Once we reach it, we go back until we reach a node that has multiple children



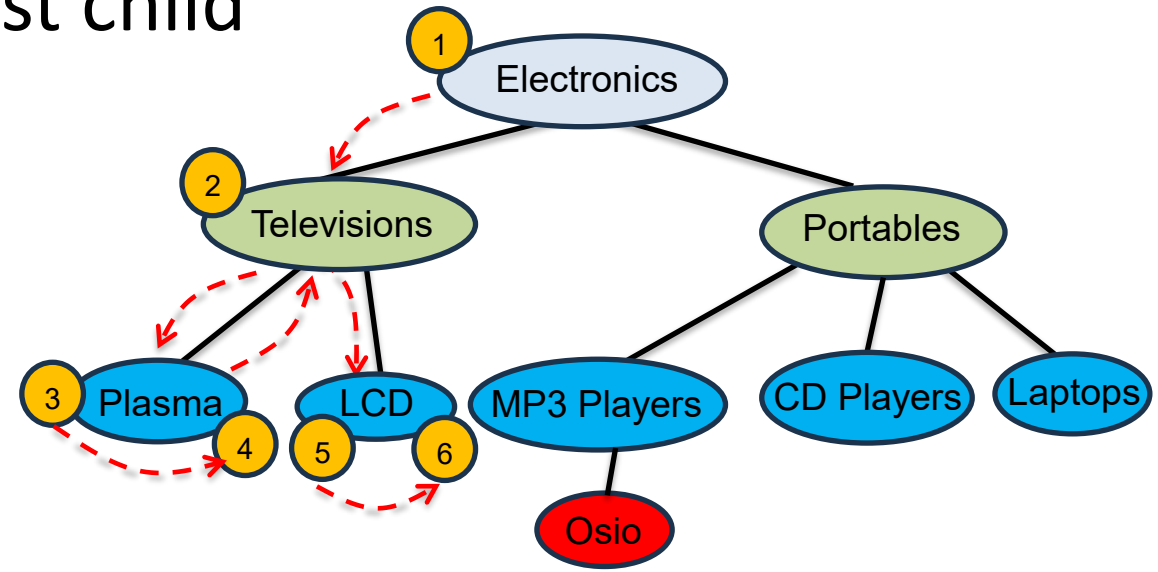
Traversing nested sets

- We start at the root node of the tree and go to the leftmost leaf node in order of visiting nodes
  - This done until we reach lead node
- Once we reach it, we go back until we reach a node that has multiple children to visit the next unvisited leftmost child



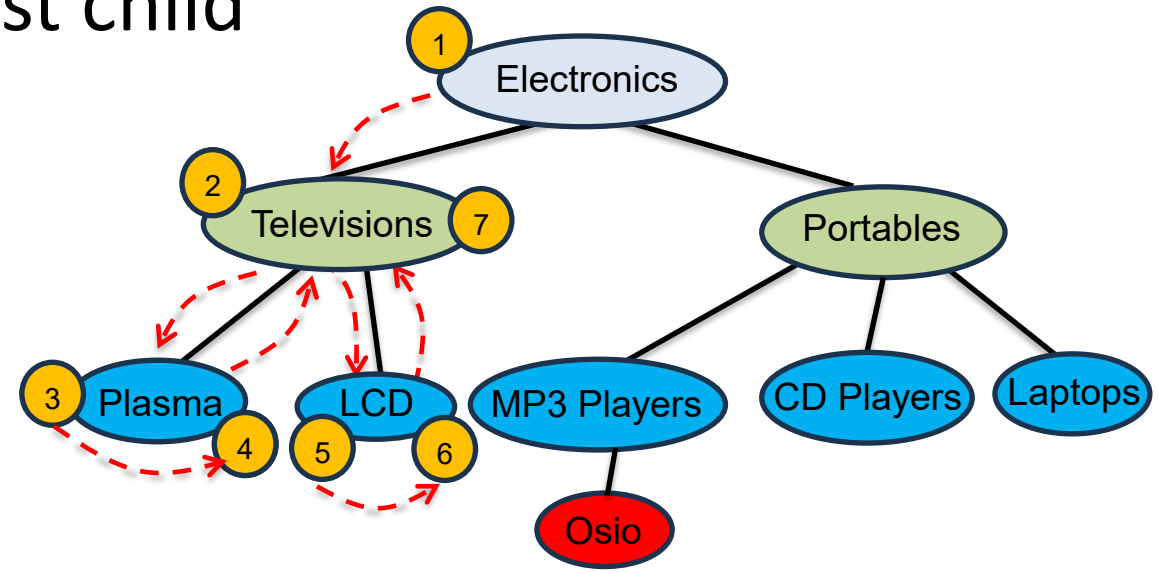
Traversing nested sets

- We start at the root node of the tree and go to the leftmost leaf node in order of visiting nodes
  - This done until we reach lead node
- Once we reach it, we go back until we reach a node that has multiple children to visit the next unvisited leftmost child



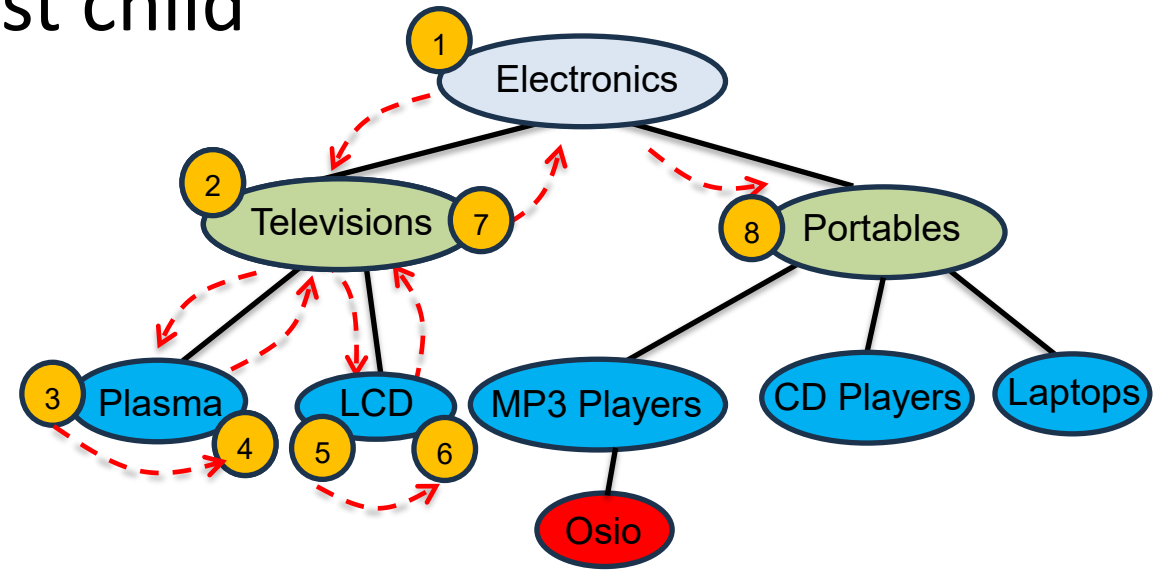
Traversing nested sets

- We start at the root node of the tree and go to the leftmost leaf node in order of visiting nodes
  - This done until we reach lead node
- Once we reach it, we go back until we reach a node that has multiple children to visit the next unvisited leftmost child



