Hierarchies on data
- and how to handle them with SQLScript

Jörg Brandeis

Me

Jörg Brandeis

- Freelancer
- BW Consulting
- SQLScript Trainings – every month here in Mannheim plus Inhouse-Trainings for customers
- Author of the Book „SQLScript [für|for] SAP HANA“
- Focus on technic – I am a developer!

Contact:

www.brandeis.de

joerg@brandeis.de

@joerg_brandeis

Xing, LinkedIn

The usual approach to work with hierarchies in the programs during loading time or in a variable exit is a **recursive algorithm**.

SQLScript doesn’t allow recursive logic. So I looked for an alternative approach for the same logic. And I found out, that there are many different solutions for storing and processing hierarchical data in SQL.

Requirement: **Find all nodes blow a given node.**

I show different ways to store and process the data to fullfil this requirement.

Criteria:

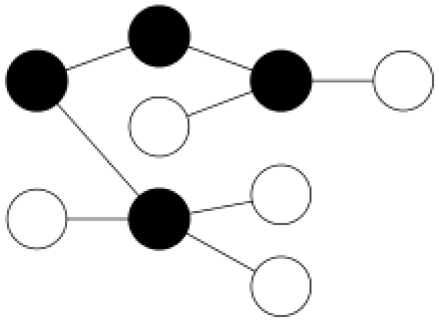
- **Performance at reporting time**
- Data volume
- Duration of loading

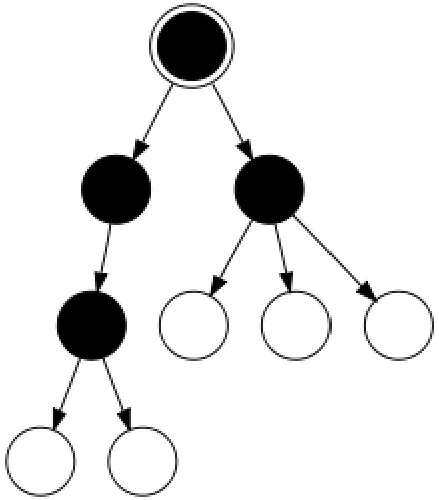
What is a hierarchy

“A hierarchy (…) is an arrangement of items (objects, names, values, categories, etc.) in which the items are represented as being "above", "below", or "at the same level as" one another. ”

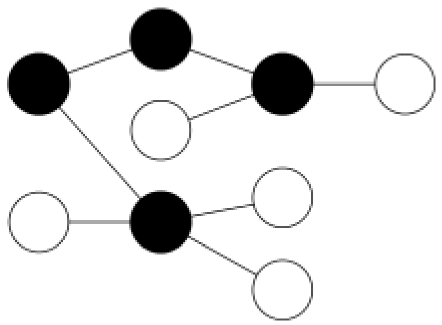
(Wikipedia, Hierarchy, 3.5.2019)

A **hierarchy** corresponds to a **directed rooted tree** in the graph theory.





A *tree* is an undirected graph G that satisfies any of the following equivalent conditions:



- G is connected and acyclic (contains no cycles).
- G is acyclic, and a simple cycle is formed if any edge is added to G.
- G is connected, but would become disconnected if any single edge is removed from G.
- G is connected and the 3-vertex complete graph K_3 is not a minor of G.
- Any two vertices in G can be connected by a unique simple path.

If G has finitely many vertices, say n of them, then the above statements are also equivalent to any of the following conditions:

- G is connected and has $n - 1$ edges.
- G is connected, and every subgraph of G includes at least one vertex with zero or one incident edges. (That is, G is connected and 1-degenerate.)
- G has no simple cycles and has $n - 1$ edges.

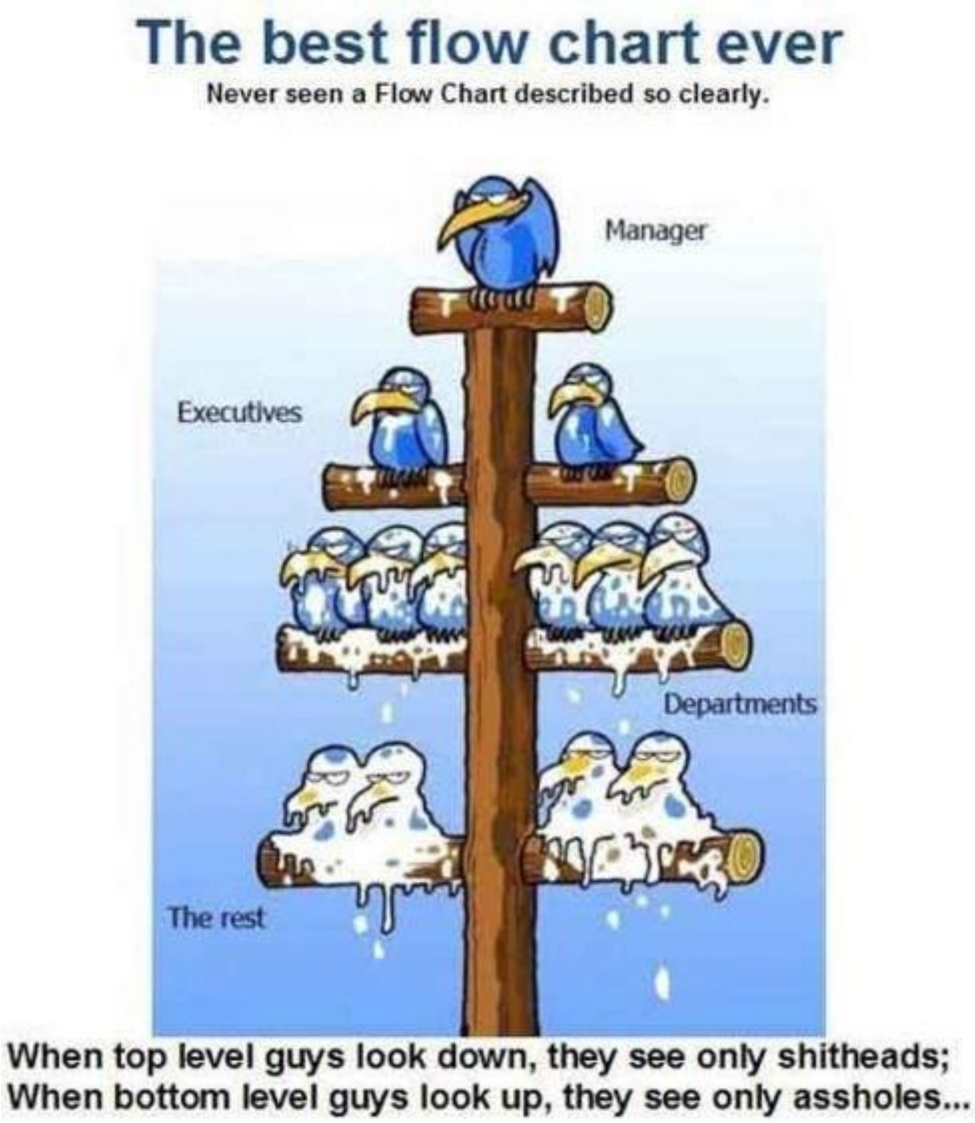
https://en.wikipedia.org/wiki/Tree_(graph_theory)#Properties

Tree vs. Hierarchy

A hierarchy has a semantic between the levels, e.g.

