Trees

Graphs and trees are ubiquitous data structures. They don't easily fit into Relational model; therefore querying them requires a little bit more ingenuity than routine select-project-join.

Compared to graphs, trees are relatively simple creatures. They are easy to draw. Almost any problem involving a tree structure is easy to solve. Algorithms on trees are generally fast. Edges, which are very important in graph definition, can be almost completely ignored for a tree. The tree structure could be encrypted in the nodes alone, and those tree encodings could be invented almost on daily basis.

In most of the chapter we'll focus on tree encodings. The rest is dedicated to smaller problems like node ordering by ad-hoc criteria. Several problems are postponed to the next chapter, where we study hierarchical aggregate queries and tree comparison. A reader who is primarily looking forward to developing some intuition with a vendor-specific hierarchical SQL extension (be that the `connect by`, or recursive `with` operator) is advised to proceed to the next chapter.

Materialized Path

Tree is a subclass of graph. We won't explore this idea in any depth in this chapter, however, because, once again, graphs are much more complex entities with their own set of problems. For all practical purposes a tree can be defined as a set of nodes arranged into a hierarchical structure via **tree encoding**. The purpose of tree encoding is to assign a special label to each node and manipulate tree nodes – i.e. query and update – by means of those labels. Informally, each node is equipped with a global positioning device that transmits the node's coordinates. Once each node's geographical position is known, we can answer typical queries like

Count all the employees who are located south of the "King", in other words, who report directly or indirectly to him.

Without a doubt you are already familiar with at least one such encoding: the UNIX directory structure[1]. Each file location in the hierarchy is defined by an *absolute pathname* -- a chain of directories that users have to navigate from the root to the leaves of hierarchy. For example, `/usr/bin/ls` is an absolute pathname. On the top of the directory structure there is a directory called `usr`, which contains a directory called `bin`, which contains a file called `ls`.

This seemingly straightforward idea can be applied to any tree structure. First, discover (or cook up) some unique key, which would distinguish a node's children. Then, list all the ancestor unique keys as the node's encoding. This list can be represented as a string (then, we have to agree upon string delimiter), or as a collection datatype. We'll refer to this encoding as **materialized path**. The adjective *materialized* emphasizes the fact that the path is stored. If the path is built dynamically, then the adjective is omitted and we refer to this dynamically generated encoding as just **path**[2].

Employee Name	Encoding
KING	1
JONES	1.1
SCOTT	1.1.1
ADAMS	1.1.1.1
FORD	1.1.2
SMITH	1.1.2.1
BLAKE	1.2
ALLEN	1.2.1
WARD	1.2.2
MARTIN	1.2.3

Table 5.1: Organization chart encoded with materialized path. We enumerated each node's children with integer numbers, and designated dot as a delimiter.

At this moment, we already have enough expressive power for basic queries:

1. An employee `JONES` and all his (indirect) subordinates:

```
select e1.ename from emp e1, emp e2
where e1.path like e2.path || '%'
and e2.ename = 'JONES'
```

[1] Ignoring symbolic links

[2] The reader undoubtedly noticed the parallel to view/materialized view terminology.

2. An employee `FORD` and chain of his supervisors:

```
select e1.ename from emp e1, emp e2
where e2.path like e1.path || '%'
and e2.ename = 'FORD'
```

Usually, query performance is unrelated to the form the query is written in SQL. In principle, a query optimizer has powerful techniques for transforming any query into an equivalent, better performing expression. Not in this case!

The first query is fine. Matching a string prefix is roughly equivalent to a range check

```
select e1.ename from emp e1, emp e2
where e1.path between e2.path and e2.path || chr(255)
and e2.ename = 'JONES'
```

where `chr(255)` is the last ASCII code. A reasonable execution strategy would be to find an (unique) employee record `e2` matching `ename='JONES'`, first. Finding a unique record is typically done via index lookup, in other words, extremely fast. The first query execution step establishes the range of paths, which the `e1.path` encoding has to fall into. If this range of paths doesn't contain too many paths, then the best way to find them is to iterate via index range scan. The more subordinates `JONES` has, the longer it would take to output them. In other words, the speed of this query is determined by size of the output – there is hardly a more efficient way to express this query.

The equivalent range check rewriting is valid for the second query as well

```
select e1.ename from emp e1, emp e2
where e2.path between e1.path and e1.path || chr(255)
and e2.ename = 'FORD'
```

Unlike the previous case, however, we know not the interval of paths, but the path `e2.path` itself, which we are going to match against all the intervals of the `e1` table. Certainly, there wouldn't be that many paths that match with `e2.path`, because the chain of ancestors in a balanced hierarchy is never too long. Yet, there is no obvious index that could leverage this idea. The condition of a point belonging to an interval consists of the two predicates `e2.path >= e1.path` and `e2.path <= e1.path || chr(255).` A normal B-Tree index on `e1.path` column could be leveraged while processing the first predicate only, and it would have to scan a half of the records on average[3].

[3] There are specialized indexing schemes – R-Tree, Interval tree, etc – that approach this problem. It remains to be seen if they could ever enjoy the same level of adoption as B-Tree and bitmapped indexes.

Finding a set of intervals covering a point

Querying ranges is asymmetric from performance perspective. It is easy to answer if a point falls inside some interval, but it is hard to index a set of intervals that contain a given point. Applied to nested sets we run into a difficulty answering queries about node's ancestors.

The critical observation is that a chain of ancestors is encoded in the node's materialized path encoding. We don't have to access the database in order to tell that the ancestors of the node `1.5.3.2` are nodes `1.5.3`, `1.5`, and `1`. A simple function could parse the materialized path. If this function is implemented on the client side, we can build a dynamic SQL query

```
select ename from emp
where path in ('1.5.3', '1.5', '1')
```

On the server side, the implementation could be little bit more sophisticated. The list of ancestors can be implemented as a temporary table built by a table function. This sketchy idea will be developed in greater detail in later sections, where we'll study much more elegant encodings than materialized path.

We conclude this section with the materialized path tree encoding schema design:

```
table TreeNodes (
   path varchar2(2000),
   …
)
```

This schema leaves the structure of `TreeNodes.path` column unspecified. Ideally, we could proceed and add some constraints, but, once again, much nicer development that doesn't require string parsing techniques awaits us ahead.

Nested Sets

Another approach to a tree structure is modeling it as nested sets.