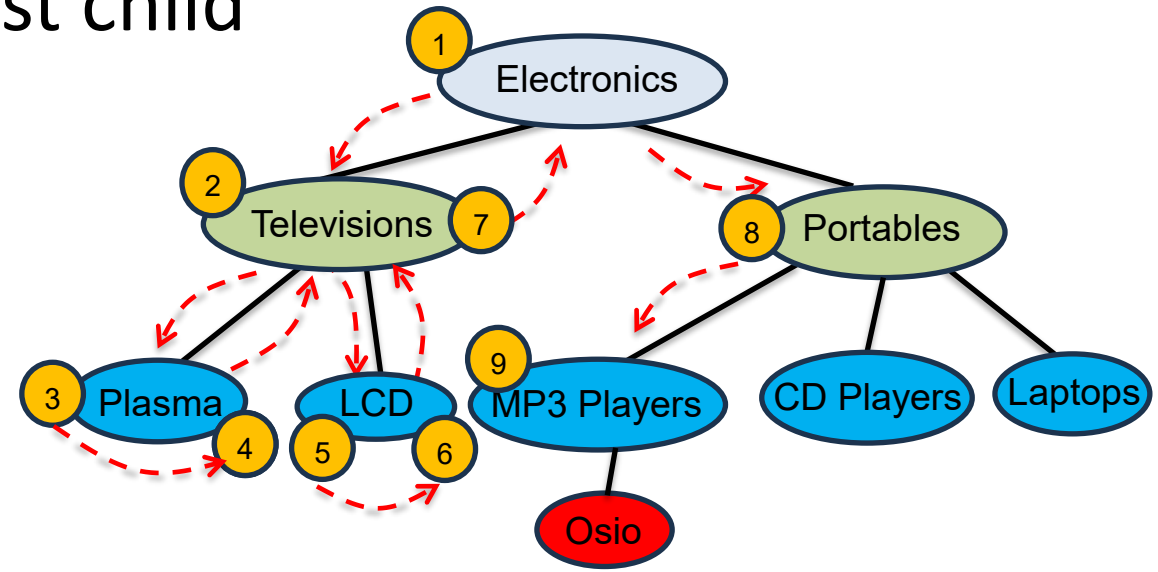
Traversing nested sets

- We start at the root node of the tree and go to the leftmost leaf node in order of visiting nodes
  - This done until we reach lead node
- Once we reach it, we go back until we reach a node that has multiple children to visit the next unvisited leftmost child



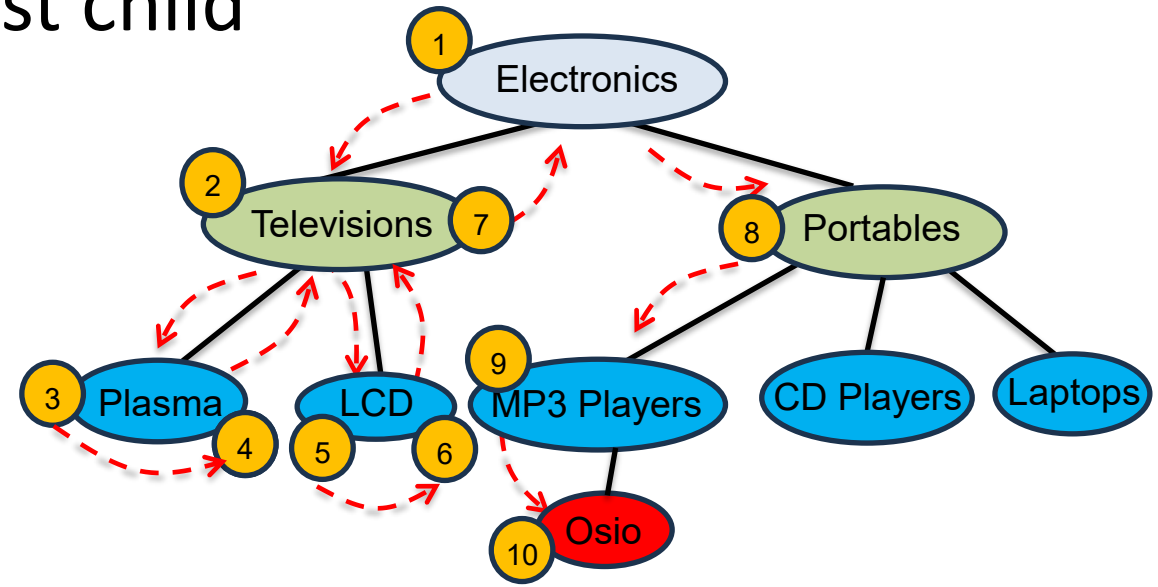
Traversing nested sets

- We start at the root node of the tree and go to the leftmost leaf node in order of visiting nodes
  - This done until we reach lead node
- Once we reach it, we go back until we reach a node that has multiple children to visit the next unvisited leftmost child



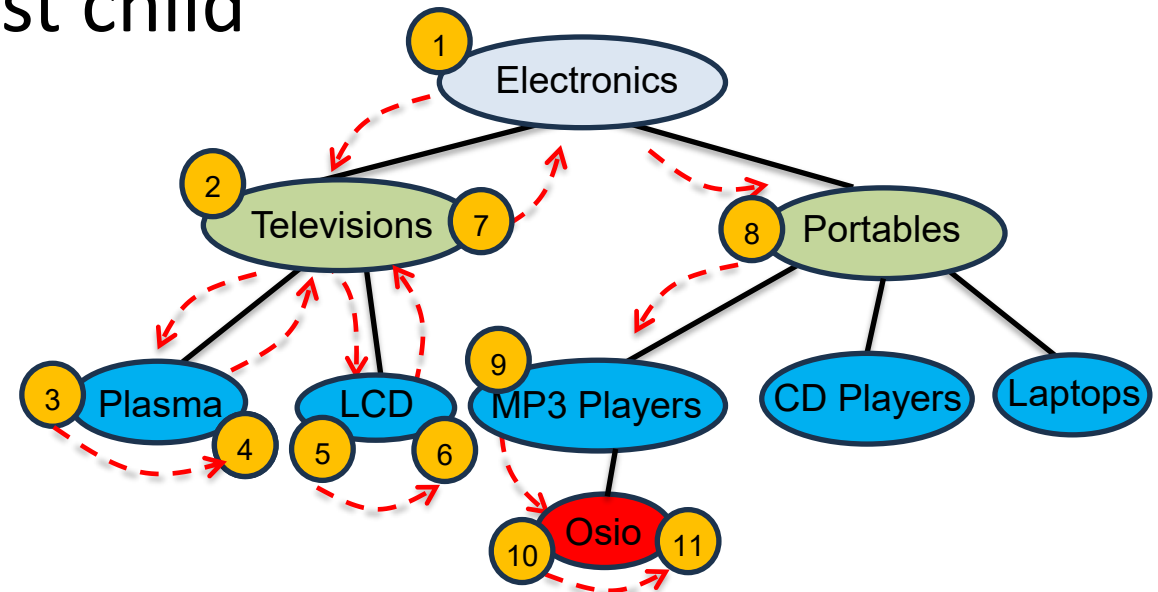
Traversing nested sets

- We start at the root node of the tree and go to the leftmost leaf node in order of visiting nodes
  - This done until we reach lead node
- Once we reach it, we go back until we reach a node that has multiple children to visit the next unvisited leftmost child



