Note: even though the examples provided are based on Oracle, they should also apply to any RDBMS supporting literal, case-sensitive primary keys. Access, to say one, accepts literal primary keys but it's case-insensitive (in any given table, the primary key "AaA" violates uniqueness if primary key "aAa" exists)

Note: the examples illustrated here apply to hierarchies made of a fixed number of named ranks, but the underlying reasoning applies equally well to contexts where the number of hierarchical levels (=ranks) isn't initially known.

Compact, searchable-by-LIKE hierarchical trees in RDBMS tables

Hierarchies

Relational databases including hierarchical data, e.g. biological taxonomies (Domain-Kingdom-Type-Class-Order…) or administrative geographical information (Continent-Nation-Region-Province…) pose very specific problems when managing queries such as (descriptive metalanguage syntax is used):

- Select all the rows from the "Photographs" Table pertaining to butterflies in the family Lycaenidae from Emilia Romagna
- Select all the rows from the "Photographs" Table pertaining to Mammals
- Select all the rows from the "Photographs" Table pertaining to Europe

When designing the database, it would be very unreasonable (as it would violate the 3rd Normal Forms, "nothing but the key") to include in the "Photographs" Table as many columns as the elements of the geographical and of the taxonomical hierarchy.
The correct solution is including just two foreign keys that represent the lowest-ranked (e.g. geographical Locality, Species or Subspecies) entity to which the table row refers.

A similar need for normalization is present for the parent tables that we may call "Taxonomy" and "Geonomy": the hierarchical cascades from continent to locality and from domain to subspecies cannot reasonably engage as many columns as the ranks.

Geonomy Table							
GEO_PK	Continent	Nation	Region	Province	Town	District	Locality
1	Europe						
2	Europe	Italy					
3	Europe	Italy	Emilia Romagna				
…	…	…	…	…	…	…	…
99	Europe	Italy	Emilia Romagna	Ferrara	Poggio Renatico	Chiesa Vecchia	Via Chiesa Vecchia, 45

Although 3NF compliant, a cascade of foreign-key-referenced tables, one table per rank, would also have some drawbacks, in particular lengthy, join-intensive, prolix queries. Furthermore, the introduction of new intermediate ranks would require a redesign of the cascade, and it would be impossible to skip one or more attribution levels, e.g. assign a Locality to a Town that has no Districts.

Locality Table		
LOC_PK	Locality	DIST_FK
1	Via Chiesa Vecchia, 45	1
…	…	…

Districts Table		
DIST_PK	District	TOWN_FK
1	Italy	1
…	…	…

Towns Table		
TOWN_PK	Town	PRO_FK
1	Poggio Renatico	1
…	…	…

Provinces Table		
PRO_PK	Province	REG_FK
1	Ferrara	1
…	…	…

Regions Table		
REG_PK	Region	NAT_FK
1	Emilia Romagna	1
…	…	…

Nations Table		
NAT_PK	Nation	CONT_FK
1	Italy	1
…	…	…

Continents Table	
CONT_PK	Continent
1	Europe
…	…

The proper solution requires to include in the parent table a "BELONGS_IN" column as a foreign key referencing the very same table. Rank names, that are not the main subject here but that in fact may improve clarity of query results, can be placed in their own special table.

Geonomy Table			
GEO_PK	Name	BELONGS_IN	GEO_RANK_FK
1	Europe		10
2	Italy	1	20
3	Emilia Romagna	2	30
4	Ferrara	3	40
5	Poggio Renatico	4	50
6	Chiesa Vecchia	5	60
...	...	...	...
99	Via Chiesa Vecchia, 45	6	70

Geo_Ranks Table	
GEO_RANK_PK	Name
10	Continent
20	Nation
30	Region
40	Province
50	Town
60	District
70	Locality

The well-designed, self-referenced parent tables are compact and comply with 3NF, which is very desirable, but lack a feature that would be granted by the unadvisable design that includes one column for each possible rank: in the latter case, i could well

```
SELECT * FROM GEONOMY WHERE PROVINCE = 'Ferrara'
```

and immediately get a list including the Province of Ferrara and all the towns, districts and localities belonging in the province. Instead, the correctly designed Geonomy table requires some computation to obtain the full content of the hierarchical tree branch rooted in the Province of Ferrara.