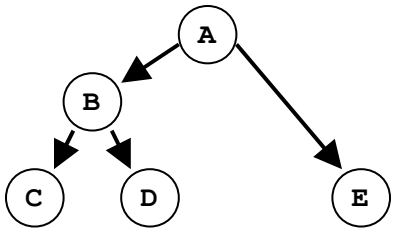


Figure 5.2a: A tree.

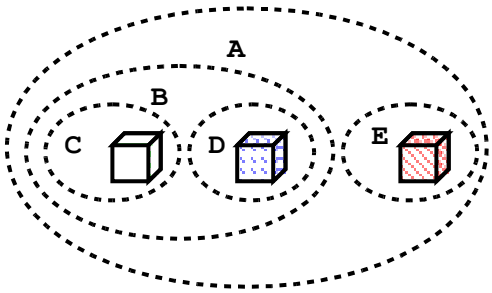


Figure 5.2b: Nested sets structure for the tree at fig. 5.2a. Set elements are boxes, and sets are the ovals including them. Every parent set contains its children sets.

Set containment could clearly accommodate any tree. Whenever we need to grow a tree by adding a new child, we just nest one more set into appropriate parent set.

Naive nested sets implementation would materialize a set of elements at each node. Aside from the fact that the RDBMS of your choice has to be capable operating with sets at the datatype level[4], this implementation would be quite inefficient. Every time a node is inserted into a tree, the chain of all the containing sets should be expanded to include (at least) one more element.

A more sophisticated variant of Nested Sets has been widely popularized by Joe Celko. The main idea behind this encoding is representing nested sets as intervals of integers (Fig. 5.3).

[4] High-end databases indeed have support for collection datatypes.

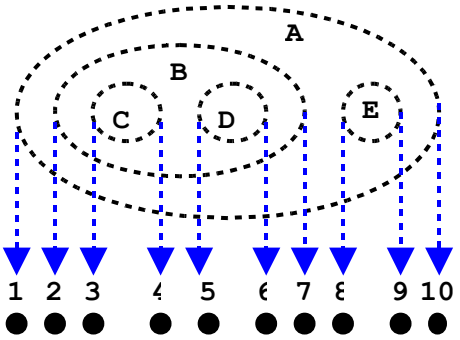


Figure 5.3: Nested Sets as intervals of integers. The node B is encoded by the interval beginning with 2 and ending with 7.

Unlike our first naïve nested sets implementation, where we must have a set datatype in order to be able to check if one set encoding contains another encoding, Celko's encoding no longer needs it.

The schema for nested sets tree encoding:

```
table EmpHierarchy (
   left integer,
   right integer,
   ename varchar(2000),
   …
)
```

A typical query checks if one interval is covered by the other interval, which, as we know already, can be easily expressed via standard SQL. For example,

```
select node1.ename
from EmpHierarchy node1, EmpHierarchy node2
where node1.left between node2.left and node2.right
and node2.ename = 'SMITH'
```

finds a chain of SMITHs supervisors. This query is essentially the same as querying the chain of ancestors in the section on materialized path encoding. This raises the concern about this query's performance. The dynamic SQL trick, which we employed for materialized path encoding, however, no longer works. There is no way to know the position of the node in the hierarchy by looking at the encoding of that node alone.

Like our naïve nested sets, intervals of integers encoding is **volatile**. Unlike naïve nested sets, inserting a new node involves a lot more work – roughly a half of the nodes must be recomputed.

The crux of the problem is that integers aren't dense. There is always a limit onto a number of intervals that you can nest inside any given interval of integers.

Fortunately, integers have nothing to do with interval nesting. They excel for illustration purposes, but if we want non-volatile encoding, we have to move on to dense domains of numbers, like rational or real numbers. With this goal in mind, let's study how to nest intervals of rational numbers. But first, we have to figure out how to divide an interval into smaller pieces.

Interval Halving

Consider splitting an interval with rational endpoints into two smaller intervals. Any point between the left and right endpoint might be good enough to the extent that we get two intervals. On the other hand, if we choose this point carelessly, then one interval might be much smaller than the other one. This might be a problem from implementation perspective, because small intervals impose much stricter requirements on arithmetic's precision. For example, checking if a point 0.7453 belongs to the interval [0.3, 0.9] is much easier than if it belongs to the interval [0.743, 0.748], since we need to go no further than 1 digit comparison in the first case versus 3 digits in the second. Therefore, we have to figure out the "most economical" way of finding the point between the two.

To repeat, given 2 rational numbers, what is the simplest number between them? Most people would probably choose the arithmetic average. For example, the simplest number between 0 and ½ is ¼, the simplest number between ¼ and ½ is 3/8 and so on. If we start with the point 0 and 1 and continue on halving the intervals iteratively then, what kind of numbers would be produced? Clearly, those whose denominator is a power of 2, or simply, dyadic fractions.

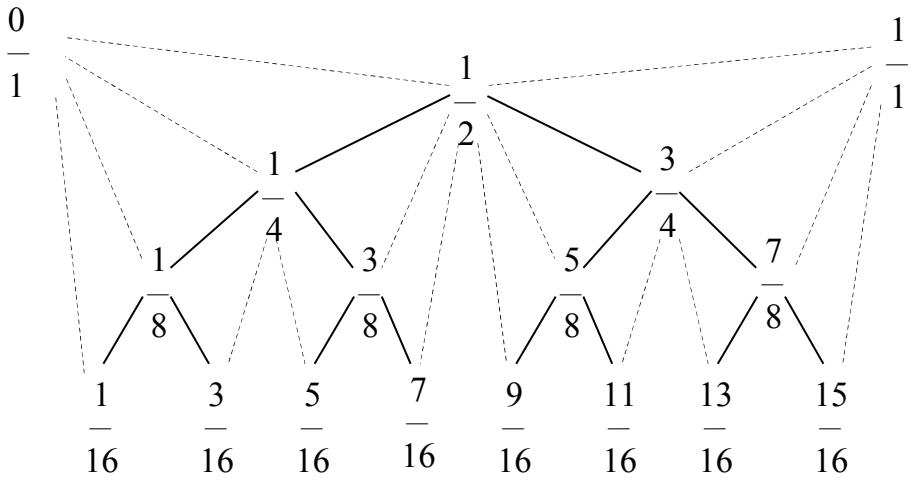
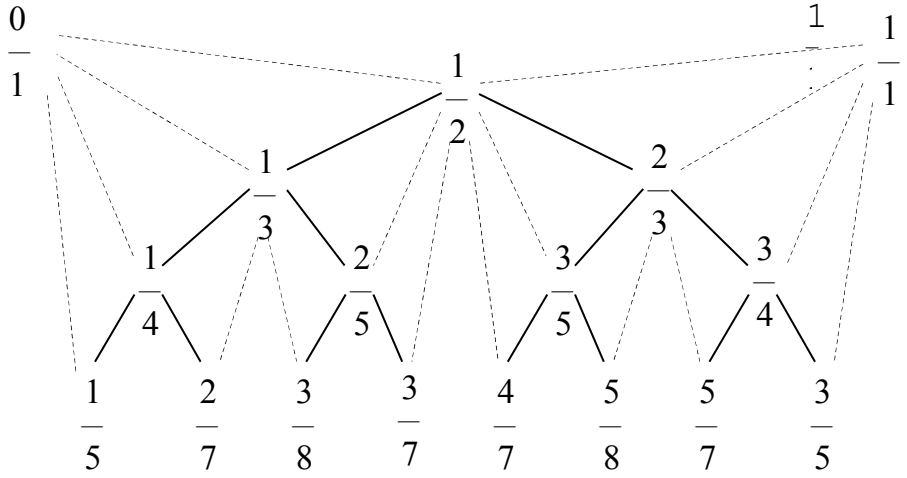