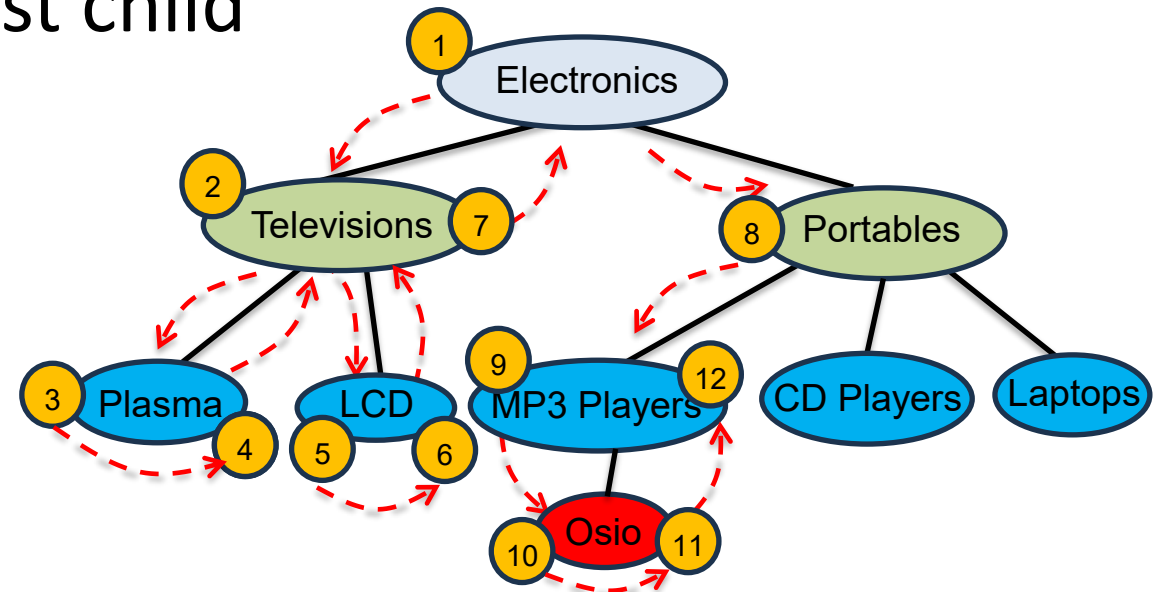
Traversing nested sets

- We start at the root node of the tree and go to the leftmost leaf node in order of visiting nodes
  - This done until we reach lead node
- Once we reach it, we go back until we reach a node that has multiple children to visit the next unvisited leftmost child



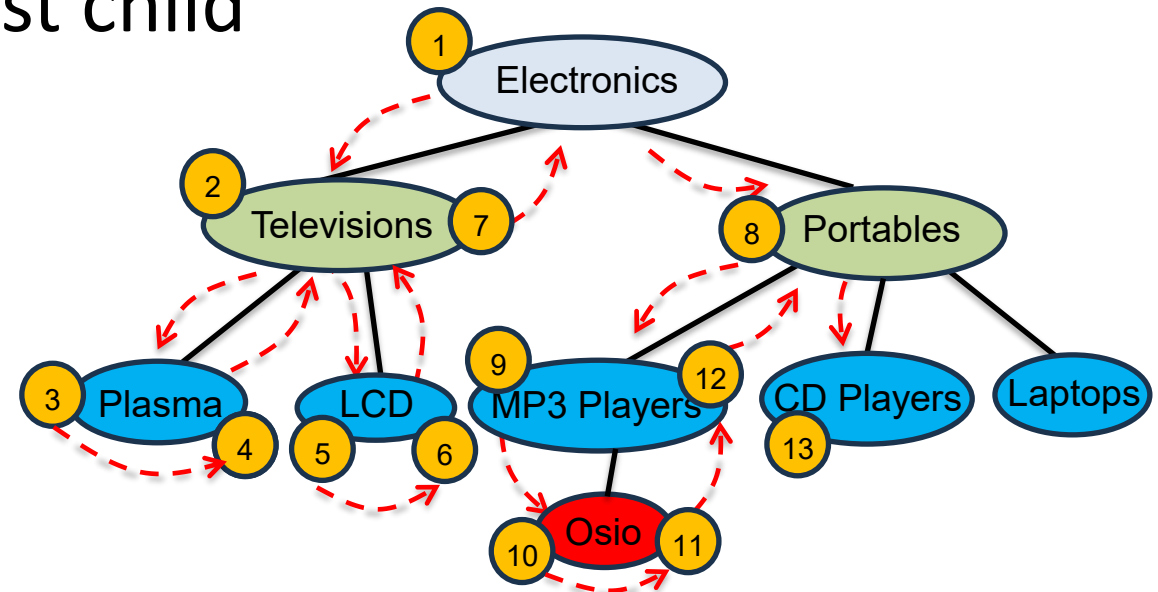
Traversing nested sets

- We start at the root node of the tree and go to the leftmost leaf node in order of visiting nodes
  - This done until we reach lead node
- Once we reach it, we go back until we reach a node that has multiple children to visit the next unvisited leftmost child



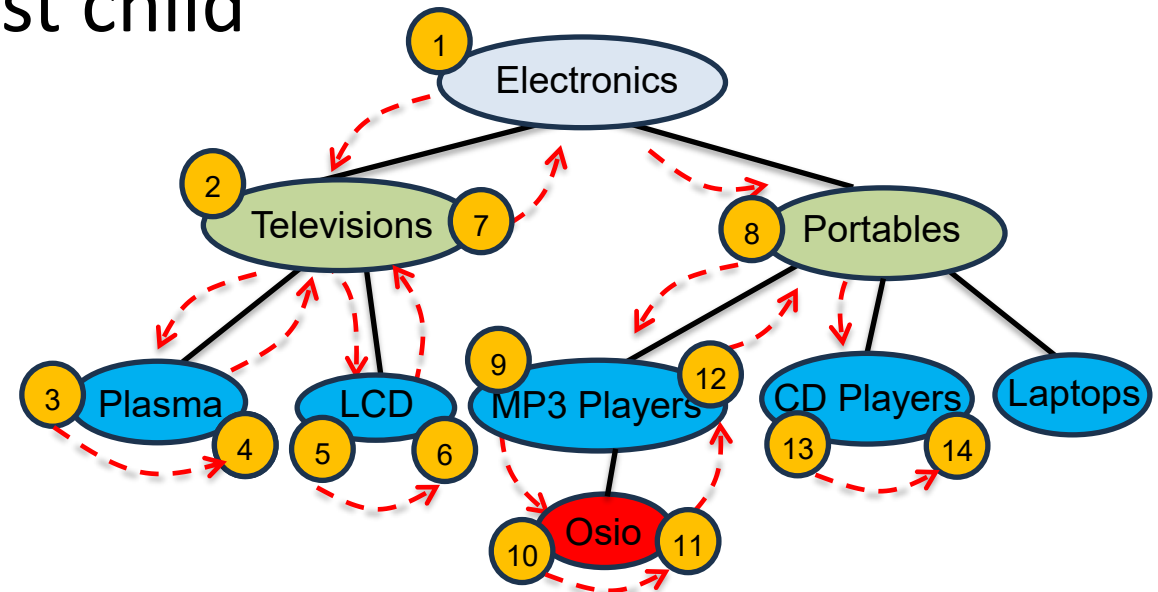
Traversing nested sets

- We start at the root node of the tree and go to the leftmost leaf node in order of visiting nodes
  - This done until we reach lead node
- Once we reach it, we go back until we reach a node that has multiple children to visit the next unvisited leftmost child



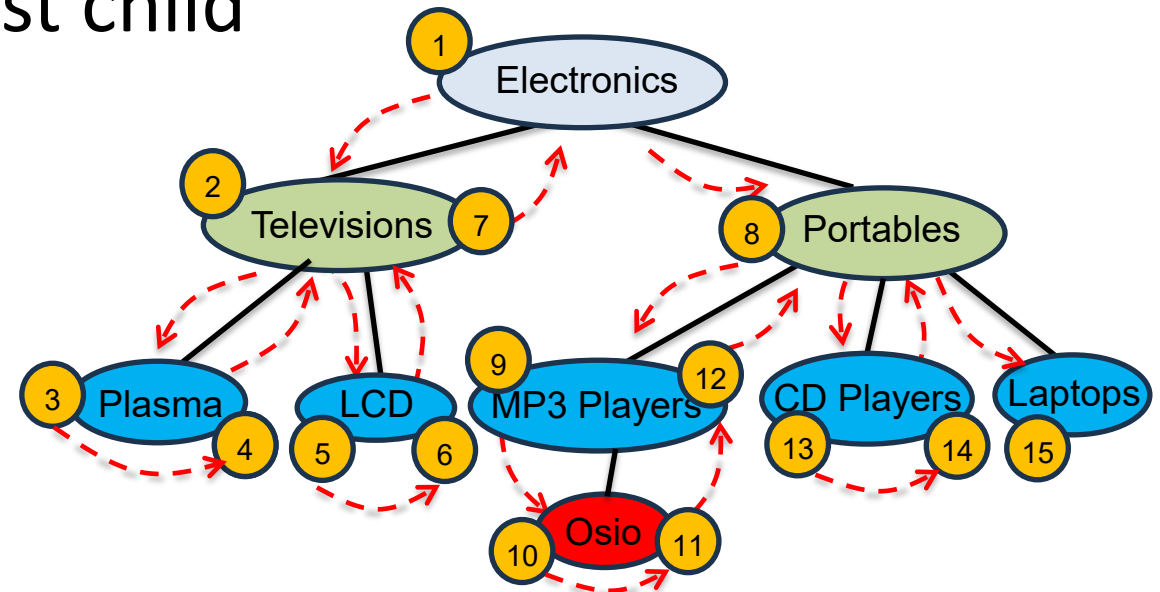
Traversing nested sets

- We start at the root node of the tree and go to the leftmost leaf node in order of visiting nodes
  - This done until we reach lead node
- Once we reach it, we go back until we reach a node that has multiple children to visit the next unvisited leftmost child