Built-in RDBMS-level hierarchical query management

To that purpose, Oracle supports hierarchical queries[1] thanks to the CONNECT BY condition complemented by features that include the PRIOR operator, by the LEVEL pseudo-column and by the SYS_CONNECT_BY_PATH() function as in the following example:

```
SELECT GEO_PK, NAME, GEO_RANK_FK, BELONGS_IN, LEVEL,
       SYS_CONNECT_BY_PATH(NAME, '/') AS GEO_HIERARCHY
FROM   GEONOMY
WHERE NAME = 'Ferrara'
CONNECT BY PRIOR GEO_PK = BELONGS_IN
ORDER BY LEVEL;
```

Such a query returns results similar to the following:

GEO_PK	Name	GEO_RANK_FK	BELONGS_IN	LEVEL	HIERARCHY
4	Ferrara	40	3	1	/Ferrara
4	Ferrara	40	3	2	/Emilia Romagna/Ferrara
4	Ferrara	40	3	3	/Italy/Emilia Romagna/Ferrara
4	Ferrara	40	3	4	/Europe/Italy/Emilia Romagna/Ferrara

The way the RDBMS manages hierarchical queries is complex and may be counterintuitive: one would expect that the example query above would return one row, when in fact it returns four, unless other WHERE conditions are set.

The SYS_CONNECT_BY_PATH() is usually invoked with a text column as an argument, because the resulting hierarchical trees are human-readable – but we could also specify a primary key column and obtain a series of GEO_PK rather than a series of NAME.

In any case it should be reminded that, despite its promising potential, unfortunately the SYS_CONNECT_BY_PATH() function can only be called in the SELECT list and in the ORDER BY clause, but not in the WHERE clause.

Summarizing, it's surely possible to create a hierarchical tree "on the fly", but to use it in a WHERE clause one should create a view as in the following example:

[1] For a more exhaustive coverage of hierarchical queries in Oracle, refer to https://docs.oracle.com/cd/B12037_01/server.101/b10759/queries003.htm#i2060615

```
CREATE VIEW GEO_TREE_VIEW AS
    SELECT GEO_PK, NAME, GEO_RANK_FK, BELONGS_IN, LEVEL,
           SYS_CONNECT_BY_PATH(NAME, '/') AS GEO_HIERARCHY
    FROM   GEONOMY
    WHERE connect_by_isleaf=1
    START WITH GEONOMY.GEO_PK = 1
    CONNECT BY PRIOR GEO_PK = BELONGS_IN;
```

As long as we are just interested in the full path to each leaf level, our where clause is implemented with one additional condition,

```
connect_by_isleaf=1
```

But, if we use it, our results will include only leaf nodes (and, as long as in the example Ferrara is the name of a province that has descendants, it would not be included: if present, the Town of Ferrara may be included only as long as it has no descendants...).

Now that it's available as a column in a view, the hierarchical tree can be queried, and in fact:

```
SELECT *
FROM     GEO_TREE_VIEW
WHERE    GEO_HIERARCHY LIKE '%Ferrara%'
    AND  GEO_RANK_FK > 40;
```

would return a list of all the "leaves" (=having no descendants) administrative subdivisions under the provincial rank, contained in the Province of Ferrara.
Such a list may, or may not, satisfy our needs. Furthermore, especially for big tables, the on-the-fly creation of hierarchical trees (or, to the same purpose, the refresh of views that contain hierarchical trees) may be computationally intensive an consequently slow.

An alternative approach

The alternative approach illustrated here is not particularly brilliant, nor puts into play any sophisticated method, but may help to manage RDBMS hierarchies. Most probably, I'm not the first to conceive this idea, that came to my mind around 1995. For sure, at the time I could not find any better alternative than developing the concept from scratch.
The approach described herein was applied to entirely different RDBMS:

- End-user X-Base databases managed via Visual Dbase applications
- Proper Enterprise RDBMS: Oracle 5 to Oracle 11

As explained in the initial notes, effectiveness requirements include the possibility for the RDBMS to create literal, case sensitive primary keys. The possibility to use a case-sensitive LIKE condition in the queries is also taken for granted.

It's absolutely clear that this method contradicts the 3rd Normal Form by implementing one or more child tables with data that – at the price of slower and much more complicated queries – could be accessed also in their original parent tables, but this is an exemplar case where "*denormalize until it works*", the second part of the old adage[2], is under the spotlight. We are exactly aiming at quicker, clearer and more effective queries for the hierarchical contexts.