Figure 5.4: Halving interval [0,1] produces dyadic fractions that are naturally organized into a binary tree.

Elementary school students might beg to differ. When questioned what the sum of ½ and ¼ is some suggest that the result is ½ + ¼ = (1+1)/(2+4) = 2/6 = 1/3[5]. Ironically, their naïve approach is not without merit. The operation of adding fractions in this "wrong way" is called the *mediant*. The mediant is the simplest number between the two fractions if we use smallness of denominator as a measure of simplicity. Indeed, the average of ¼ and ½ has denominator 8, while the mediant has denominator 3.

If we start with the point 0 and 1 and continue on taking the mediant iteratively then, we produce another famous set of numbers – *Farey fractions*.



[5] In American educational system adding rational numbers correctly is a skill developed somewhere between middle school and college.

Figure 5.5: Dividing interval [0,1] by taking a mediant produces Farey fractions organized into *Stern-Brocot* tree.

The two systems of rational numbers that we just described are very closely related. In fact, there is a fascinating map between them, but the details are perhaps too advanced for a SQL programming book. An interested reader might Google *Minkovski question mark function*.

Although it is possible to develop nested interval tree encodings with both approaches, in this book we'll study Farey fractions only. The main reason is the encoding size. For the tree of dyadic fractions denominators multiply by 2 at each next level. For Stern-Brocot tree denominators grow slightly slower, approximately as powers of 1.618, where 1.618 is the golden ratio. In a word, Farey fractions is the most economical way to organize a system of nested intervals.

From Binary to N-ary Trees

The idea of interval splitting applies to binary trees. It could be easily extended to general n-ary trees by a well known one-to-one mapping of binary trees to n-ary trees. Given an n-ary tree we transform it into binary tree as follows: each parent node connected to the first child stays connected that way in the binary tree;ach next child is detached from its parent and is reconnected to its older sibling. Therefore, the second sibling has the first sibling as a parent in the binary tree; the third is connected to the second, and so on. The resulting tree is obviously binary, since each node has exactly two connections:

1. The left child as a former first child in the n-ary tree.
2. The right child as a former younger sibling in the n-ary tree.

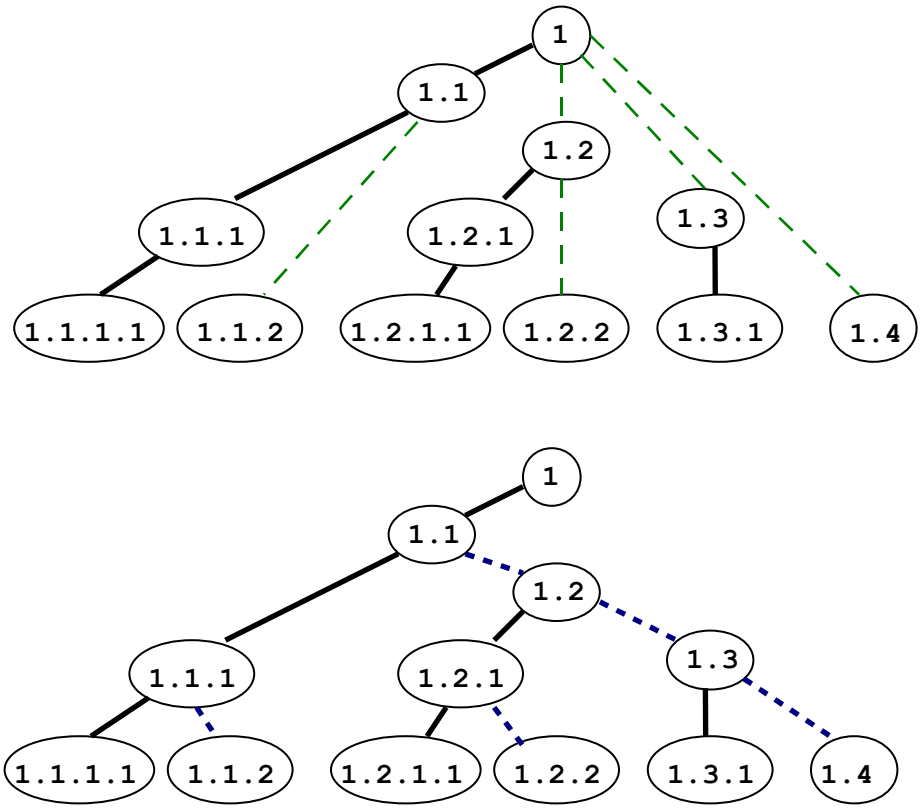


Figure 5.6: Mapping n-ary into binary tree could be viewed as a reorganization of edges between the tree nodes. A link between a parent and its first child remains unchanged, while a link from a younger child is transferred to the older sibling.

Let's transform the binary tree in fig. 5.5 into an n-ary tree. The root node has only one child. Therefore, it is convenient to agree that the root node of the binary tree in fig. 5.5 is 1/1 rather than ½. Then, node ½ is the first child, 2/3 is the second child, and so on. The first child of ½ is 1/3, the second child is 2/5, etc (Fig. 5.7).