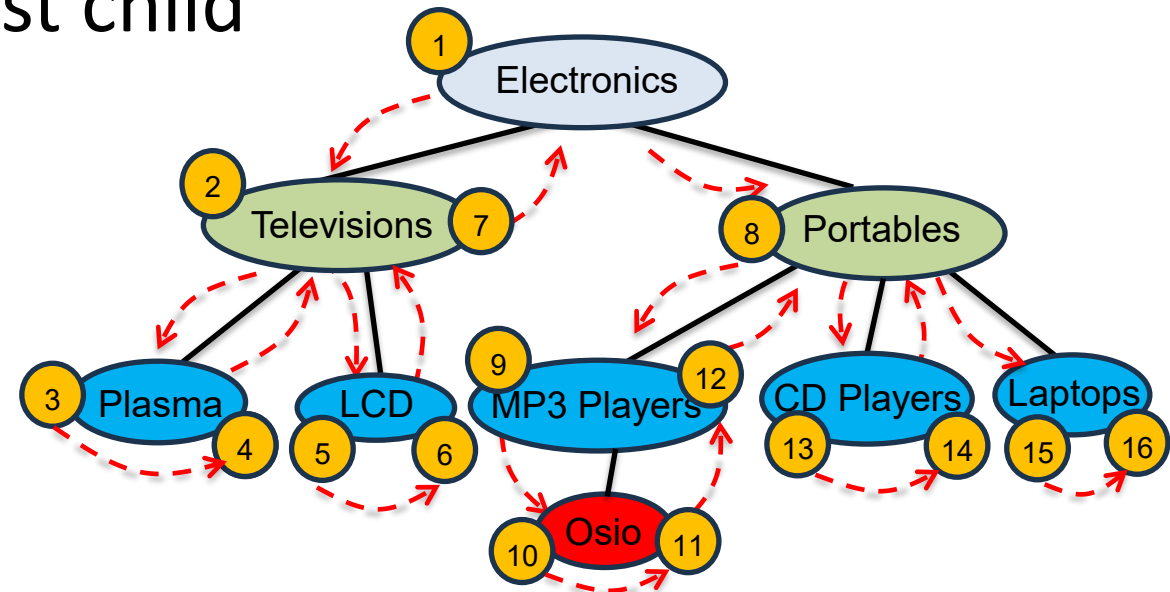
Traversing nested sets

- We start at the root node of the tree and go to the leftmost leaf node in order of visiting nodes
  - This done until we reach lead node
- Once we reach it, we go back until we reach a node that has multiple children to visit the next unvisited leftmost child



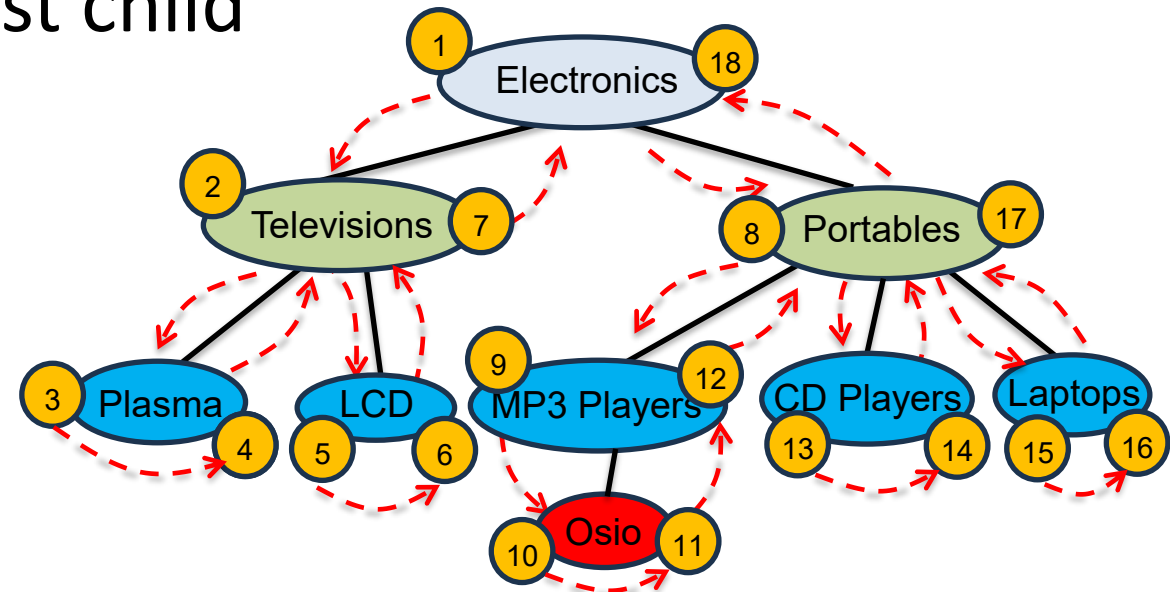
Traversing nested sets

- We start at the root node of the tree and go to the leftmost leaf node in order of visiting nodes
  - This done until we reach lead node
- Once we reach it, we go back until we reach a node that has multiple children to visit the next unvisited leftmost child



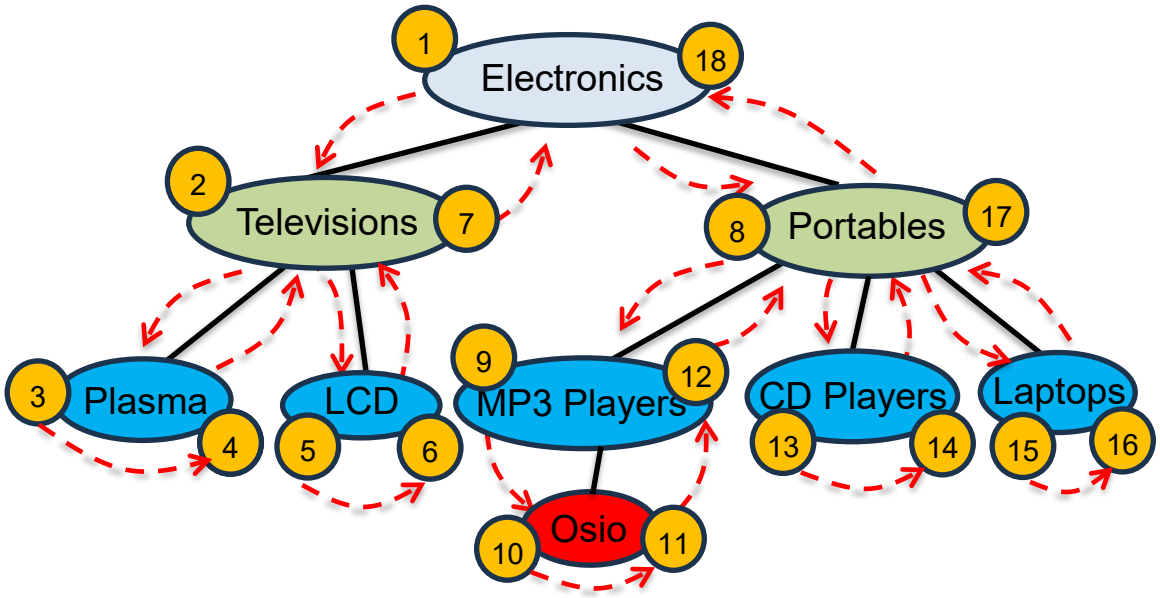
Traversing nested sets

- We start at the root node of the tree and go to the leftmost leaf node in order of visiting nodes
  - This done until we reach lead node
- Once we reach it, we go back until we reach a node that has multiple children to visit the next unvisited leftmost child