The method can be described as follows:

- the columns of the hierarchical tables must include at least a primary key, a name/description of each entity, the "belongs in" column with the primary key of their "ancestor", a column for the compact hierarchical tree. The RANK column is desirable for filtering the query results further. Usually, there is only one "universal ancestor" at the top rank, representing the root of all the hierarchy;

[2] The full version is "Normalize until it hurts, denormalize until it works"

- provisions are made to generate literal, text-searchable (LIKE condition), primary keys made by a fixed number of characters - in our example, 3 (three) alphanumeric characters;
- any child table including a foreign key to a hierarchical table (including the hierarchical table itself[3]), also includes a column for the compact hierarchical tree;
- provisions are made to generate text-searchable (LIKE condition), compact hierarchical trees, that are created for each new record in the hierarchical table and saved in the new record;
- for systems (such as those considered in our example) whose hierarchies are based on a preset and fixed number of levels (ranks), the maximum length of a hierarchical tree is known in advance, and the column can be dimensioned consequently. Otherwise, it's wise to reserve an adequate space for a tree whose length will grow as soon as new hierarchical levels are added;
- considering its modest impact on table size and performance, the hierarchical trees (geographical, taxonomical...) are also added as columns in the non-hierarchical child tables that reference the hierarchical tables: that way, the full trees can be successfully explored by querying just one non-hierarchical table.

Back to our initial problem that was exemplified as:

«Select all the rows from the "Photographs" Table pertaining to butterflies in the family Lycaenidae from Emilia Romagna»

executing such a query boils down to:

- obtain the primary key of the Lycaenidae family, e.g. **A02**, from the "Taxonomy" table
- obtain the primary key of Italy, e.g. **2cG**, from the "Geonomy" table
- then,

```
SELECT * FROM PHOTOGRAPS WHERE TREE_TAX LIKE '%A02%' and TREE_LOC like '%2cG%'
```

The relevant values may also be obtained with subqueries, by prompting the user for the taxon and for the locality name. Here follows an example query from a real-world Oracle installation, that demonstrates the efficacy of the method for our desired purpose.

```
SELECT
    anagcoll.codice, anagcoll.destax,
    anagcoll.note, anagcoll.detsic,
    anagcoll.ultimamod, stato.dessta,
    riferim.desrif, localita.desloc,
    localita.intoloc, taxonomy.nometax
FROM anagcoll
     LEFT JOIN taxonomy ON anagcoll.codtax = taxonomy.codetax
     LEFT JOIN riferim ON anagcoll.codrif = riferim.codrif
     LEFT JOIN stato ON anagcoll.codsta = stato.codsta
     LEFT JOIN localita ON anagcoll.codloc = localita.codloc
WHERE
    anagcoll.treeloc LIKE (Select chr(37)||codloc||chr(37) from localita where
desloc='&LOCALITY_UPPERCASE') AND
    anagcoll.treetax LIKE (Select chr(37)||codetax||chr(37) from taxonomy where
nometax='&TAXON_UPPERCASE')
ORDER BY localita.desloc;
```

Query results, as expected, include:

[3] As explained above, the column BELONGS_IN references the primary key of another row in the hierarchical table