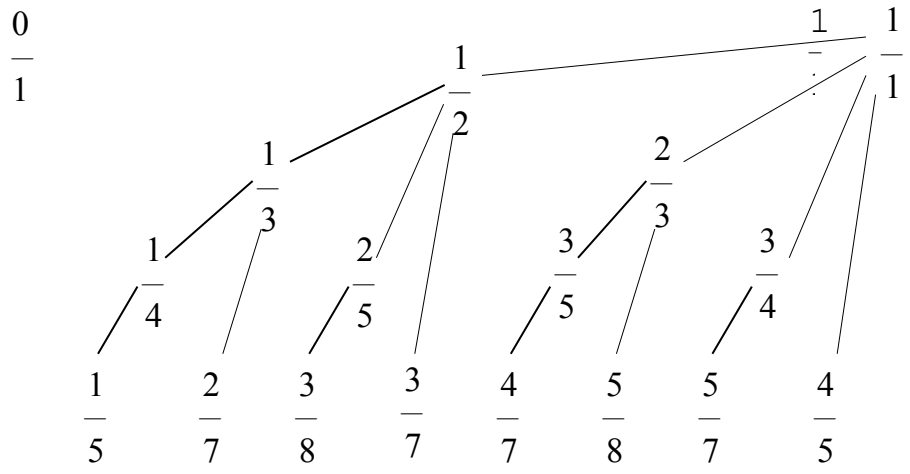


Figure 5.7: Stern-Brocot binary tree reorganized into n-ary tree.

Let's wrap these vague ideas into more rigorous form. In the next section we'll see that, Farey fractions are essentially *continued fractions*, which would leads to a simple algebra of 2×2 matrices. Matrices are required, because matrix multiplication mimics materialized path concatenation. Our development is essentially translating the algebra of materialized path strings into matrix form.

Matrix Encoding

Firstly a quick reminder of how to multiply 2×2 matrices of integer numbers.

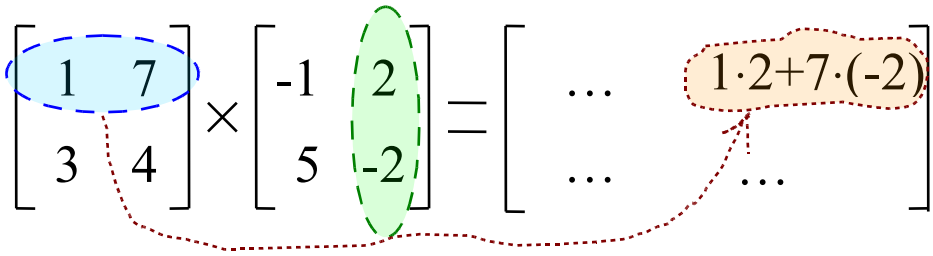


Figure 5.8: Multiplying 2×2 matrices. Each row in the left operand is matched element by element against a column in the right operand.

Matrix multiplication obeys the same rules as string concatenation. It is associative

$$(AB)C = A(BC)$$

and distributive

$$AB + AC = A(B+C)$$

but not commutative

$$AB \neq BA$$

Now, any materialized path is a concatenation of *atomic* materialized paths. For example, `.1.3.2.5` can be viewed as `.1` linked to `.3`, then joined with `.2`, and finally connected to `.5`. Can we do the same thing with matrices? The trick is to define atomic matrices, such that multiplying them would produce the matrix encoding for the full path.

Atomic matrices turned out to be quite simple. In fact every atomic matrix has three constant entries: 0 in the lower right corner, and 1 in the lower left, and -1 in the upper right. The upper left entry is the node's sequence number in the chain of siblings incremented by 1. For example, `.5` corresponds to the matrix

$$\begin{bmatrix} 6 & -1 \\ 1 & 0 \end{bmatrix}$$

Then, multiplying matrices corresponding to `.1`, `.3`, `.2`, and `.5` we get

$$\begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 4 & -1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 3 & -1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 6 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 107 & -19 \\ 62 & -11 \end{bmatrix}$$

Although we didn't seem to progress much so far, we can at least round up matrix tree encoding schema design

```
table MatrixTreeNodes (
   a11 integer,
   a12 integer,
   a21 integer,
   a22 integer
);
```

A lot of questions might emerge in the reader's mind at this moment, and we will address all of them one by one. The most important being: how is this matrix encoding related to nested intervals? Indeed, all these matrix manipulations should be evaluated from the querying perspective. In particular, how would we query a node's descendants and ancestors? An even more basic question: how do we find a node's parent and immediate children?

Parent and Children Queries

In our example

$$\begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 4 & -1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 3 & -1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 6 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 107 & -19 \\ 62 & -11 \end{bmatrix}$$

there are certain constraints that this node encoding obeys. The entries in the left column are positive, and the entries in the right column are negative. Also absolute values of the entries in the right column are component-wise smaller than left column. Likewise, absolute values in the upper row are greater than in the lower row. These properties are obvious for atomic matrices, but what about arbitrary nodes?

Consider an arbitrary node with encoding

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

Its n-th child encoding is calculated as a matrix product

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} n+1 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} a_{11}(n+1) + a_{12} & -a_{11} \\ a_{21}(n+1) + a_{22} & -a_{21} \end{bmatrix}$$

Let us examine these expressions closely. First we see that the parent left row entries are moved into the child right row with the sign changed. Let assume that our intuition about the entries in the left row being positive and the entries in the right row being negative is correct in the case of the parent encoding. Then, it must carry over to the child. By induction, it follows that any node will obey these rules. A similar line of reasoning proves our insight about absolute values.

The second very important constraint that matrix encoding satisfies is

$$a_{11}\, a_{22} - a_{12}\, a_{21} = 1$$

A reader with basic linear algebra background has likely recognized the matrix **determinant** here. Determinants obey a multiplication law: when matrices multiply, their determinants multiply as well. Therefore, the determinant of a node encoding matrix is a product of the atomic matrix determinants! Since all atomic matrices have determinant equal to 1, the determinant of any node encoding matrix must be 1.

The determinant constraint reduces the number of independent matrix entries to three. Given any three matrix elements, the fourth entry is unambiguously calculated from the determinant

constraint equation. We could do even better – reducing the number of independent elements to two.

Given the two elements a_{11} and a_{21}, relabeled conventionally as a and $-b$, plus unknowns a_{12} and a_{22} relabeled as y and x, the determinant equation reads

$$a\,x + b\,y = 1$$

This is perhaps the most celebrated equation in the elementary number theory. Its integer solutions are calculated via the extended **Euclidean** algorithm. Here is the algorithm illustrated on our familiar matrix encoding example

$$\begin{bmatrix} 107 & -19 \\ 62 & -11 \end{bmatrix}$$

We solve the equation

$$107\,x - 62\,y = 1$$

in a series of steps illustrated on fig 5.9.