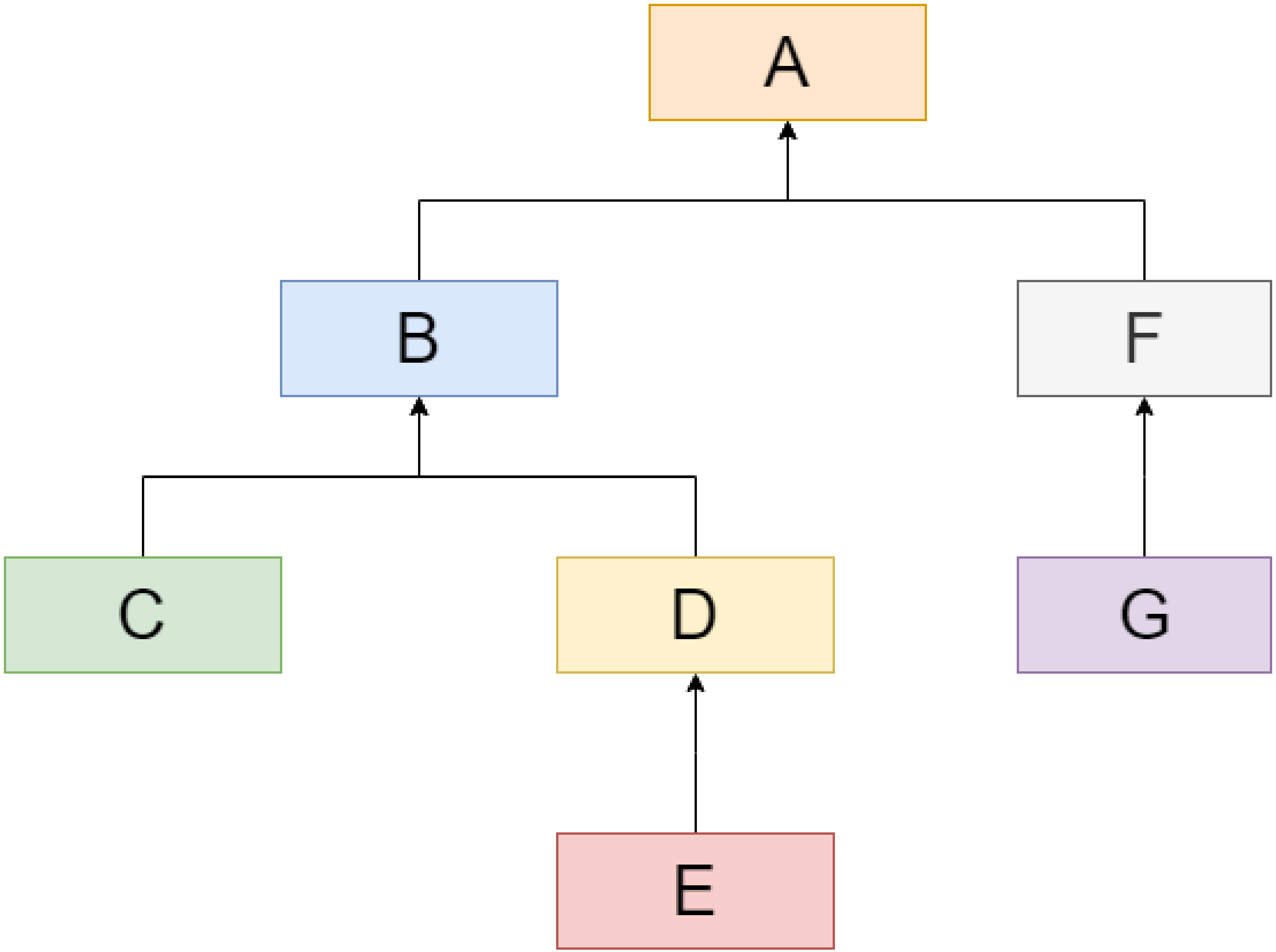
- Is above / below
- Consists of / is part of
- Give orders / has to obey the orders

A tree in the graph theory has no defined semantics between two connected nodes.


The best flow chart ever
Never seen a Flow Chart described so clearly.
Manager
Executives
Departments
The rest
When top level guys look down, they see only shitheads;
When bottom level guys look up, they see only assholes...

Examples for real world hierarchies

- Company structure
- Taxonomies of things, e.g. Animals
- Accounts in a chart of accounts (COA)
- Bill of Material (BOM)


A
B
F
C
D
G
E

Terminology for hierarchies in the IT

Node – an element in the hierarchy

Root node – the upmost node

Child node – a node below an other node

Parent node – a node above an other node

Sibling – a node with the same parent node

Leafnode – a node without child nodes

Level – the distance from the root node + 1

Ordered hierarchies – The sibling nodes have a fixed order. This order can be arbitrary or the order can be produced by sorting by a criteria, e.g. nodename

For an ordered hierarchy, you need to store extra information.

Balanced hierarchies

The nodes on the same level have the same nodetype. The number of levels is fixed.

Example: Geography

- Continent
  - Country
    - Region
      - City
        - Street
          - House
            - Flat
              - Room

Unbalanced hierarchies

The number of levels is not fixed. The nodetype of the levels can be different.

Examples:

- Employees and their superiors in a company
- A chart of accounts

Representation of hierarchies in the database

There are different formats to store hierarchy information in the database, e.g.:

1. In a flat structure:
   a. In a table with one column per level **or**
   b. in a normalized way with multiple tables
2. As a parent/child relation – And how the SAP BW is storing hierarchy information
3. As nested sets
4. Resolved hierarchy: Each node with all its children
5. With a path string ROOT=>LEVEL1=>LEVEL2=>…=>LEAF
6. … Other suggestions?

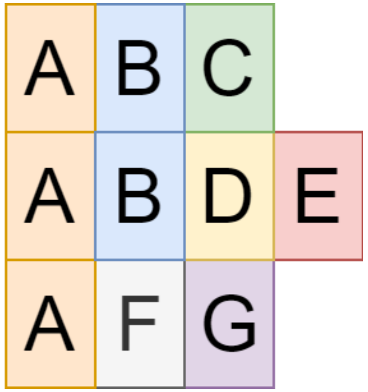
The following descriptions and examples are reduced to the absolute minimum. Additional information, e.g. the node level, can easily be enhanced and have a big impact on the performance of some algorithms.