Traversing nested sets

- The equivalent table



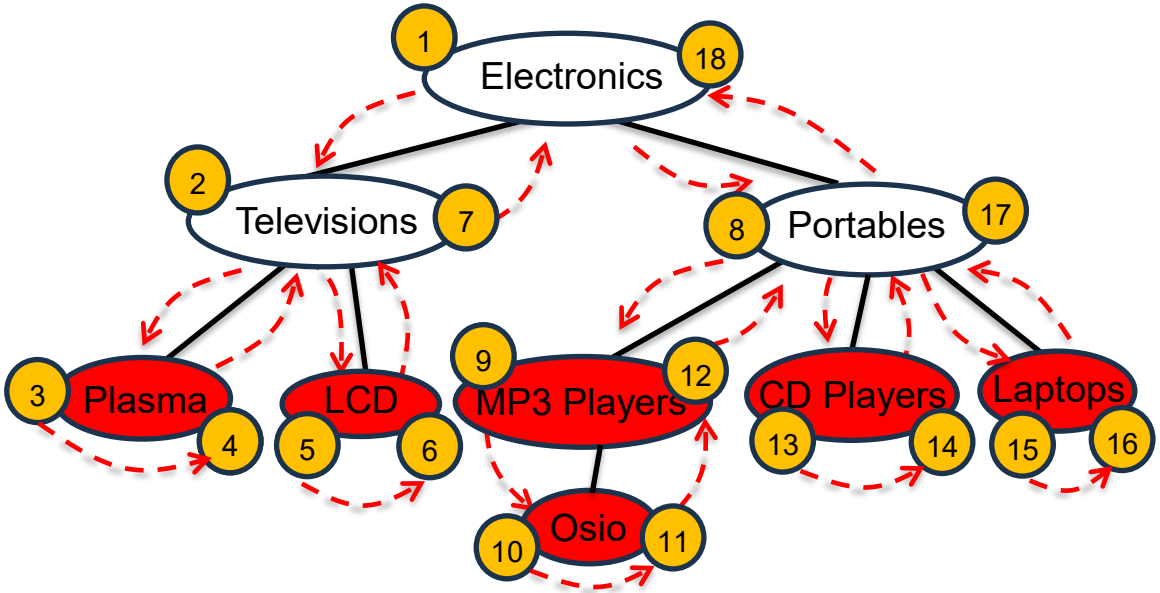
Electronics	1	18
Televisions	2	7
Portables	8	17
Plasma	3	4
LCD	5	6
MP3	9	12
CD	13	14
Laptops	15	16
Osio	10	11

Querying

- Find all Leaf Nodes

```
-- Find all Leaf Nodes
SELECT title
FROM categories
WHERE rgt = lft + 1;
```

title character varying (50)
Plasma
LCD
Osio
CD
Laptops

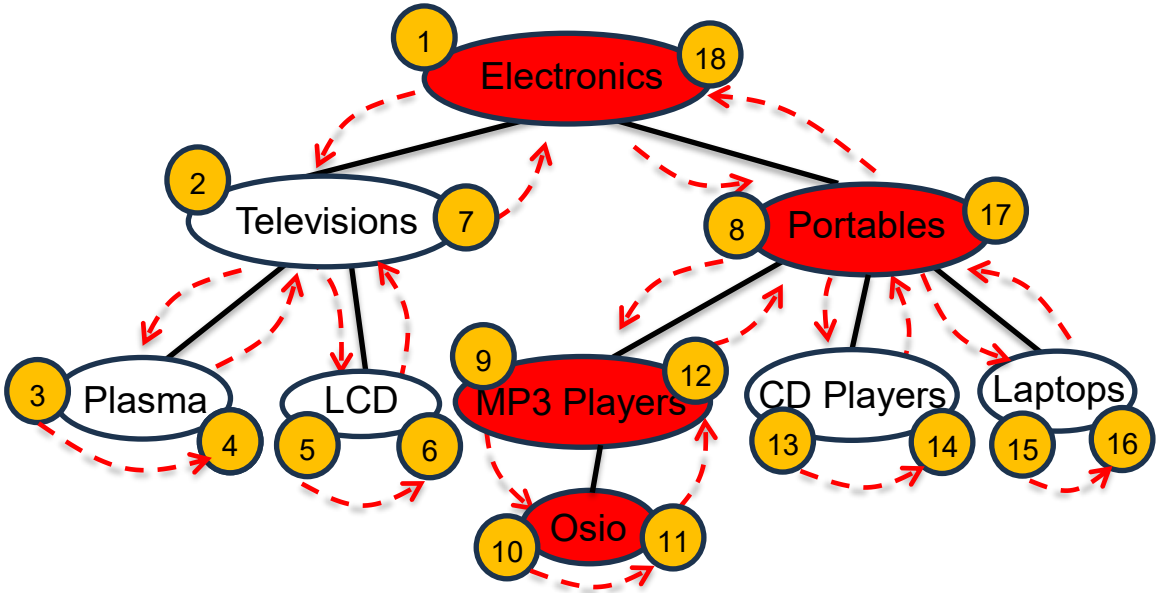


Querying

- Retrieve a Full Path

```
-- Retrieve a Full Path
SELECT parent.title
FROM categories AS node,
     categories AS parent
WHERE node.lft BETWEEN parent.lft AND parent.rgt
  AND node.title = 'Osio'
ORDER BY parent.lft;
```

title character varying (50)
Electronics
Portables
MP3
Osio

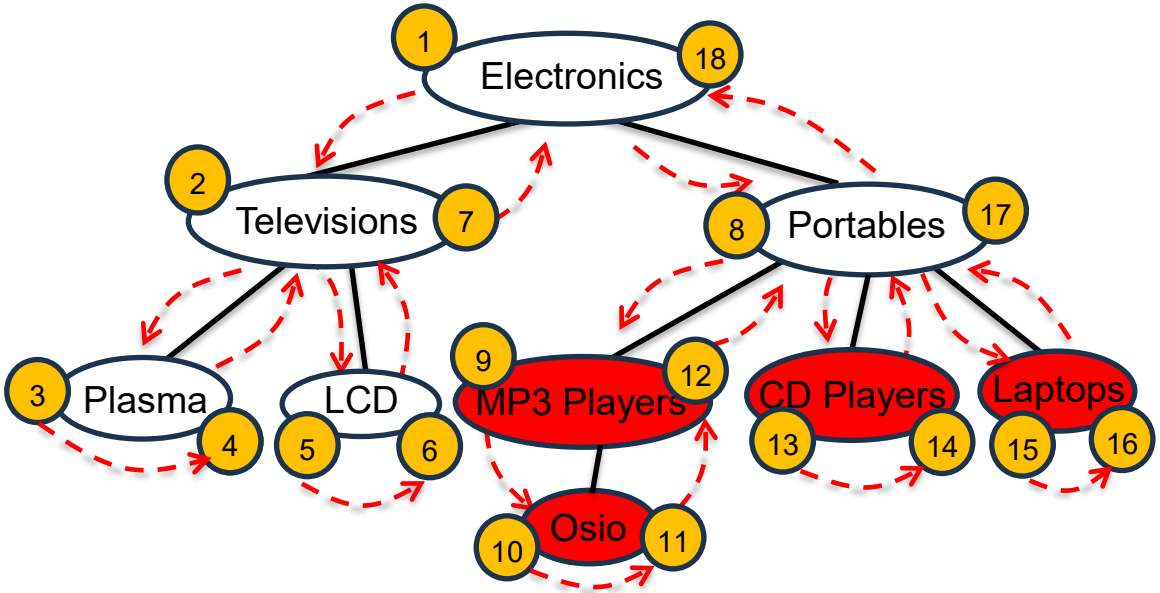


Querying

- Find the total number of Descendants

```
-- Find the total number of Descendants
SELECT title, (rgt - lft - 1) / 2 AS total_descendants
FROM categories
WHERE title = 'Portables';
```