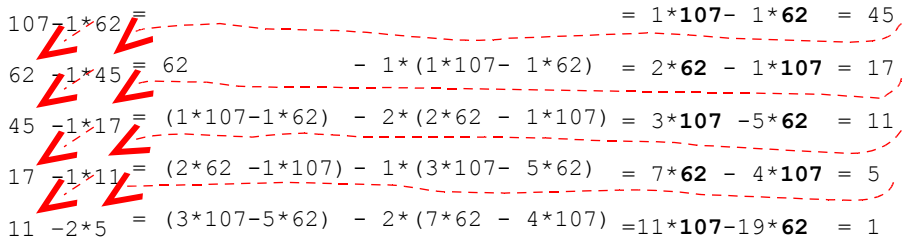


Figure 5.9: Extended Euclidean algorithm applied to numbers 107 and 62. We find an integer 1 such that 1*62 is no larger than 107, and then show that the largest common measure of 62 and 107 is the same as largest common measure of 62 and 107-1*62. Lather, rinse, repeat. In the third column we accumulate x and y factors .

At the last algorithm iteration we arrive at the values $x=11$ and $y=19$ that satisfy the equation.

Is this the only solution? Certainly not. Consider

$$62\cdot 107 - 107\cdot 62 = 0$$

Add it to

$$11\cdot 107 - 19\cdot 62 = 0$$

We have

$$(11+62)\cdot 107 - (19+107)\cdot 62 = 0$$

which implies another solution x=73, y=126! Fortunately, we know that x (i.e. a_{22}) and y (i.e. a_{12}) have to be smaller than 62 (i.e. a_{21}) and 107 (i.e. a_{11}), respectively. Therefore we can dismiss them.

The most important implication of our research in this section is that the combination of a_{11} with a_{21} is always **unique**. We can go as far as reducing the MatrixTreeNodes definition to these two attributes (and calculate the other two columns on the fly via the extended Euclidean algorithm), or leave the redundant attributes in the table and just declare the unique key. We choose the second alternative, which is justified by our next step. Knowing that a_{12} and a_{22} are always negative, we are going to store their absolute values. Then, as we have seen already, the child values a_{12} and a_{22} have to refer to some parent identified by a_{11} and a_{21}. In other words, a child always refers to its parent explicitly via a **foreign key** constraint.

Therefore, let's enhance our tree schema design

```
table MatrixTreeNodes (
   a11 integer,
   a12 integer,
   a21 integer,
   a22 integer
);

alter table MatrixTreeNodes
ADD CONSTRAINT uk_node UNIQUE (a11,a21)
ADD CONSTRAINT fk_adjacency FOREIGN KEY (a12,a22)
              REFERENCES MatrixTreeNodes (a11,a21);
```

Hierarchy design where a node refers to the parent name explicitly is called an *adjacency* tree model, and its scope is actually bigger than trees. The schema for adjacency model is the following

```
table AdjacentTreeNodes (
   id integer,
   parent_id integer
);

alter table AdjacentTreeNodes
ADD CONSTRAINT uk_node UNIQUE (id)
ADD CONSTRAINT fk_adjacency FOREIGN KEY (parent_id)
              REFERENCES AdjacentTreeNodes (id);
```

Unlike matrix encoding, there is no theory on how to choose a set of node identifiers, except obvious the restrictions that the id column is a unique identifier, and the parent_id always refers to the parent node. The general adjacency model is the main topic of the next chapter.

There is one subtle distinction between matrix and adjacency encodings. What does the root node refer to? In the adjacency encoding the root node parent_id has to be null, as there is no parent. In the matrix encoding we just apply the extended

Euclidean algorithm and obtain the four numbers. The root node refers to some nonexistent parent! What if we change the root node matrix encoding

$$\begin{bmatrix} 1 & -1 \\ 1 & 0 \end{bmatrix}$$

into

$$\begin{bmatrix} 1 & null \\ 1 & null \end{bmatrix}$$

Technically, we can't force `null`s into the matrix entries – it would destroy all the algorithms that were developed so far. It is more reasonable to admit that the formal referential constraint declaration for matrix encoding is invalid, and therefore, should be retracted from our schema design. This is not a big issue, however, given that matrix encoding enjoys a lot more sophisticated constraints than referential integrity.

Parent is NULL?

In the adjacency model the root node refers to the `NULL` parent. Does it mean that we can't answer the query "Find the root node's parent"? In the matrix model the root refers to the imaginary parent, and the query "Find the root node's parent" returns the empty set as it supposed to.

Once again, we were able to establish direct links between parent and children because we negated the values of a_{12} and a_{22}. From now on, we'll refer to the generic matrix node encoding as

$$\begin{bmatrix} a_{11} & -a_{12} \\ a_{21} & -a_{22} \end{bmatrix}$$

Now that we have informal referential integrity constraint, querying parent and children nodes becomes obvious.

Find all the employees who report directly to Jones.

```
select child.name
from MatrixTreeNodes parent, MatrixTreeNodes child
where parent.name = 'Jones'
and child.a11 = parent.a12 and child.a21 = parent.a22
```

Who is Jones' manager?

```
select parent.name
from MatrixTreeNodes parent, MatrixTreeNodes child
where child.name = 'Jones'
and child.a11 = parent.a12 and child.a21 = parent.a22
```

Nested Intervals

Querying descendants has to be done via nested intervals. Given a matrix

$$\begin{bmatrix} a_{11} & -a_{12} \\ a_{21} & -a_{22} \end{bmatrix}$$

we calculate the interval boundaries as

$$\frac{a_{11}}{a_{21}}, \frac{a_{11} - a_{12}}{a_{21} - a_{22}}$$

Which of these two numbers is the interval lower bound and which is the upper bound? Let's compare them. Multiplying both numbers to the product of their denominators and simplifying the result, we would reduce the problem to answering if

$$0 < a_{11}\, a_{22} - a_{12}\, a_{21}$$

Here is the determinant expression, again, which evaluates to 1. Hence, interval boundaries are ordered as

$$\frac{a_{11}}{a_{21}} < \frac{a_{11} - a_{12}}{a_{21} - a_{22}}$$

Next, how can we be sure those intervals are indeed nested? Let's compare an arbitrary node interval ends with that of its children. The nth child interval encoding is

$$\begin{bmatrix} a_{11} & -a_{12} \\ a_{21} & -a_{22} \end{bmatrix} \cdot \begin{bmatrix} n+1 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} a_{11}(n+1) - a_{12} & -a_{11} \\ a_{21}(n+1) - a_{22} & -a_{21} \end{bmatrix}$$

Therefore, the child interval boundaries are

$$\frac{a_{11}(n+1) - a_{12}}{a_{21}(n+1) - a_{22}}, \frac{a_{11}\, n - a_{12}}{a_{21}\, n - a_{22}}$$

Note, that the second endpoint is the same expression as the first one, with n decremented by 1. Therefore, we have to check if

$$\frac{a_{11}}{a_{21}} \le \frac{a_{11}\,n - a_{12}}{a_{21}\,n - a_{22}}, \frac{a_{11}\,n - a_{12}}{a_{21}\,n - a_{22}} \le \frac{a_{11} - a_{12}}{a_{21} - a_{22}}$$

for any $n \ge 1$. Both inequalities reduce to

$$0 \le a_{11}\,a_{22} - a_{12}\,a_{21}, 0 \le n\,(a_{11}\,a_{22} - a_{12}\,a_{21})$$

respectively. This proves that the parent node interval indeed contains its child intervals.