Flat structure – Favorite for balanced hierarchies

In a table with one column per level

- Intuitive
- Redundant information
- Only for balanced hierarchies

City	Region/State	Country	Continent
Mannheim	Baden-Württemberg	Germany	Europe
Karlsruhe	Baden-Württemberg	Germany	Europe
Frankfurt a.M.	Hessen	Germany	Europe
Berlin	Berlin	Germany	Europe
Paris	Île-de-France	France	Europe
New York	New York	USA	North america
Harrisburg	Pennsylvania	USA	North america



In a normalized way with multiple tables

- Still intuitive
- No redundancy
- Only for balanced data

Region/State	Country
Baden-Württemberg	Germany
Hessen	Germany
Berlin	Germany
Île-de-France	France
New York	USA
Pennsylvania	USA

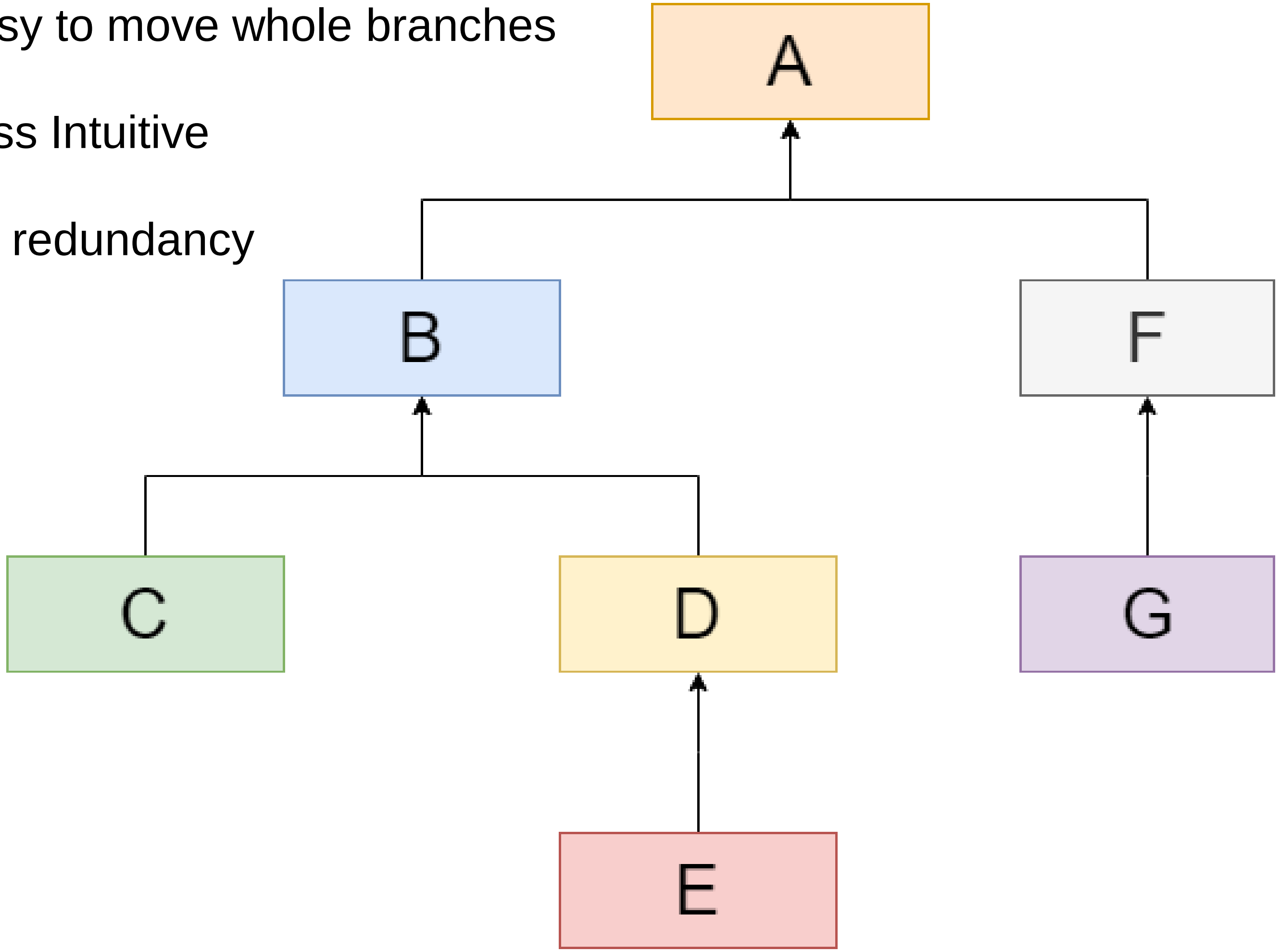
City	Region/State
Mannheim	Baden-Württemberg
Karlsruhe	Baden-Württemberg
Frankfurt a.M.	Hessen
Berlin	Berlin
Paris	Île-de-France
New York	New York
Harrisburg	Pennsylvania

Country	Continent
Germany	Europe
France	Europe
USA	North america

No special SQL-Skills needed – This is normalization

Parent/child relation

- Very flexible - no limits regarding the levels
- Easy to move whole branches
- Less Intuitive
- No redundancy