1	L457	EVERES ALCETAS	(null)	S	12-DIC-98	BUONA	NOVAK-SEVERA - LE FARFALLE	Castel d'Aiano e Dintorni	05u	ALCETAS
2	L303	EVERES ALCETAS	(null)	S	12-DIC-98	NORMALE	CHINERY - INSECTS	Castel d'Aiano e Dintorni	05u	ALCETAS
3	L615	CALLOPHRYS RUBI	(null)	S	11-GIU-01	NORMALE	CHINERY - INSECTS	Castel d'Aiano e Dintorni	05u	RUBI
4	L614	CALLOPHRYS RUBI	(null)	S	11-GIU-01	BUONA	CHINERY - INSECTS	Castel d'Aiano e Dintorni	05u	RUBI
5	L613	LYCAENIDAE GLAUCOPSYCHE	(null)	S	17-GIU-01	OTTIMA	NOVAK-SEVERA - LE FARFALLE	Castel d'Aiano e Dintorni	05u	GLAUCOPSYCHE
6	L586	PLEBEJUS ARGUS	(null)	S	18-NOV-00	OTTIMA	CHINERY - INSECTS	Castel d'Aiano e Dintorni	05u	ARGUS
7	L587	POLYOMMATUS ICARUS	(null)	S	18-NOV-00	OTTIMA	CHINERY - INSECTS	Castel d'Aiano e Dintorni	05u	ICARUS
8	L452	IOLANA IOLAS	(null)	S	12-DIC-98	OTTIMA	NOVAK-SEVERA - LE FARFALLE	Castel d'Aiano e Dintorni	05u	IOLAS
9	L532	LYCAENA PHLAEAS	(null)	S	06-NOV-99	OTTIMA	NOVAK-SEVERA - LE FARFALLE	Castel d'Aiano e Dintorni	05u	PHLAEAS
10	L275	POLYOMMATUS ICARUS	(null)	S	12-DIC-98	OTTIMA	RUFFO	CENTO	05S	ICARUS
11	L603	POLYOMMATUS ICARUS	(null)	S	10-SET-00	BUONA	NOVAK-SEVERA - LE FARFALLE	LAGO DI PRATIGNANO	05p	ICARUS
12	L602	POLYOMMATUS ICARUS	(null)	S	10-SET-00	BUONA	NOVAK-SEVERA - LE FARFALLE	LAGO DI PRATIGNANO	05p	ICARUS
13	L248	HEODES VIRGAUREAE	(null)	S	20-APR-98	BUONA	CHINERY - INSECTS	LAGO DI PRATIGNANO	05p	VIRGAUREAE
14	L245	CYANIRIS SEMIARGUS	(null)	S	12-DIC-98	BUONA	CHINERY - INSECTS	LAGO DI PRATIGNANO	05p	SEMIARGUS
15	L249	HEODES VIRGAUREAE	(null)	S	20-APR-98	NORMALE	CHINERY - INSECTS	LAGO DI PRATIGNANO	05p	VIRGAUREAE
16	L246	LYSANDRIA CORIDON	(null)	S	20-APR-98	NORMALE	CHINERY - INSECTS	LAGO DI PRATIGNANO	05p	CORIDON
17	L247	LYSANDRIA CORIDON	(null)	S	20-APR-98	SCARSA	CHINERY - INSECTS	LAGO DI PRATIGNANO	05p	CORIDON
18	L101	HEODES VIRGAUREAE	(null)	S	20-APR-98	BUONA	CHINERY - INSECTS	MADONNA DELL' ACERO	05p	VIRGAUREAE
19	L100	HEODES VIRGAUREAE	(null)	S	20-APR-98	BUONA	CHINERY - INSECTS	MADONNA DELL' ACERO	05p	VIRGAUREAE
20	L115	LYSANDRA CORIDON	(null)	S	12-DIC-98	NORMALE	CHINERY - INSECTS	MADONNA DELL' ACERO	05p	CORIDON
21	L114	LYSANDRA CORIDON	(null)	S	12-DIC-98	NORMALE	CHINERY - INSECTS	MADONNA DELL' ACERO	05p	CORIDON
22	L113	CELASTRINA ARGIOLUS	(null)	S	20-APR-98	NORMALE	CHINERY - INSECTS	MADONNA DELL' ACERO	05p	ARGIOLUS
23	L682	CELASTRINA ARGIOLUS	(null)	S	14-SET-03	OTTIMA	CHINERY - INSECTS	MADONNA DELL' ACERO	05p	ARGIOLUS
24	L681	CELASTRINA ARGIOLUS	(null)	S	14-SET-03	OTTIMA	CHINERY - INSECTS	MADONNA DELL' ACERO	05p	ARGIOLUS
25	L631	CELASTRINA ARGIOLUS	(null)	S	13-GEN-02	OTTIMA	CHINERY - INSECTS	MADONNA DELL' ACERO	05p	ARGIOLUS
26	L116	LYSANDRA CORIDON	(null)	S	12-DIC-98	SCARSA	CHINERY - INSECTS	MADONNA DELL' ACERO	05p	CORIDON
27	L230	HEODES VIRGAUREAE	(null)	S	20-APR-98	BUONA	CHINERY - INSECTS	MADONNA DELL' ACERO	05p	VIRGAUREAE
28	L231	HEODES VIRGAUREAE	(null)	S	20-APR-98	BUONA	CHINERY - INSECTS	MADONNA DELL' ACERO	05p	VIRGAUREAE
29	L241	EVERES ALCETAS	(null)	S	12-DIC-98	BUONA	CHINERY - INSECTS	MADONNA DELL' ACERO	05p	ALCETAS
30	L630	CELASTRINA ARGIOLUS	(null)	S	13-GEN-02	BUONA	CHINERY - INSECTS	MADONNA DELL' ACERO	05p	ARGIOLUS
31	L112	LAMPIDES BOETICUS	(null)	S	20-APR-98	NORMALE	HAUPT - INSECTES ETC.	MADONNA DELL' ACERO	05p	BOETICUS
32	L163	LEPTOTES PIRITHOUS	(null)	S	12-DIC-98	NORMALE	HAUPT - INSECTES ETC.	MADONNA DELL' ACERO	05p	PIRITHOUS
33	L111	LAMPIDES BOETICUS	(null)	S	20-APR-98	BUONA	HAUPT - INSECTES ETC.	MADONNA DELL' ACERO	05p	BOETICUS
34	L164	LEPTOTES PIRITHOUS	(null)	S	12-DIC-98	BUONA	HAUPT - INSECTES ETC.	MADONNA DELL' ACERO	05p	PIRITHOUS
35	L013	LYSANDRIA BELLARGUS	(null)	S	20-APR-98	NORMALE	RUFFO	MADONNA DELL' ACERO	05p	BELLARGUS
36	L014	LYSANDRIA BELLARGUS	(null)	S	20-APR-98	NORMALE	RUFFO	MADONNA DELL' ACERO	05p	BELLARGUS
37	L632	MELEAGERIA DAPHNIS	(null)	N	13-GEN-02	OTTIMA	NOVAK-SEVERA - LE FARFALLE	MADONNA DELL' ACERO	05p	DAPHNIS
38	L700	ARICIA AGESTIS	(null)	S	28-DIC-03	OTTIMA	NOVAK-SEVERA - LE FARFALLE	MADONNA DELL' ACERO	05p	AGESTIS
39	L701	CELASTRINA ARGIOLUS	(null)	S	28-DIC-03	OTTIMA	NOVAK-SEVERA - LE FARFALLE	MADONNA DELL' ACERO	05p	ARGIOLUS