The second property of nested intervals – sibling node intervals being disjoint – can be proved in a similar fashion.

Descendants Query

Now that we have a nested intervals structure, we can move on to querying node's descendants. As a first approximation, let's take the descendants' query in terms of nested sets as a template and rewrite it in terms of nested intervals

```
select descendant.*
from MatrixTreeNodes descendant, MatrixTreeNodes node
where descendant.a11/descendant.a21 between node.a11/node.a21
                  and (node.a11-node.a12)/(node.a21-node.a22)
and node.name = …  -- predicate uniquely identifying a node
```

Unfortunately, this query would produce a wrong result. None of the database vendors supports a rational number datatype[6]. The ratios of integers would be silently cast into floating point numbers with accompanying errors due to lack of precision. We have to rewrite all of the expressions with divisions within the scope of integer arithmetic

```
select descendant.*
from MatrixTreeNodes descendant, MatrixTreeNodes node
where descendant.a11*node.a21 >= descendant.a21*node.a11
and   descendant.a11*node.a22 >= descendant.a21*node.a12
and node.name = …  -- predicate identifying a node uniquely
```

When we discussed descendant query performance in the context of nested sets, we emphasized index range scanning as an efficient way to extract all the descendant nodes. This idea generalizes to nested intervals, although we have to index interval boundaries. Let's enhance our tree encoding schema design with two function-based indexes:

```
table MatrixTreeNodes (
   a11 integer,
   a12 integer,
   a21 integer,
   a22 integer
);
```

[6] In fact, rational datatype is not even part of SQL standard.

```
CREATE INDEX idx_left ON MatrixTreeNodes(a11/a21);
CREATE INDEX idx_right ON MatrixTreeNodes((a11-a12)/(a21-a22));
```

We have to rewrite the query in such a way that optimizer can leverage these indexes

```
select descendant.*
from MatrixTreeNodes descendant, MatrixTreeNodes node
where descendant.a11*node.a21 >= descendant.a21*node.a11
and   descendant.a11*node.a22 >= descendant.a21*node.a12
and   descendant.a11/descendant.a21
      between node.a11/node.a21 - 0.0000001
      and (node.a11-node.a12)/(node.a21-node.a22) + 0.0000001
and node.name = …  -- predicate uniquely identifying a node
```

The constant `0.0000001` is designed to compensate for floating point arithmetic rounding errors. It essentially is a minimal supported mantissa. Please refer to your favorite database SQL manual in order to find out the exact value. This way an index range scan would capture all the nodes in the interval and, possibly, some extra[7]. Then, the (small) list of nodes is filtered with the exact condition.

Ancestor Criteria

Suppose we have two nodes: one encoded with matrix `A`, and the other encoded with `B`. Node `A` is an ancestor of `B` if and only if there is a (directed) path from `A` to `B`. In matrix terms, there has to be a sequence of atomic matrices, so that after we multiply `A` to it, we obtain matrix `B`. By the matrix multiplication associativity law, we can combine all those atomic matrices into a single matrix `X`. In other words, node `A` is an ancestor of `B` if there is a matrix `X` such that

$$A\,X = B$$

If matrix `A` has inverse A^{-1} then, multiplying both sides to A^{-1} we get

$$X = A^{-1}\,B$$

The formula for the inverse of the 2×2 matrix is

$$\begin{bmatrix} a_{11} & -a_{12} \\ a_{21} & -a_{22} \end{bmatrix}^{-1} = \begin{bmatrix} -a_{22} & a_{12} \\ -a_{21} & a_{11} \end{bmatrix}$$

where we leveraged the knowledge that our matrices always have determinant 1. Therefore, given any nodes `A` and `B` we can always find matrix `X` that encodes the path between them.

[7] But not too many extra!

This is absurd, as node `B` might not necessarily be a descendant of `A`! Let's examine the phenomenon more closely. As usual, an example might be handy. Consider the nodes `A=1.7` and `B=1.3.2.5` in matrix encoding

$$\begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 8 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 15 & -2 \\ 8 & -1 \end{bmatrix}$$

$$\begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 4 & -1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 3 & -1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 6 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 107 & -19 \\ 62 & -11 \end{bmatrix}$$

Then, $A^{-1}B$ evaluates to

$$\begin{bmatrix} 17 & -3 \\ 74 & -13 \end{bmatrix}$$

This is not a valid matrix encoding, however. It violates the constraint that the entries in the upper row are greater than the ones in the lower row.

Here is more detailed explanation why this is happening. Since matrix `A` is decomposed into a product of (atomic) matrices, why don't we leverage *the law of inverse of matrix product*:

$$(P\ Q)^{-1} = Q^{-1}\ P^{-1}$$

In our example

$$\left(\begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 8 & -1 \\ 1 & 0 \end{bmatrix} \right)^{-1} = \begin{bmatrix} 8 & -1 \\ 1 & 0 \end{bmatrix}^{-1} \begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix}^{-1}$$

Hence, $A^{-1}B$ expands into the following product of atomic matrices and their inverses

$$\begin{bmatrix} 8 & -1 \\ 1 & 0 \end{bmatrix}^{-1} \begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix}^{-1} \begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 4 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 3 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 6 & -1 \\ 1 & 0 \end{bmatrix}$$

where

$$\begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix}^{-1} \begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix}$$

collapses into the identity matrix. This is due to the fact that both `A=.1.7` and `B=.1.3.2.5` start with the same prefix `.1`. Therefore, the above expression for $A^{-1}B$ simplifies to

$$\begin{bmatrix} 8 & -1 \\ 1 & 0 \end{bmatrix}^{-1} \begin{bmatrix} 4 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 3 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 6 & -1 \\ 1 & 0 \end{bmatrix}$$

which can't be further reduced. It is multiplication by an atomic matrix inverse that violates the constraint.