1	A	
2	B	1
3	C	2
4	D	2
5	E	4
6	F	1
7	G	6

Hierarchies in SAP BW – Data Format

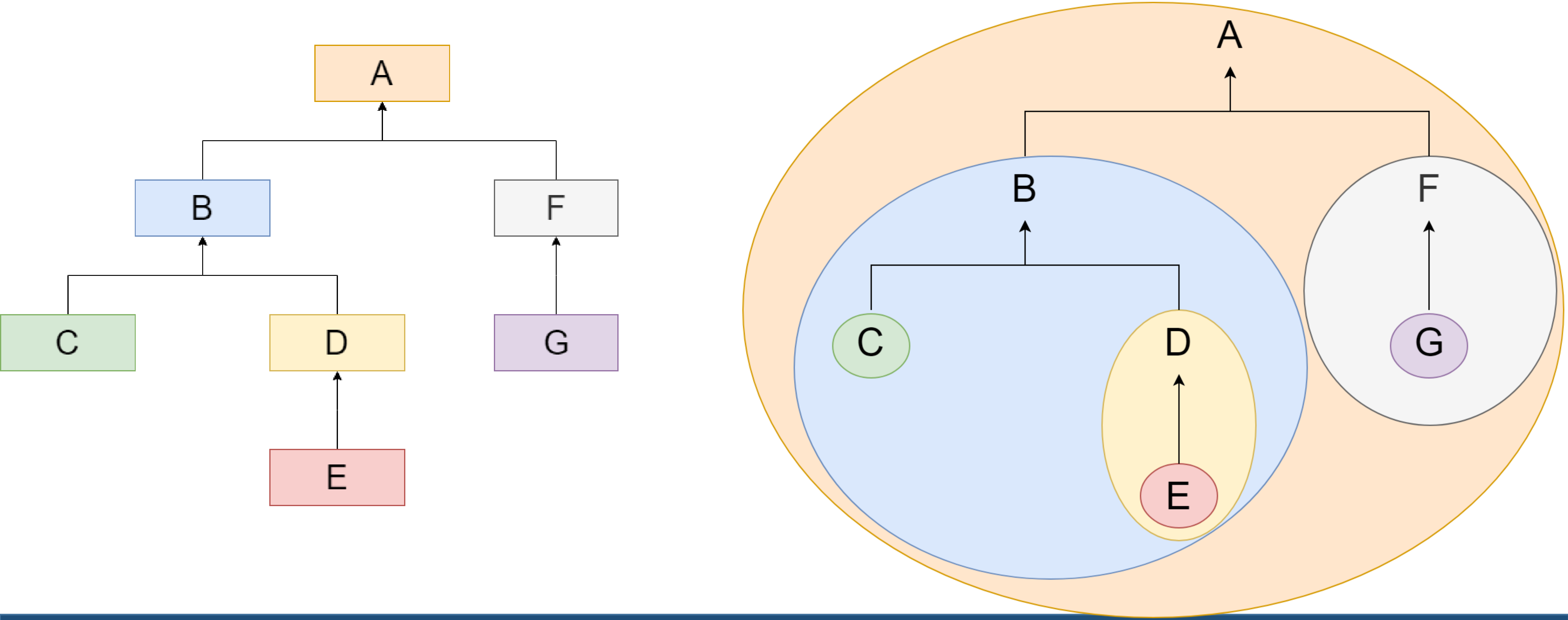
- The hierarchies for an InfoObject are stored in a **Parent/Child format** in a generated table with a standardized structure.
- For each parent node, the **first child node** is stored.
- For each child node is the **next sibling** stored.
- Each node has a **type** information (InfoObject)

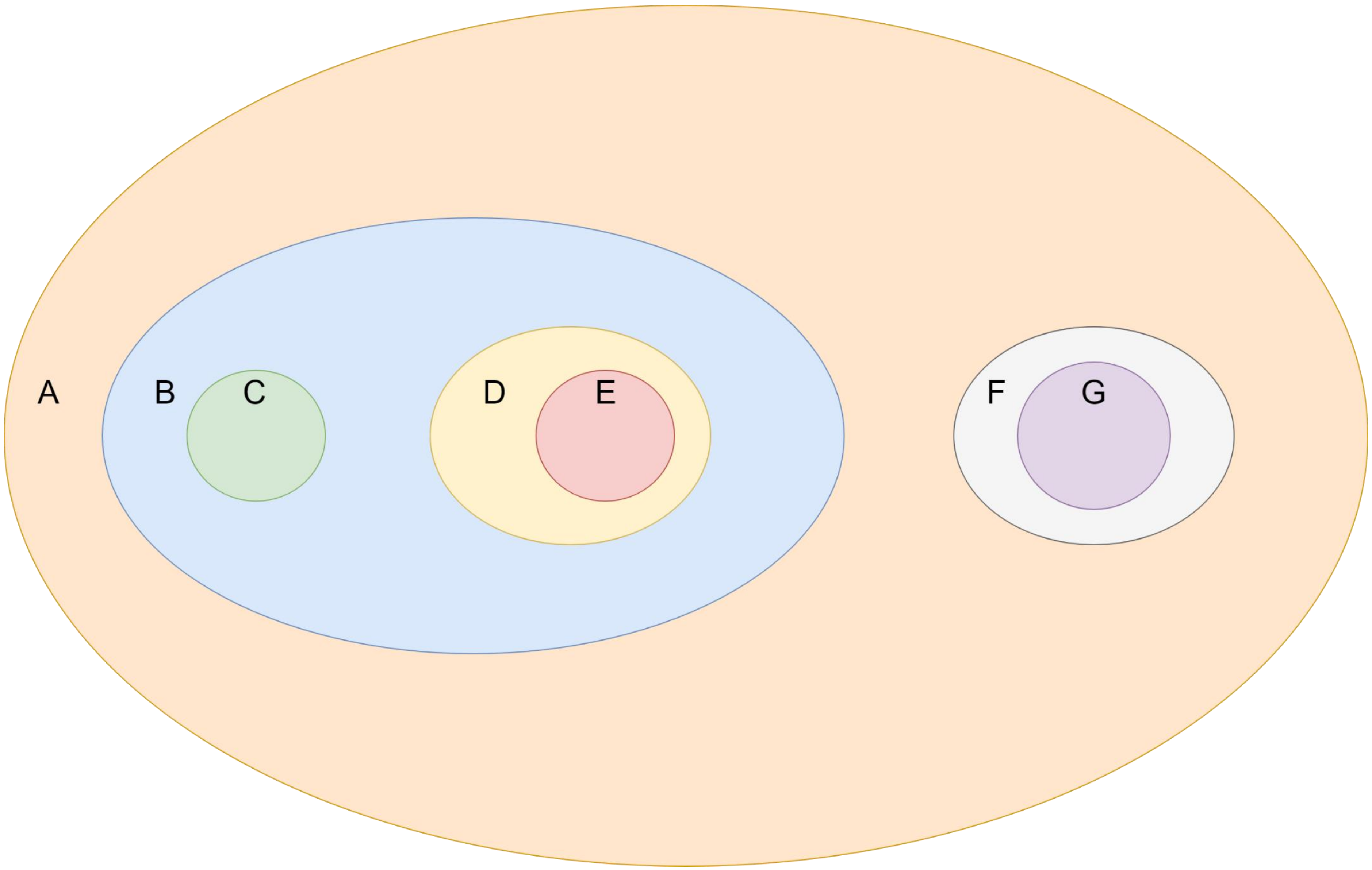
These information about the order are very fragile, if you upload hierarchies manually. It is easy, to create inconsistent hierarchies.