Random example values of TREETAX (taxonomical tree) and TREELOC (geographical tree) are as follows:

TREETAX	TREELOC
000 00m 01w 03g 04B 04G 06e 06i	001 044 065 066
000 00m 01w 03g 04B 04G 06e 06i	001 044 065 066
000 00m 01w 03g 04B 04L 09i 09k 0At 0Bt 0CD	01R 02E 050 05M 05R 05p 04y
000 00m 01w 03g 04B 04L 09i 09k 0Aq 0Db 0Dk	01R 02E 050 05M 05R 05p 04y
000 00m 01w 03g 04B 04L 09i 09k 0Am	01R 02E 050 05M 05R 05p 04y
000 00m 01w 03g 04B 04L 09i 09k 0D0 0D1 0D2	01R 02E 050 05M 05R 05p 04y
000 00m 01w 03g 04B 04L 09i 09k 0At 0E4 0Ty	01R 02E 050 05M 05R 05p 04y
000 00m 01w 03g 04B 04L 09i 09k 0Ae 0DL 0DN	01R 02E 050 05M 05R 053
000 00m 01w 03g 04B 04L 09i 09k 0At 0By 0E3	01R 02E 050 05M 05R 053

After demonstrating that, thanks to the compact hierarchical trees, a query against a single table can return rows filtered by any hierarchical parent, including the rows referencing all its descendants, in the following pages I'll illustrate the simple steps needed to achieve that result.

Key-generating function

You can observe that in the compact tree every triplet of characters is separated from the adjacent ones by the blank character, that for obvious reasons is excluded from primary key generation. This grants independent findability of each primary key in each tree when searched by the LIKE clause with the generic pattern matching operator %. As stated above, the tree is generated whenever a new row is added to a hierarchical table, and copied in the child table as soon as a new row is added and a foreign key value is instantiated.

To grant a successful search by LIKE, specific characters (including the pattern matching ones) must not enter a primary key. This is one of the reasons why the generation of a new primary key in a hierarchical table is performed by a specific table-based function[4]. Surely, the same result could have been granted by the usual triggers and sequences, but in this case the function seems capable to grant more flexibility and can be invoked programmatically.

A generic example is provided in the following page. For each hierarchical table, as many tables as the number of characters in the key contain the last values (decimal ASCII table indexes) used for the creation of each byte in the key.