In order to carry over this idea to SQL, we have to write $A^{-1}B$ in generic form

$$\begin{bmatrix} -a_{22} & a_{12} \\ -a_{21} & a_{11} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & -b_{12} \\ b_{21} & -b_{22} \end{bmatrix} = \begin{bmatrix} b_{21}\,a_{12} - b_{11}\,a_{22} & b_{12}\,a_{22} - b_{22}\,a_{12} \\ -b_{11}\,a_{21} + b_{21}\,a_{11} & -b_{22}\,a_{11} + b_{12}\,a_{21} \end{bmatrix}$$

which translates to the descendants query from the previous section

```
select B.*
from MatrixTreeNodes A, MatrixTreeNodes B
where B.a21*A.a12 - B.a11*A.a22 > -B.a11*A.a21 + B.a21*A.a11
and   B.a12*A.a22 - B.a22*A.a12 > -B.a22*A.a11 + B.a12*A.a21
and A.name = …  -- predicate identifying a node uniquely
```

Admittedly, this query is slightly more complicated than the nested intervals version. The real contribution of this section is introducing inverse matrices, which we will leverage later when relocating subtrees.

Ancestors Query

Logically, finding all the ancestors can be accomplished by swapping the roles of the two join operands in the descendants query. Once again, such a query won't be a good performer. In the section on materialized path encoding we split the problem into two parts: computing all the node encodings in the chain first, and extracting all the nodes by those keys from the database, second. As we have already seen a close tie between materialized path and matrix encoding, it would come as no surprise that we can perform the same trick with matrices.

Let's look into the matrix encoding of parent and child nodes one more time:

$$\begin{bmatrix} a_{11} & -a_{12} \\ a_{21} & -a_{22} \end{bmatrix} \cdot \begin{bmatrix} n+1 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} a_{11}\,(n+1) - a_{12} & -a_{11} \\ a_{21}\,(n+1) - a_{22} & -a_{21} \end{bmatrix}$$

As we have already seen, the child just inherited the entries a_{11} and a_{21} from its parent. Therefore, to calculate the left row entries of the parent, just take them from the right row of the child matrix. The right row elements are the remainders of the division of the parent left row by the right row. As an added bonus, we obtained a sibling order number n.

Let's demonstrate it on a familiar example of child node `1.3.2.5`

$$\begin{bmatrix} 107 & -19 \\ 62 & -11 \end{bmatrix}$$

The modulo function calculates remainders

$$19\text{-mod}(107,19) = 7$$

$$11\text{-mod}(62,11) = 4$$

Let's double-check the results

$$107 = (5+1)*19 - 7$$

$$62 = (5+1)*11 - 4$$

Hence, the parent encoding

$$\begin{bmatrix} 19 & -7 \\ 11 & -4 \end{bmatrix}$$

We continue this process and find the grandparent

$$\begin{bmatrix} 7 & -2 \\ 4 & -1 \end{bmatrix}$$

and great grandparent

$$\begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix}$$

which happens to be the root – a matrix with $a_{22} = 0$. Now that we have a list of ancestor matrices, how would we extract them from database? One solution would be to build a dynamic query like this:

```
select *
from MatrixTreeNodes
where a11=19 and a12=7 and a21=11 and a22=4
   or a11=7 and a12=2 and a21=4 and a22=1
   or a11=2 and a12=1 and a21=1 and a22=0
```

A better approach would be to store the ancestor encoding in a temporary `Ancestors` table, and use the generic query:

```
select n.*
from MatrixTreeNodes n, Ancestors a
where n.a11=a.a11 and n.a12=a.a12 and n.a21=a.a21 and n.a22=a.a22
```

Some RDBMS engines allow programming table functions, so that the table of ancestor encodings can be produced as an output of such a function. Syntactically, the query would become

```
select n.*
from MatrixTreeNodes n, Table(Ancestors(49,9,38,7)) a
where n.a11=a.a11 and n.a12=a.a12 and n.a21=a.a21 and n.a22=a.a22
```

Given the entries `a11=107`, `a12=19`, `a21=62`, and `a22=11`, the `Ancestors` table function is supposed to calculate the chain of ancestor encodings.

Converting Matrix to Path

In previous section we looked at the identity connecting the parent and child encoding

$$\begin{bmatrix} a_{11} & -a_{12} \\ a_{21} & -a_{22} \end{bmatrix} \cdot \begin{bmatrix} n+1 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} a_{11}(n+1) - a_{12} & -a_{11} \\ a_{21}(n+1) - a_{22} & -a_{21} \end{bmatrix}$$

and mentioned that a sibling order number `n` is a remainder of integer division `floor((a11*(n+1)-a12)/a12)`. In our example, the node `1.3.2.5` is the 5th children of the node `1.3.2`:

floor(107/19) = 5

floor(62/11) = 5

Working all the way to the root we'll find the other numbers in the path.

The path is generated in the order from leaf to root. Perhaps, it would be more convenient to generate it in the opposite order. The procedure is essentially the same, but applied to the transposed matrix.

Inserting Nodes

So far our attention was on queries. But how do we fill in the tree with nodes? The new node location is unambiguously defined by node's parent, and node's position among the other children. Normally, given a parent, one would like to attach a new node as the youngest child. Therefore, node's insertion is accomplished in two steps:

1. Query all the immediate children and find the oldest child.

```
Select max(floor(a11/a12)) as N from MatrixTreeNodes
where a11 = :parent_a12
and a21 = :parent_a22
```

where `:parent_a12` and `:parent_a22` are the host variables of your parent node encoding.

2. Insert the node at the `n`-th position:

```
insert into MatrixTreeNodes (a11,a12,a21,a22) values
(:parent_a11*(:N+1) - :parent_a12,
 :parent_a11,
 :parent_a21*(:N+1) - :parent_a22,
 :parent_a21);
```

These two steps can be combined into a single `insert as select` statement.

Relocating Tree Branches

This is the section where matrix algebra really shines. Consider a tree branch located at node encoded with matrix A, and suppose we want to move it to the new location under node B. How would the encoding of some node C (which is located in the tree branch under A) change?