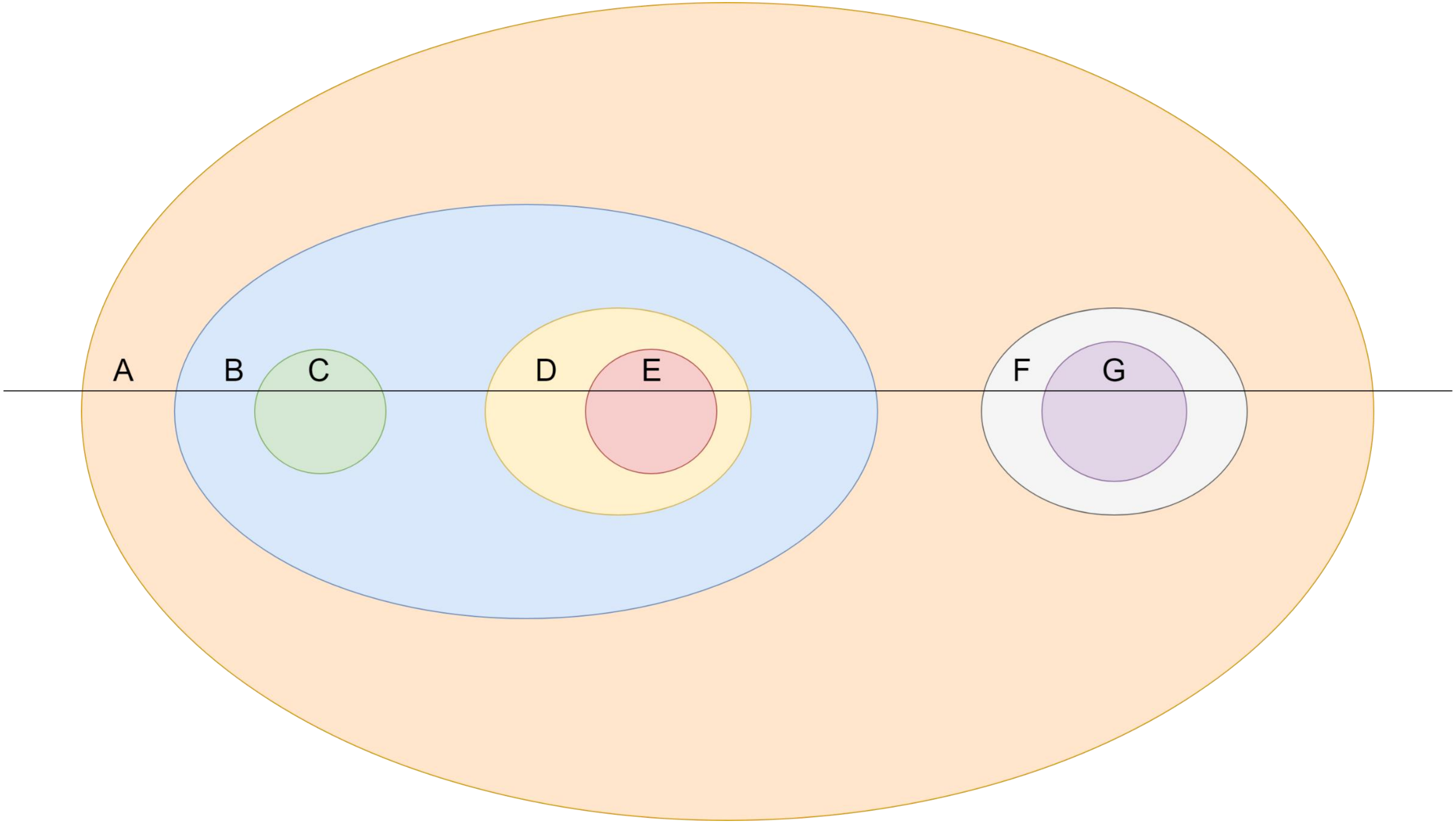
```
CREATE COLUMN TABLE "SAPDEV"."/BIC/HPROFITCT" (
 "HIEID" NVARCHAR(25) DEFAULT '' NOT NULL ,
 "OBJVERS" NVARCHAR(1) DEFAULT '' NOT NULL ,
 "NODEID" NVARCHAR(8) DEFAULT '00000000' NOT NULL ,
 "IOBJNM" NVARCHAR(30) DEFAULT '' NOT NULL ,
 "NODENAME" NVARCHAR(1333) DEFAULT '' NOT NULL ,
 "TLEVEL" NVARCHAR(2) DEFAULT '00' NOT NULL ,
 "LINK" NVARCHAR(1) DEFAULT '' NOT NULL ,
 "PARENTID" NVARCHAR(8) DEFAULT '00000000' NOT NULL ,
 "CHILDID" NVARCHAR(8) DEFAULT '00000000' NOT NULL ,
 "NEXTID" NVARCHAR(8) DEFAULT '00000000' NOT NULL ,
 "INTERVL" NVARCHAR(1) DEFAULT '' NOT NULL ,
 CONSTRAINT "/BIC/HBI_PROFCT~0" PRIMARY KEY ("HIEID",
 "OBJVERS",
 "NODEID")) UNLOAD PRIORITY 5 AUTO MERGE
GROUP TYPE "sap.bw.iobj"
GROUP SUBTYPE "H"
GROUP NAME "PROFITCT"
```

Introduced by Joe Celko in the book “SQL for smarties”

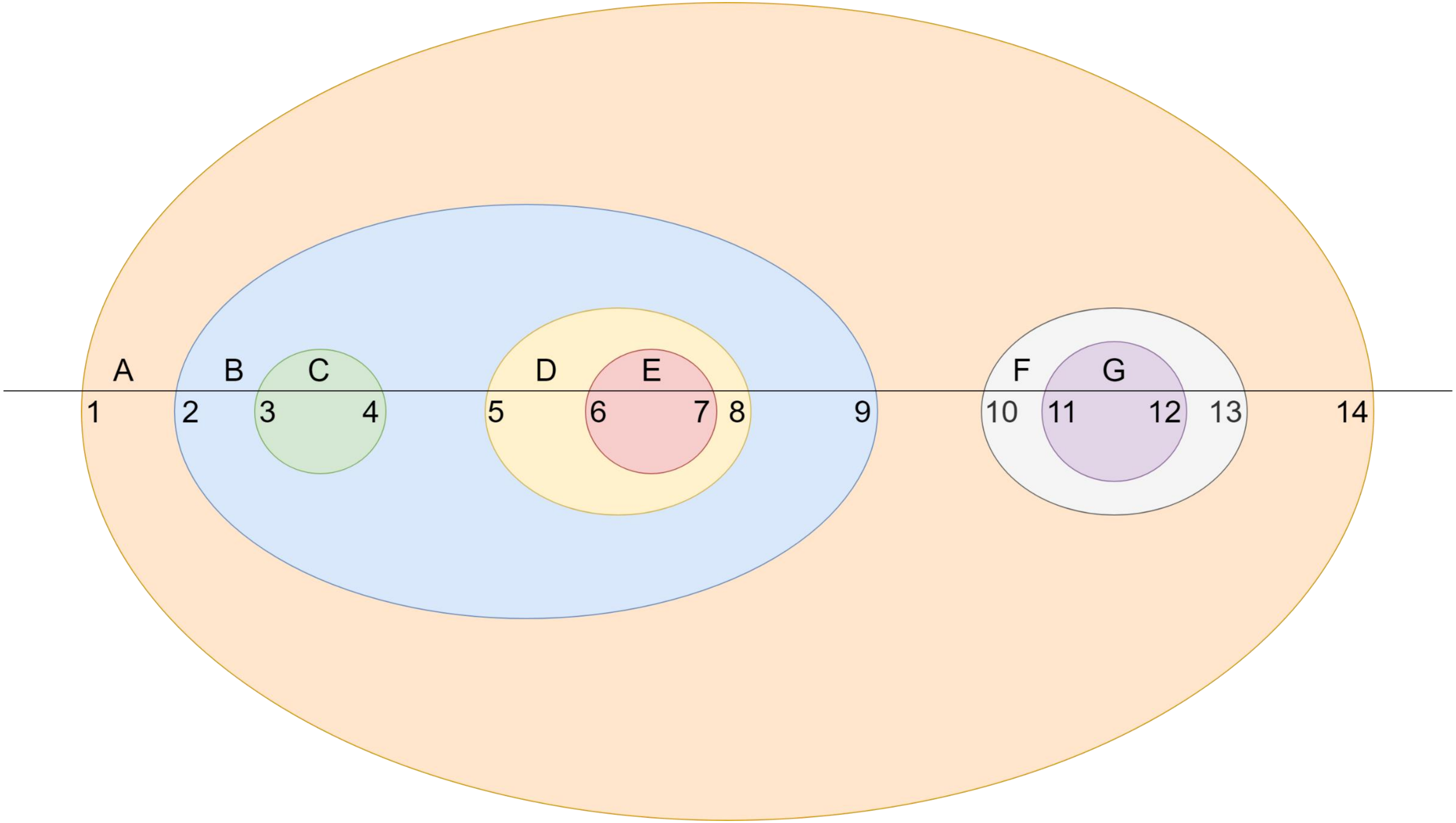
The basic idea: Think of a hierarchy node as a set. All nodes blow (they are also sets) are elements of this set.