title character varying (50)	total_descendants integer
Portables	4

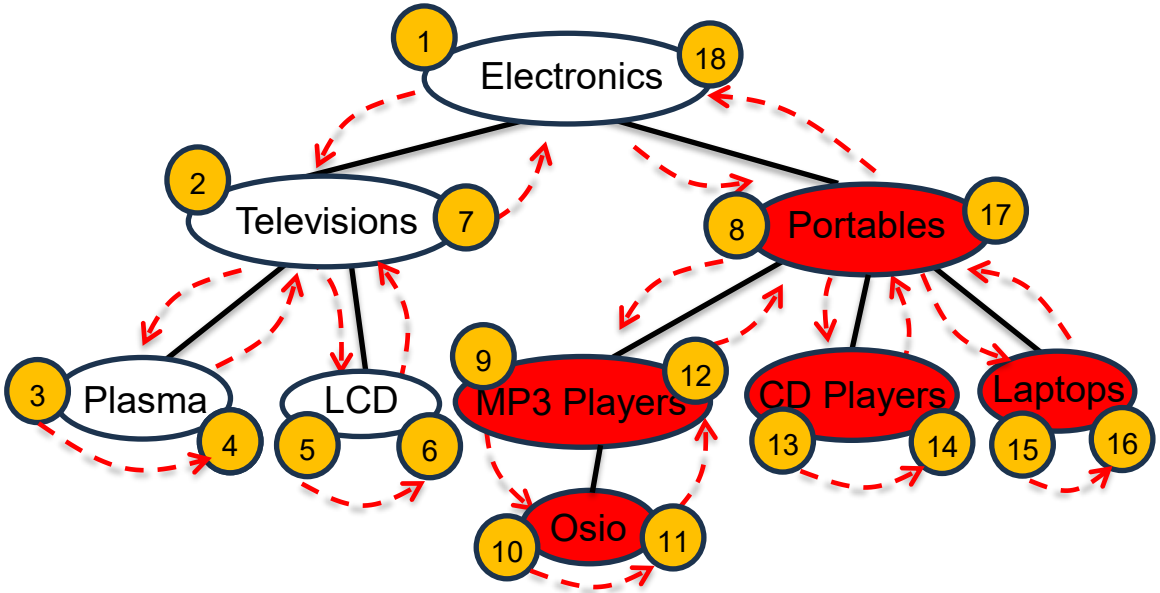


Querying

- Finding a subtree

id [PK] integer	title character varying (50)	parent_id integer	lft integer	rgt integer
3	Portables	1	8	17
6	MP3	3	9	12
7	CD	3	13	14
8	Laptops	3	15	16
9	Osio	6	10	11

```
-- Finding a subtree
SELECT child.*
FROM categories AS parent
JOIN categories AS child ON child.lft BETWEEN parent.lft AND parent.rgt
WHERE parent.title = 'Portables'
ORDER BY id asc;
```



Comparison

- Depending on the criteria the three models may be assessed as follows

Feature	Adjacency	Nested Set	Closure Table
Insert Speed	Very fast	Slow	Medium
Subtree Query	Complex (Slow)	Very fast	Fast
Simplicity	High	Low	Medium
Storage Cost	Very low	Low	High

CTEs and recursion in SQL

Recursive queries – definition

- A recursive query in SQL is a query that calls itself repeatedly until it reaches a termination condition

Recursive CTE for querying

- A CTE is a temporary data set returned by a query, which is then used by another query
  - It's temporary because the result is not stored anywhere; it exists only when the query is run
- Syntax

```
WITH RECURSIVE cte_name AS
(
  cte_query_definition
    UNION ALL
  cte_query_definition
 )
SELECT *
FROM cte_name;
```

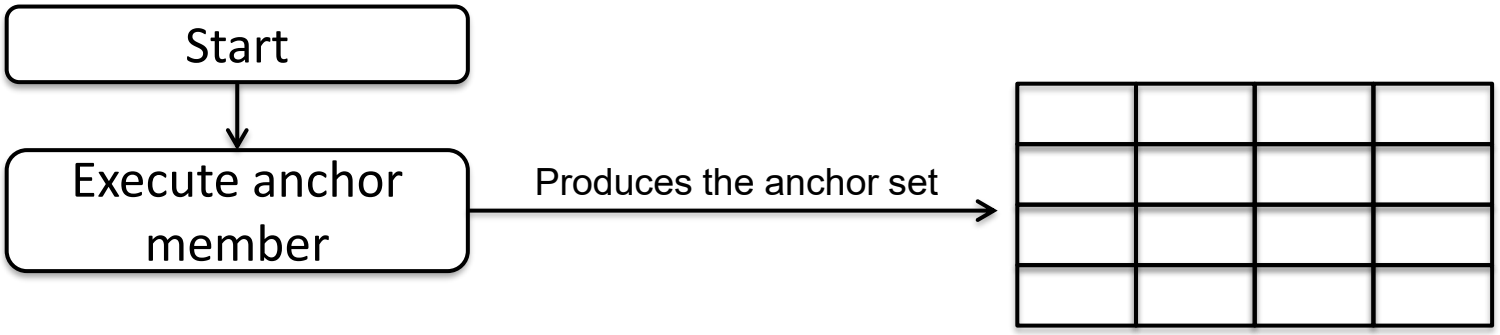
Recursive queries (cont.)

- A recursive query in SQL is a query that calls itself repeatedly until it reaches a termination condition
- Structure
  - Common Table Expression (CTE)
    - A CTE is a named temporary result set within an SQL query that can be referenced within the main query
  - Recursive Part
    - The recursive part of the CTE refers back to the CTE itself
    - It is essential to have two parts in a recursive CTE
      - the initial (or anchor) part that establishes the base case or starting point of the recursion
      - the recursive part which iteratively builds upon the initial result set

```
WITH RECURSIVE cte AS
(
-- Base(anchor) query
...
UNION ALL
-- Recursive query
...
)
SELECT ...
FROM cte;
```

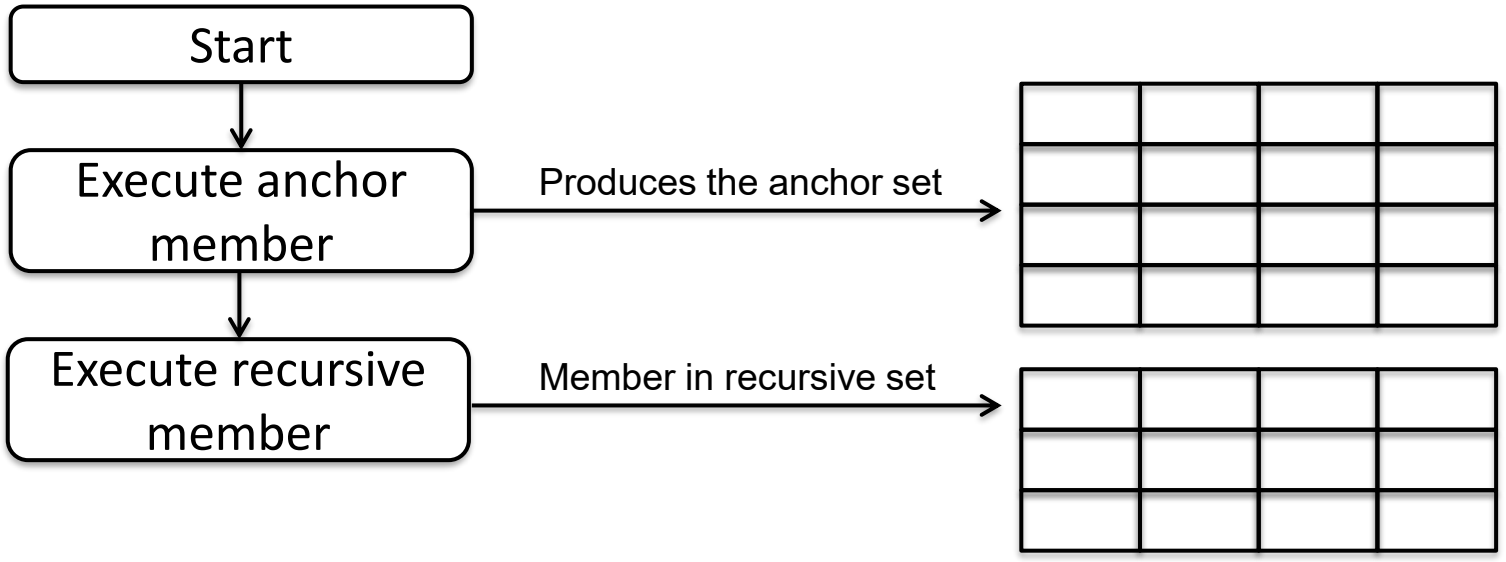
Recursive queries (cont.)