[4] The tactics adopted to exclude the undesirable ASCII codes from the key generation process are simple and may also be applied when using sequences. A function is exemplified more under.

```
-- create the three tables (for three-character keys)
CREATE TABLE FIRST_A (NUM INTEGER);
CREATE TABLE FIRST_B (NUM INTEGER);
CREATE TABLE FIRST_C (NUM INTEGER);

-- initialize the three tables (for three-character keys)
insert into FIRST_A values(48);
insert into FIRST_B values(48);
insert into FIRST_C values(47);

create or replace function NEW_KEY return CHAR IS NEW_TRIO char(3);
/******************************************************************************

NAME:       NEW_KEY

REVISIONS:
Version    Date       Author          Description
---------  ---------- --------------- ------------------------------------
0.1        Late 1990's Cesare Brizio  0. Conceived and used in Visual Dbase
1.0        17/03/2005  Cesare Brizio  1. Created
2.0        21/01/2023  Cesare Brizio  2. Added new parameters

PARAMETERS
-----------
INPUT: NONE

Returns: A primary key composed by three ASCII characters based on three TABLES,
                 FIRST_A, FIRST_B and FIRST_C.

DESCRIPTION
-----------
Other versions of the same function use SEQUENCES to the same purpose.
The initial version reproduced here allows an higher degree of control
on the process of skipping specific ASCII characters.
Every character in the key cycles in a sequence that includes:
      - the digits 0-9
      - the 26 alphabetical uppercase characters
      - the 26 alphabetical lowercase characters
All control characters or ambiguous characters are excluded from key generation.

This system can generate up to 62 x 62 x 62 = 238.328 different keys

Here follow some examples of the keys generated in consecutive calls up to
the final acceptable combination:

      000
      001
      002
      003
      ....
      009
      00A
      00B
      ....
```

```
    ....
    zzx
    zzy
    zzw
    zzz
*****************************************************************************/
    */

    CHAR_1 char(1);
    CHAR_2 char(1);
    CHAR_3 char(1);
    CHANGE_CHAR_2 char(1);
    CHANGE_CHAR_1 char(1);

    ---NEWTRIO char(3);

    VALCHAR_1 number;
    VALCHAR_2 number;
    VALCHAR_3 number;

    BEGIN

    CHANGE_CHAR_2 := 'N'; -- Unless proved otherwise, there is no need to change
    CHANGE_CHAR_1 := 'N'; -- the first nor the second character of the key

    --Let's check whether the least significant (rightmost) character
     --is at the end of its cycle.
    --In that case, I must increment the central character
    --and reset the rightmost to initial value

    select num into VALCHAR_3 from FIRST_C;
    select num into VALCHAR_2 from FIRST_B;

    if VALCHAR_3 = 122 then CHANGE_CHAR_2 := 'Y';

          --Furthermore, I check whether the middle character
          --is at the end of its cycle.
          --In that case, I must increment the leftmost character
          --and reset the center to initial value
          if VALCHAR_2 = 122 then CHANGE_CHAR_1 := 'Y';
          end if;
    end if;

    --Get the first (most significant, leftmost) element

    select num into VALCHAR_1 from FIRST_A;

    <<new3>>
    -- Increase the least significant (rightmost) element

    VALCHAR_3 := VALCHAR_3+1;

    update FIRST_C set num = VALCHAR_3;

    CHAR_3 := chr(VALCHAR_3);
```

```
    -- If CHAR_3 has an unadmissible value, I increase it again

    if (VALCHAR_3 between 58 and 64) or (VALCHAR_3 between 91 and 96) then goto new3;
    end if;

    -- IF I MUST CHANGE THE SECOND ELEMENT ...

    If CHANGE_CHAR_2 = 'Y' Then

        <<new2>>
        -- Increase the central element

        VALCHAR_2 := VALCHAR_2+1;

        update FIRST_B set num = VALCHAR_2;

        CHAR_2 := chr(VALCHAR_2);

        -- If CHAR_3 has an unadmissible value, I increase it again

        if (VALCHAR_2 between 58 and 64) or (VALCHAR_2 between 91 and 96) then goto
new2;
        else
        -- if the middle element is increased, the rightmost element starts from the
beginning
          update FIRST_C set num = 48;
          CHAR_3 := CHR(48);
        end if;
    else
        -- get the current value

        CHAR_2 := chr(VALCHAR_2);

    end if;


    -- IF THE FIRST (LEFTMOST, MOST SIGNIFICANT) ELEMENT ...
    -- DOES NOT NEED ANY CHANGE...
    -- Note: we do not address the case when, after 238000-plus cycles,
    -- the first element cannot increase anymore.
    -- There is no remedy!!!!
    -- Furthermore, such a condition can easily be verified by checking
    -- the value contained in FIRST_A before invoking this function.

    If CHANGE_CHAR_1 = 'Y' Then

        <<new1>>
        -- Increase the most significant element

        VALCHAR_1 := VALCHAR_1+1;

        update FIRST_A set num = VALCHAR_1;

        CHAR_1 := chr(VALCHAR_1);
```

```
            -- If CHAR_1 has an unadmissible value, I increase it again

            if (VALCHAR_1 between 58 and 64) or (VALCHAR_1 between 91 and 96) then goto
new1;
            else
            -- if the left element is increased, the middle element starts from the
beginning
               update FIRST_B set num = 48;
               CHAR_2 := CHR(48);
            end if;
      else
            -- Rilevo il valore corrente

            CHAR_1 := chr(VALCHAR_1);

      end if;

      NEW_TRIO := CHAR_1 || CHAR_2 || CHAR_3;

      Return NEW_TRIO;
End NEW_KEY;
```