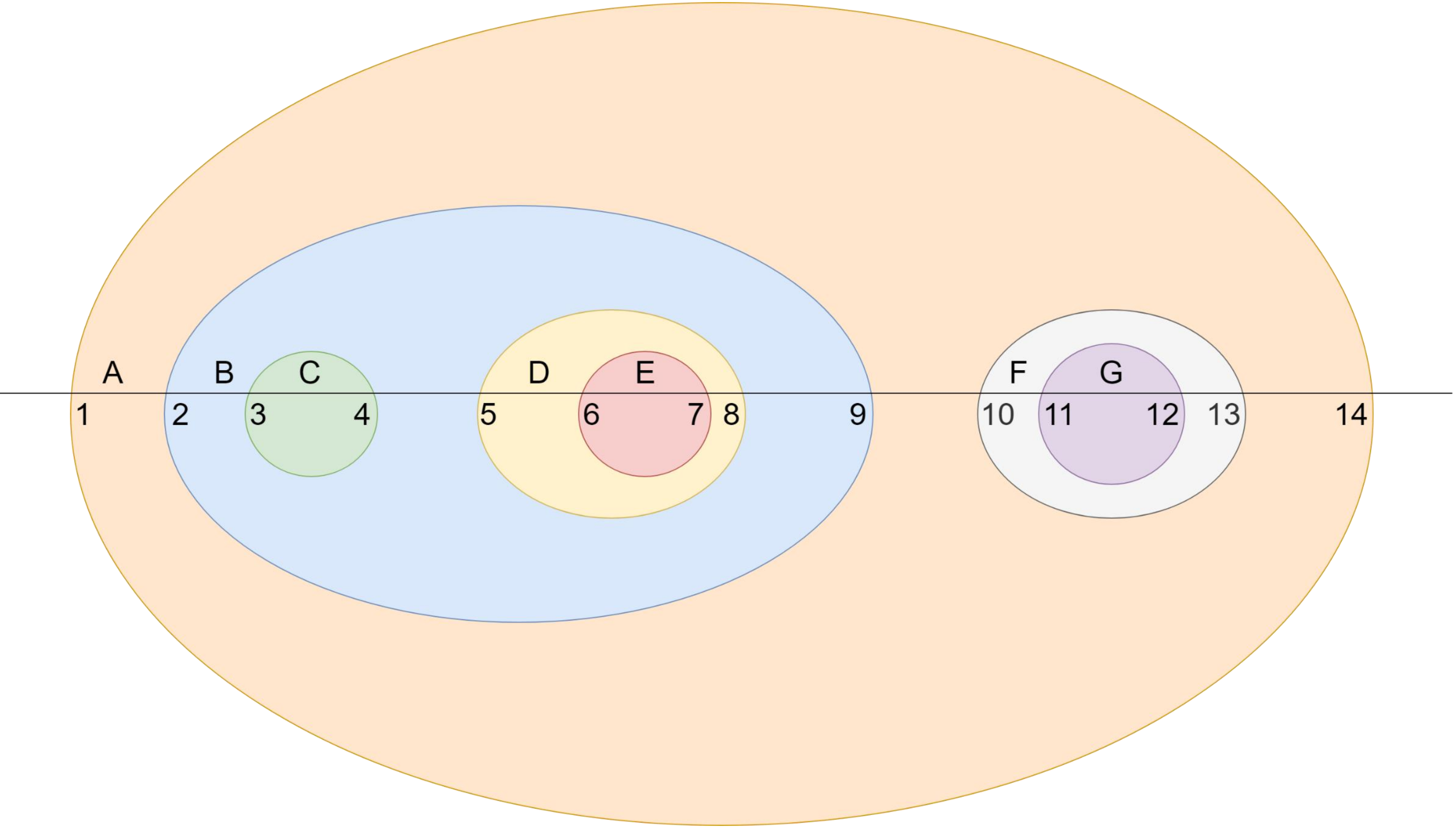

A
B
C
D
E
F
G


A
B
C
D
E
F
G

Nested sets



Nested sets – Table structure



A	1	14
B	2	9
C	3	4
D	5	8
E	6	7
F	10	13
G	11	12

Nested sets characteristics

- Changes in structure are expensive
- Not intuitive
- No redundancy
- But: For some usecases very efficent