- Steps involved



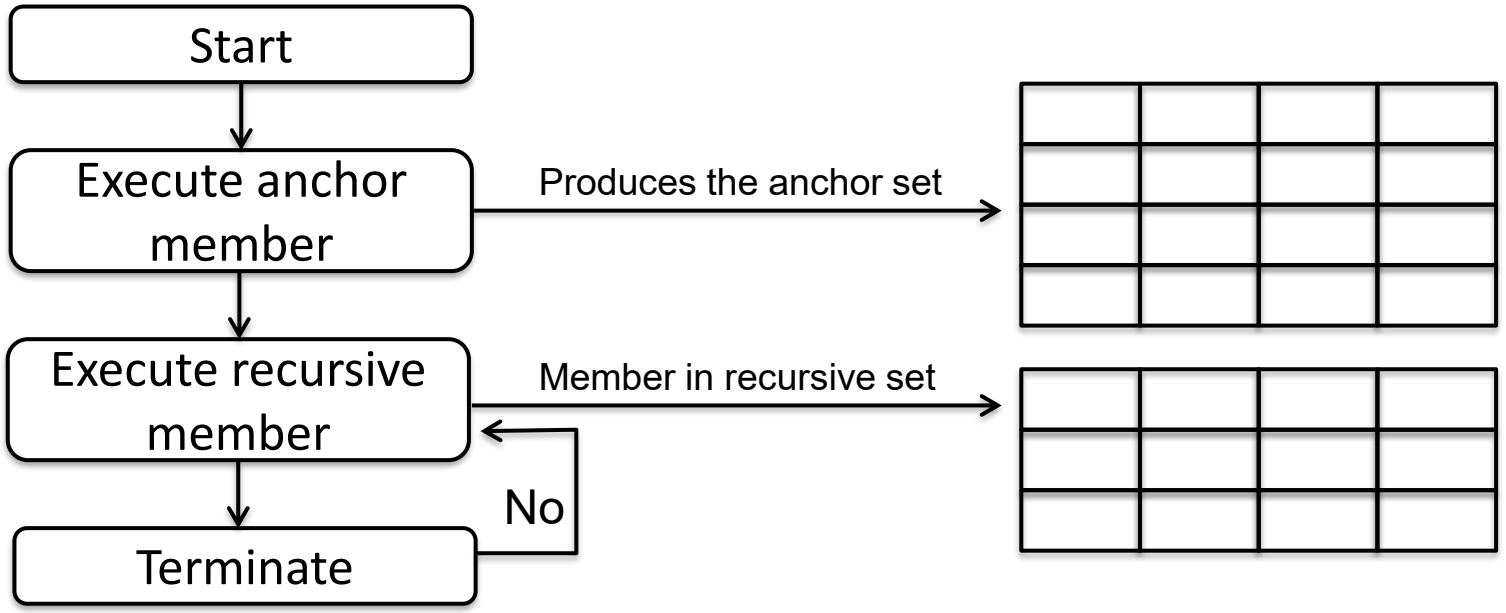
Recursive queries (cont.)

- Steps involved



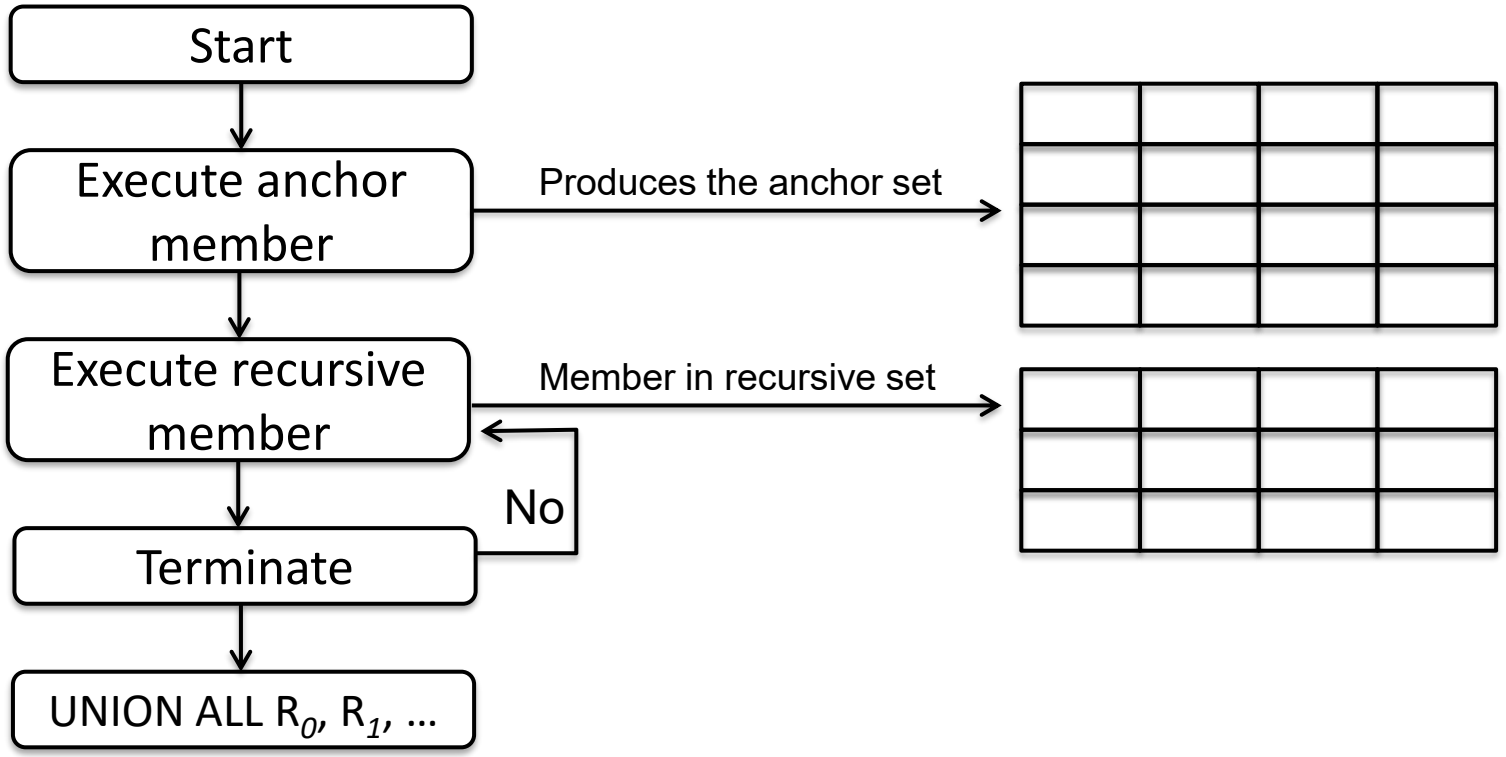
Recursive queries (cont.)