The function can be easily tested, and is capable to provide the desired keys. First,

```
create table FIRST_TEST (key char(3));
```

then, repeatedly launch:

```
declare
x varchar2(3);
Begin
x := NEW_KEY();
insert into FIRST_TEST values (x);
end;
```

After a few cycles, the content of FIRST_TEST is easily checked:

```
select * from FIRST_TEST;
```

resulting in:

```
KEY
---
000
001
002
003
004
005
006
007
008
009

6 rows selected.
```

Building the tree

Once the machinery for key generation is in place, we need a way to encode the tree by recursively collecting ancestors of the current (new) row being added to the hierarchical table.
The function, available in the next page, is very compact and has many arguments, allowing it to be used with diverse table and column names.

```
create or replace FUNCTION BUILDTREE (PrimKey VARCHAR2,TableName VARCHAR2, KeyColName
VARCHAR2, ParColName VARCHAR2, RankColName VARCHAR2, TopRankPK VARCHAR2)
RETURN varchar2 AS TREE VARCHAR2(999);

/******************************************************************************

   NAME:       BUILDTREE

   REVISIONs:
   Version    Date        Author           Description
   ---------  ----------  ---------------  ------------------------------------
   1.0        17/03/2005  Cesare Brizio    1. Created
   2.0        21/01/2023  Cesare Brizio    2. Modified

PARAMETERS
-----------
INPUT:      the PRIMARY KEY of the entity whose hierarchical tree is needed
            the TABLE NAME of the hierarchical table
            the NAME OF THE KEY COLUMN of the hierarchical table
            the NAME OF THE PARENT ("BELONGS IN") column of the hierarchical table
            the NAME OF THE RANK COLUMN of the hierarchical table
            the PRIMARY KEY of the entity where the hierarchical tree is rooted, or
                            of the entity where we want the tree creation to stop

Returns: A string containing the compact hierarchical tree of the entity

DESCRIPTION
-----------
== IT SHOULD BE LAUNCHED IMMEDIATELY AFTER THE CREATION OF A NEW ROW IN THE
==    HIERARCHICAL TABLE!!!
== INITIALLY, THE NEW ROW IS CREATED WITH AN EMPTY VALUE IN THE COLUMN OF THE
==    COMPACT HIERARCHICAL TREE
== THIS FUNCTION POUPULATES THE VARIABLE CONTAINING THE TREE
== THE COLUMN OF THE NEW RECORD CONTAINING THE COMPACT HIERARCHICAL TREE
==   SHOULD BE IMMEDIATELY UPDATED WITH THE RESULT OF THIS FUNCTION

There are at least two ways to decide whether the creation of the tree is complete:
- stopping as soon as the highest rank (owned by only one entity in the table,
  the universal common ancestor that marks the root of the entire tree) is reached
- stopping as soon as the primary key of the common ancestor has been added to the
  tree
For the highest flexibility (including the generation of partial trees), this
version of the function accepts a primary key that identifies the ancestor
that closes the process of tree creation.
******************************************************************************/

CURR_ID varchar2(3);
FETCH_ID varchar2(3);
FETCH_RANK INTEGER;
FETCH_PARENT varchar2(3);
type MYCURSOR is ref cursor;
TREE_CURSOR MYCURSOR;
MY_STATEMENT Varchar2(200) ;

BEGIN
```

```
CURR_ID := PrimKey;

TREE := PrimKey;

LOOP

MY_STATEMENT := 'SELECT '||KeyColName||','||RankColName||','||ParColName||' from
'||TableName||' WHERE '||KeyColName||' = '||CHR(39)||CURR_ID||CHR(39);

 OPEN TREE_CURSOR FOR MY_STATEMENT;
  LOOP
      FETCH TREE_CURSOR INTO FETCH_ID, FETCH_RANK, FETCH_PARENT;
      EXIT WHEN TREE_CURSOR%NOTFOUND;
  END LOOP;
 CLOSE TREE_CURSOR;

 -- See the function explanatory header. We may hardcode here the
 -- rank of the topmost hierarchical level, or cycle until TopRankPK
 -- is reached
 -- if FETCH_RANK = 1 then
 --    exit;

 if FETCH_PARENT = TopRankPK then
      TREE := FETCH_PARENT||chr(32)||TREE;
      exit;
 else
      TREE := FETCH_PARENT||chr(32)||TREE;
      CURR_ID := FETCH_PARENT;
 end if;

END LOOP;

RETURN TREE;
END BUILDTREE;
```

The function may be tested anytime as in the following example, based on a real taxonomical table:

```
select BUILDTREE ('Oum','TAXONOMY', 'CODETAX', 'INTOTAX', 'RANKTAX', '000') from dual;
```

The example query above returns the following compact hierarchical tree:

```
 000 00m 01w 03g 04B 04L 09i 09k 0Aq 0Db 0um
```

Decoding the tree

The last missing piece is a quick way to decode the tree to provide a concatenated string including the name of all the taxa in the tree. This step is not needed to execute queries such as the one on page 4, yet it may be interesting to provide a full description of the hierarchical cascade, for clarity or completeness.