A	1	14
B	2	9
C	3	4
D	5	8
E	6	7
F	10	13
G	11	12

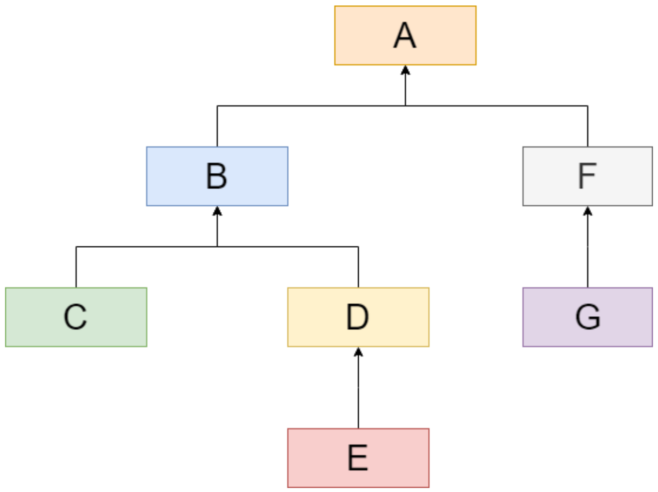
Resolved hierarchy: Each node with all its (grand)children

Store precalculated information: per node all its children

- High redundancy and data volume (about factor 10, depending on the depth of the hierarchy)
- Changes in structure are expensive

Optimized for the two requirements:

- All childrens of a node
- All parents of a node



A	B
A	C
A	D
A	E
A	F
A	G
B	C
B	D
B	E
D	E
F	G

Pros:

- Intuitive to read
- Like the flat format, but not limited by a number of columns

Cons:

- Limited by the length of datatypes, e.g. VARCHAR: 5000 chars
- Text operations are expensive
- High data volume
- High redundancy
- Expensive changes on hierarchy structure

1	A	A
2	B	A=>B
3	C	A=>B=>C
4	D	A=>B=>D
5	E	A=>B=>D=>E
6	F	A=>F
7	G	A=>F=>G

The hierarchy functions are SQL functions, that are available on the SAP HANA. They are very generic, which allows a wide range of usages.

Reference: SAP HANA Hierarchy Developer Guide

Introduced with HANA 2.0 SPS 01

Hierarchy functions can handle unclean data!

The generator functions

The **generator functions** create a generic hierarchy table from a parent/child or flat format aka. Leveled hierarchy.

1	A	
2	B	1
3	C	2
4	D	2
5	E	4
6	F	1
7	G	6

HIERARCHY()
HIERARCHY_SPANTREE()
HIERARCHY_TEMPORAL()

A	B	C	
A	B	D	E
A	F	G	

HIERARCHY_LEVELED()

A common, uniform Parent/Child format

- HIERARCHY_RANK
- HIERARCHY_TREE_SIZE
- HIERARCHY_PARENT_RANK
- HIERARCHY_LEVEL
- HIERARCHY_IS_CYCLE
- HIERARCHY_IS_ORPHANT
- NODE_ID
- PARENT_ID

+ additional fields from the source

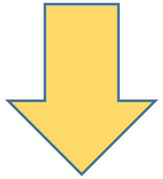
*"Specialized hierarchy generator functions translate the diverse relational source data into a generic and normalized tabular format that, for the sake of brevity, is just termed **HIERARCHY**."* , SAP Doku

Example of a HIERARCHY-Table

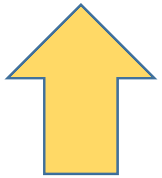
	HIERARCHY_RANK	HIERARCHY_TREE_SIZE	HIERARCHY_PARENT_RANK	HIERARCHY_LEVEL	HIERARCHY_IS_CYCLE	HIERARCHY_IS_ORPHAN	NODE_ID	PARENT_ID	NODENAME
1	7	1	6	7	0	0	1111111	1111110	AAAAAAA
2	8	1	6	7	0	0	1111112	1111110	AAAAAAB
3	9	1	6	7	0	0	1111113	1111110	AAAAAAC
4	10	1	6	7	0	0	1111114	1111110	AAAAAAD
5	11	1	6	7	0	0	1111115	1111110	AAAAAAE
6	12	1	6	7	0	0	1111116	1111110	AAAAAAF
7	13	1	6	7	0	0	1111117	1111110	AAAAAAG
8	14	1	6	7	0	0	1111118	1111110	AAAAAAH
9	15	1	6	7	0	0	1111119	1111110	AAAAAAI
10	17	1	16	7	0	0	1111121	1111120	AAAAABA
11	18	1	16	7	0	0	1111122	1111120	AAAAABB
12	19	1	16	7	0	0	1111123	1111120	AAAAABC
13	20	1	16	7	0	0	1111124	1111120	AAAAABD
14	21	1	16	7	0	0	1111125	1111120	AAAAABE
15	22	1	16	7	0	0	1111126	1111120	AAAAABF
16	23	1	16	7	0	0	1111127	1111120	AAAAABG
17	24	1	16	7	0	0	1111128	1111120	AAAAABH
18	25	1	16	7	0	0	1111129	1111120	AAAAABI
19	27	1	26	7	0	0	1111131	1111130	AAAAACA
20	28	1	26	7	0	0	1111132	1111130	AAAAACB
21	29	1	26	7	0	0	1111133	1111130	AAAAACC
22	30	1	26	7	0	0	1111134	1111130	AAAAACD
23	31	1	26	7	0	0	1111135	1111130	AAAAACE
24	32	1	26	7	0	0	1111136	1111130	AAAAACF

The three navigation functions work on an existing hierarchy table and return a hierarchy table themselves.

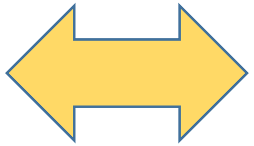
- HIERARCHY_DESCENDANTS()



- HIERARCHY_ANCESTORS()



- HIERARCHY_SIBLINGS()



They use these parameters:

- SOURCE – The HIERARCHY-Table
- START – The start of the navigation. This can be multiple nodes
- DISTANCE – To limit the results to a certain distance from the START. Not available for HIERARCHY_SIBLINGS()

The Aggregate Functions

“The hierarchy … aggregation function provides optimized hierarchical aggregate capabilities. By reusing results of subordinate notes, all aggregates can be calculated by one single linear index traversal.”, [*SAP Doku*]

The aggregate functions are parameterized like the navigation functions, but you can also define aggregated columns, a join to a fact table and specify some extra features, like subtotals.

- HIERARCHY_DESCENDANTS_AGGREGATE()
- HIERARCHY_ANCESTORS_AGGREGATE()

```
do begin

    lt_hier =  select *
                 from hierarchy(
                         source (
                              select nodeid   as node_id,    --required name
                                     parentid as parent_id   --required name
                                from hier_pc )
                         start where parentid is null ); --root-node
  select * from :lt_hier; --to display intermediate values....
    lt_tmp =
                 select start_id as nodeid,
                        node_id  AS childid
                 from hierarchy_descendants(
                          source :lt_hier
                          START ( SELECT hierarchy_rank AS start_rank, --required name
                                         node_id AS start_id
                                  FROM :lt_hier ) )
                 order by nodeid asc,
                          childid asc ;
   select * from :lt_tmp; --to display the result;
end
```

Requirement: Find all subnodes (NODEID, NODENAME) for a given NODEID.