- Steps involved



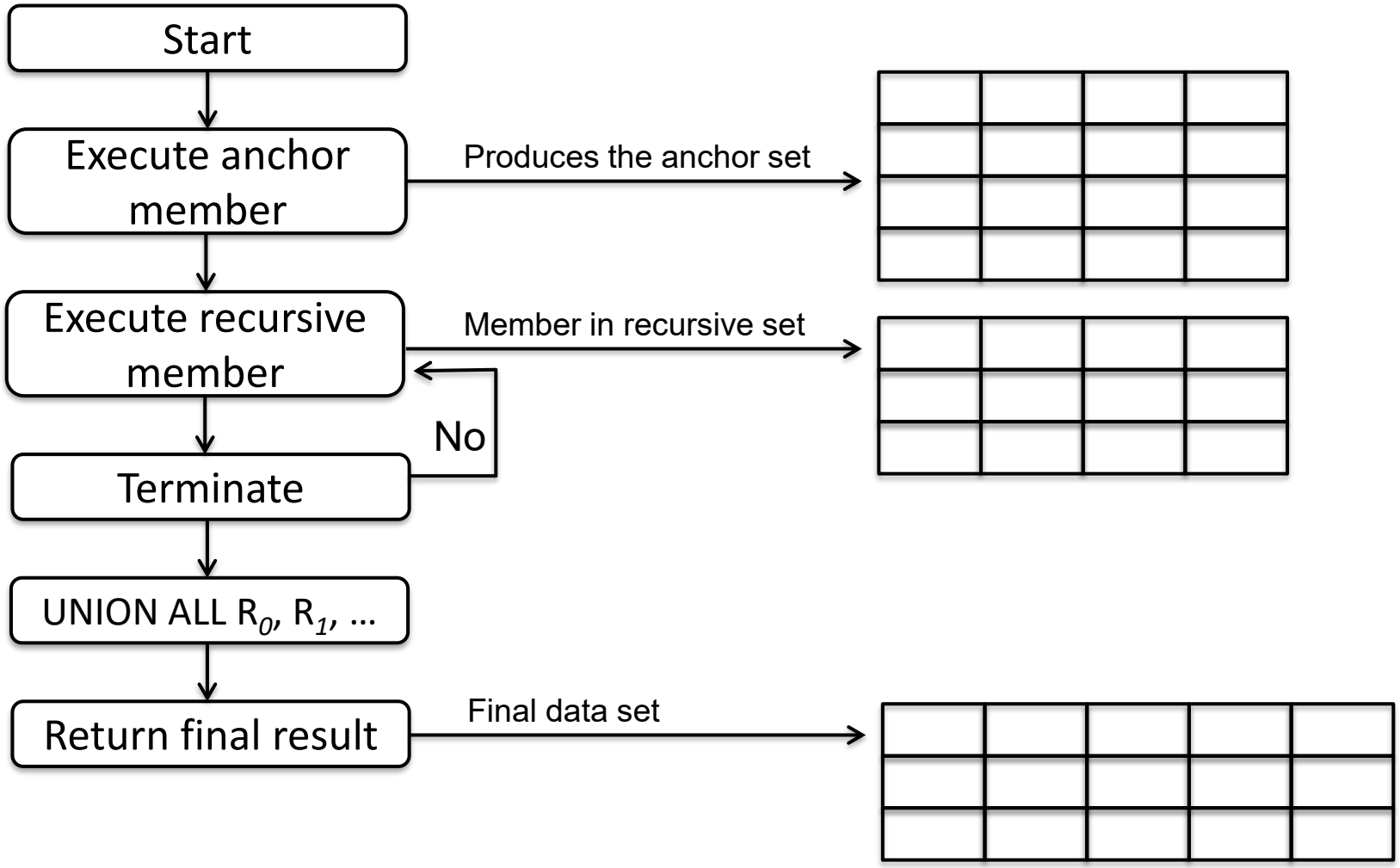
Recursive queries (cont.)

- Steps involved



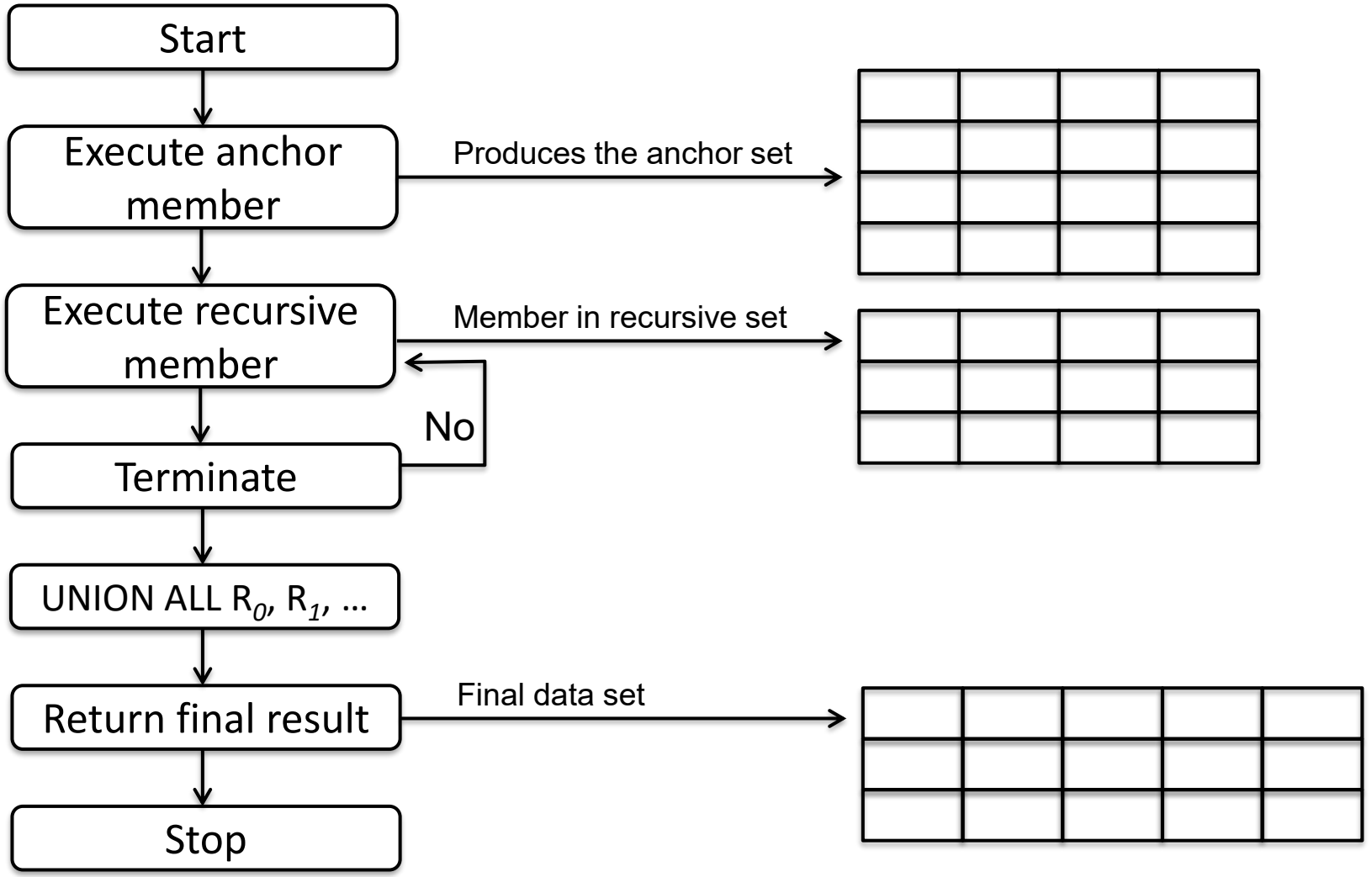
Recursive queries (cont.)

- Steps involved



Recursive queries (cont.)

- Steps involved



This is the end