An example of a tree-decoding function is provided in the following page.

```
create or replace FUNCTION DECODETREE (CompTree VARCHAR2, TableName VARCHAR2, KeyColName
VARCHAR2, DesColName VARCHAR2, RankColName VARCHAR2)
RETURN varchar2 AS DECODED_TREE VARCHAR2(1000);
/******************************************************************************

   NAME:       DECODETREE

   REVISIONS:
   Version  Date        Author           Description
   ---------  ----------  ---------------  ------------------------------------
   1.0        17/03/2005  Cesare Brizio    1. Created
   2.0        21/01/2023  Cesare Brizio    2. Added new parameters

PARAMETERS
-----------
INPUT: the compact hierarchical tree to decode
             the TABLE NAME of the hierarchical table
             the NAME OF THE KEY COLUMN of the hierarchical table
             the NAME OF THE RANK COLUMN of the hierarchical table
             the NAME OF THE NAME/DESCRIPTION COLUMN of the hierarchical table

Returns: A string containing the fully decoded hierarchical tree of the entity

DESCRIPTION
-----------
This generic version accepts table and column names as parameters.
The function is hard-coded to accept 3-character long primary keys,
separated by a blank space.
The function assumes that the compact tree was built with the corresponding
"Buildtree()" function, that ensures the "3 characters - one blank" format for
all the tree entries except the last.
******************************************************************************/
I INTEGER;
CURR_ID varchar2(3);
SEPARATOR varchar2(3);
MYTREE varchar2(72);
FETCH_RANK INTEGER;
FETCH_DESC varchar2(30);
type MYCURSOR is ref cursor ;
DECODE_CURSOR MYCURSOR;
MY_STATEMENT Varchar2(200) ;

BEGIN

-- how many pieces does the compact tree contain?
-- by adding one character, it becomes a multiple of four characters
MYTREE := CompTree||CHR(32);
SEPARATOR := chr(32)||'|'||chr(32);
--SEPARATOR := ' : ';

LOOP

CURR_ID := substr(MYTREE,1,3);
```

```
MY_STATEMENT := 'SELECT '||DesColName||','||RankColName||' from '||TableName||' WHERE
'||KeyColName||' = '||CHR(39)||CURR_ID||CHR(39);

 OPEN DECODE_CURSOR FOR MY_STATEMENT;
  LOOP
      FETCH DECODE_CURSOR INTO FETCH_DESC, FETCH_RANK;
      EXIT WHEN DECODE_CURSOR%NOTFOUND;
  END LOOP;
 CLOSE DECODE_CURSOR;

-- Specific ranks may be formatted differently.
-- As an example, in case of a taxonomy where genera, subgenera, species
-- and subspecies have the ranks from 15 to 18, one may add the following
-- lines:
-- If FETCH_RANK = 15 then FETCH_DESC := INITCAP(FETCH_DESC);
-- end if;
-- If FETCH_RANK = 16 then FETCH_DESC := '('||INITCAP(FETCH_DESC)||')';
-- end if;
-- If FETCH_RANK in (17,18) then FETCH_DESC := lower(FETCH_DESC);
-- end if;


    DECODED_TREE := DECODED_TREE || SEPARATOR || FETCH_DESC;

    if length(MYTREE) < 5 then
      exit;
    Else
      MYTREE := substr(MYTREE,5);
    End If;
End loop;

RETURN DECODED_TREE;
END DECODETREE;
```

The function may be tested anytime as in the following example, based on the same taxonomical table as the previous example:

```
select DECODETREE ('000 00m 01w 03g 04B 04L 09i 09k 0Aq 0Db 0um','TAXONOMY', 'CODETAX',
'NOMETAX', 'RANKTAX') from dual;
```

The example query above returns the following decoded tree:

```
| ANIMALIA | ARTHROPODA | TRACHEATA | INSECTA | PTERYGOTA | COLEOPTERIA | COLEOPTERA |
POLYPHAGA | SCARABAEIDAE | TRICHIUS | ABDOMINALIS
```

Cesare Brizio, January 2023