Procedure name	Description	#
HIER_LOOKUP_PC	Parent/Child with hierarchy function	
HIER_LOOKUP_NS	Nested Sets	
HIER_LOOKUP_PC_IMP	Parent/Child with a manualy build, imperative SQLScript procedure	
HIER_LOOKUP_PGC	Parent/(grand-)children – The data is already precalculated	
HIER_LOOKUP_PATH	Data stored with a complete Path	

Lookup with Nested Sets

```
create procedure hier_lookup_ns(in iv_nodeid int,
                                out et_result table(nodeid int, nodename nvarchar(30) ) )
as begin
   declare lv_left_value int;
   declare lv_right_value int;
   select left_value,
          right_value
          into lv_left_value, lv_right_value
          from hier_ns
          where nodeid = :iv_nodeid;

et_result = select nodeid,
                   nodename
                   from hier_ns
                   where left_value >= :lv_left_value
                     and right_value <= :lv_right_value;
end
```

A slow version of the same algorithm…

```
create procedure hier_lookup_ns_slow(in iv_nodeid int,
                                     out et_result table(nodeid int, nodename nvarchar(30) )
)
as begin

et_result = select nodeid,
                   nodename
                   from hier_ns
                   where left_value >=
                             (select left_value from hier_ns where nodeid = :Iv_nodeid)
                     and right_value <=
                             (select right_value from hier_ns where nodeid = :Iv_nodeid);
end
```

This is 10 times slower…

Lookup Parent/Child with hierarchy function

```
create procedure hier_lookup_pc(in  iv_nodeid int,
                                out et_result table(nodeid int, nodename nvarchar(30) ) )
as begin
    et_result =  select node_id as nodeid,
                        nodename
                        from hierarchy_descendants ( source hier_hf
                                                     start where node_id = :iv_nodeid );
 end
```

Lookup Parent / Child with a manual, imperative function

```
create procedure hier_lookup_pc_imp(in iv_nodeid int,
                                    out et_result table(nodeid int, nodename nvarchar(30) ) )
as begin
    declare lv_cnt int;
lv_tmp_sel = select nodeid,
                    nodename
               from hier_pc
               where nodeid = :iv_nodeid;

while not is_empty(:lv_tmp_sel) do

  et_result = select * from :et_result
              union
              select * from :lv_tmp_sel;

  lv_tmp_sel = select pc.nodeid,
                      pc.nodename
                 from hier_pc as pc
                 inner join :lv_tmp_sel
                 on pc.parentid = :lv_tmp_sel.nodeid;

end while;

end
```

```
create procedure hier_lookup_path(in iv_nodeid int,
                                  out et_result table(nodeid int, nodename nvarchar(30) ) )
as begin
declare lv_pattern nvarchar(5000);
select  path || '=>%'
into lv_pattern
from hier_path
where nodeid = :iv_nodeid;

et_result = select nodeid,
                   nodename
            from hier_path
            where path like :lv_pattern
            union all
            select nodeid,
                   nodename
            from hier_path
    where nodeid = :iv_nodeid;
end
```

Lookup from the precalculated Parent/(grand)-child table


```
create procedure hier_lookup_pgc(in iv_nodeid int,
                                 out et_result table(nodeid int, nodename nvarchar(30) ) )
as begin
et_result = select pgc.childid as nodeid,
                   pc.nodename
                   from hier_pgc as pgc
                   left outer join hier_pc as pc
                   on pgc.childid = pc.nodeid
                   where pgc.nodeid = :iv_nodeid;
end
```

Which algorithm is the fastest?

Procedure name	Description	#
HIER_LOOKUP_PC	Parent/Child with hierarchy function	
HIER_LOOKUP_NS	Nested Sets	
HIER_LOOKUP_PC_IMP	Parent/Child with a manualy build, imperative SQLScript procedure	
HIER_LOOKUP_PGC	Parent/(grand-)children – The data is already precalculated	
HIER_LOOKUP_PATH	Data stored with a complete Path	

General observation: The first execution was always slower then the subsequent executions of a SELECT Query. This was caused by the LOAD of the tables. To get stable results, the tables should be loaded before the runtime measurement.

The problem with my little system: When i load the big table HIER_PGC, the other tables were partitialy is unloaded.

Workaround: Test the algorithms seperately.

Result of the measurement

ALGORITHM	AVG(RUNTIME_MS)	MIN(RUNTIME_MS)	MAX(RUNTIME_MS)
Hierarchy function	537	516	558
Nested Sets	7	7	8
Parent/Child imper.	24	22	28
Path string search	7	7	7
Precalculated	14	10	26

Result for different number of rows in the result

ALGORITHM	MIN(RUNTIME_MS)	#rows
Hierarchy function	517	10
Hierarchy function	516	91
Hierarchy function	515	820
Hierarchy function	518	7.381
Hierarchy function	556	66.430
Hierarchy function	804	597.871
Nested Sets	5	10
Nested Sets	5	91
Nested Sets	6	820
Nested Sets	8	7.381
Nested Sets	13	66.430
Nested Sets	37	597.871
Parent/Child imper.	7	10
Parent/Child imper.	10	91
Parent/Child imper.	13	820
Parent/Child imper.	23	7.381
Parent/Child imper.	51	66.430
Parent/Child imper.	252	597.871
Path string search	6	10
Path string search	5	91
Path string search	5	820
Path string search	7	7.381
Path string search	10	66.430
Path string search	35	597.871
Precalculated	8	10
Precalculated	8	91
Precalculated	8	820
Precalculated	13	7.381
Precalculated	25	66.430
Precalculated	91	597.871

Observations

- Algorithms on Nested Sets and Path! are the fastest.
- The precalculated result is slower
- Algorithm with manual, imperative SQLScript is fast for small result sets, but it doesn‘t scale well.
- Algorithms with HANA Hierarchy Functions are much slower.

Interpretation of the observations

The algorithms that decides on a single line are the fastest. The runtime of the string comparison seems not to be so bad.

The precalculated result is not faster! This can be caused by the much larger table.

The runtime of the imperative algorithm depends of the number of loops.

Conclusions

- The **LIKE-predicate** is faster than expected
- The **Hierarchy Functions** are comfortable, if you don‘t want to implement an algorithm yourself. But slower.
- The **imperative logic** is not so bad.
- Processing hierarchies is easier in **SQLScript** than in **ABAP**. All algorithms are very small.
- Access to unloaded tables/columns is very slow. If an algorithm is significantly faster in subsequent executions, check the load status.
- **Try different approaches, even the silly ones.**
- **Expect the unexpected!**

You can combine the approaches, if you have different requirements.

For example: Nested Sets plus Parent/Child plus Level Information

Jörg Brandeis

Contact:
www.brandeis.de
joerg@brandeis.de
@joerg_brandeis
Xing, LinkedIn