That is quite an easy task for materialized path encoding. First, find out the path from the node A to C. Then, append it to the node B. This idea transfers to matrices almost literally. The encoding of the node C is a product of it's ancestor A and some other matrix

$$A\ X = C$$

Matrix X corresponds to the path from A to C. This path is appended to path B, hence is we multiply matrices and obtain the resulting encoding

$$B\ X$$

The unknown matrix X is calculated via inverse matrix, so we get the final answer

$$B\ A^{-1}\ C$$

In order to be able to translate it into SQL, let's expand the answer component-wise

$$\begin{bmatrix} b_{11} & -b_{12} \\ b_{21} & -b_{22} \end{bmatrix} \cdot \begin{bmatrix} -a_{22} & a_{12} \\ -a_{21} & a_{11} \end{bmatrix} \cdot \begin{bmatrix} c_{11} & -c_{12} \\ c_{21} & -c_{22} \end{bmatrix} =$$

$$[(-b_{11}\,a_{22} + b_{12}\,a_{21})\,c_{11} + (b_{11}\,a_{12} - b_{12}\,a_{11})\,c_{21}\ ,$$
$$-(-b_{11}\,a_{22} + b_{12}\,a_{21})\,c_{12} - (b_{11}\,a_{12} - b_{12}\,a_{11})\,c_{22}]$$

$$[(-b_{21}\,a_{22} + b_{22}\,a_{21})\,c_{11} + (b_{21}\,a_{12} - b_{22}\,a_{11})\,c_{21}\ ,$$
$$-(-b_{21}\,a_{22} + b_{22}\,a_{21})\,c_{12} - (b_{21}\,a_{12} - b_{22}\,a_{11})\,c_{22}]$$

which can be coded in SQL as

```
update MatrixTreeNodes c
set c.a11 = (:b12*:a21-:b11*:a22)*c.a11
          +(:b11*:a12-:b12*:a11)*c.a21
    c.a12 = (:b12*:a21-:b11*:a22)*c.a12
          +(:b11*:a12-:b12*:a11)*c.a22
    c.a21 = (:b22*:a21-:b21*:a22)*c.a11
          +(:b21*:a12-:b22*:a11)*c.a21
    c.a22 = (:b22*:a21-:b21*:a22)*c.a12
          +(:b21*:a12-:b22*:a11)*c.a22
where c.a11*:a21 >= c.a21*:a11  -- all the descendants of matrix
and   c.a11*:a22 >= c.a21*:a12  -- [[:a11,:a12][:a21,:a22]]
```

Ordering

So far we emphasized how to query the tree structure in various encodings. The presentation layer often demands an ordered output. The difficulty here is that the end user is expected to specify some ordering criteria at runtime, and the order might not conform to the order encoded in the tree structure. For example, consider a familiar hierarchy of employees with nested sets encoding

ENAME	LEFT	RIGHT
KING	1	20
JONES	2	7
SCOTT	3	6
ADAMS	4	5
FORD	8	11
SMITH	9	10
BLAKE	12	19
ALLEN	13	14
WARD	15	16
MARTIN	17	18

The tree is effectively ordered by the `LEFT` column. A user might ask to display it (locally) ordered by the employee names at each hierarchy level. It is easy to see that the (global) ordering criteria is essentially the path from the root to a node, which is made of concatenated names

ENAME	PATH
KING	KING
BLAKE	KING.BLAKE
ALLEN	KING.BLAKE.ALLEN
MARTIN	KING.BLAKE.MARTIN
WARD	KING.BLAKE.WARD
JONES	KING.JONES
FORD	KING.JONES.FORD
SMITH	KING.JONES.FORD.SMITH
SCOTT	KING.JONES.SCOTT
ADAMS	KING.JONES.SCOTT.ADAMS

Unlike the materialized path, the path of concatenated names must be generated dynamically, and the technical problem here is aggregating the names into a path string. Fortunately, we have learned the `LIST` aggregate function in chapter 3. Therefore, we proceed by taking the familiar nested sets query, which returns a list of all node ancestors, and aggregating those lists into paths

```
SELECT ii.left
       ,CONCAT_LIST(CAST( COLLECT('.'||i.ename) AS strings )) path
```

```
FROM   Employees i, Employees ii
where ii.left between i.left and i.right
group by ii.left;
```

On a cautionary note, this particular implementation of the LIST aggregate is agnostic of the order of the aggregation summands. The reader must double check that his favorite string concatenation method concatenates summands in the right order.

Ordering by dynamic path seems to be a natural solution until we are asked to order by numeric criteria, salary, for example. Numbers sorted as strings go in wrong order. They have to be padded. The other concern is negative values. All these inconveniences make the dynamic path solution not quite satisfactory.

An alternative solution is recreating hierarchy dynamically with the structure, which is conforming to the required ordering. At first thought, recreating hierarchy smells performance problems, but our rationale hinges upon a typical usage scenario. It must be the GUI that wants to display ordered hierarchy, and GUI rendering capabilities impose common sense limits on the tree size. Even if GUI were able to display the whole tree, then an end user would be overwhelmed by the volume of information that he were presented “at once”. It is fair to assume that a reasonably designed GUI would display only a **local** portion of a hierarchy, no matter how big the whole hierarchy might be.

Therefore, the issue of hierarchy ordering should be considered within the scope of application design. An application programmer designs a method of hierarchy navigation and display, which guarantees that the GUI displays only a small part of the hierarchy. A familiar example is an organizational chart where no more than two levels of hierarchy is displayed at each tree node: employee subordinates, and his supervisor. Ordering such a puny tree is a laughably easy exercise for a reader who has advanced thus far in the book.

An alternative design is a dynamic tree widget where each node exists in one of the two states: expanded or collapsed. The Windows file manager utility is a typical example. In principle, a directory tree could be expanded fully, but it is unreasonable to expect that there is a user who is up to the challenge of manually expanding a hierarchy of any significant size to the full depth.

Exotic Labeling Schemas

Tree encoding area is flourishing with various methods. It is so easy to invent yet another tree labeling schema! This contrasts to general graphs, which appear to defy encoding ideas. This section reviews others, arguably, less practical tree encoding methods.

Dietz Encoding

One natural way to label a tree is pre-order traversal (fig 5.10).

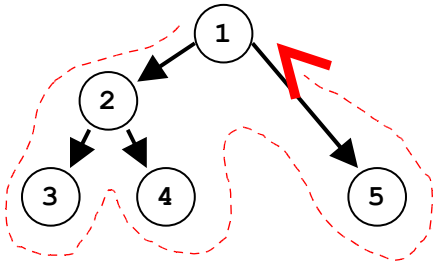


Figure 5.10: Pre-order traversal.

It is natural because the tree nodes are indexed in the depth-first order, and tree node records are is depth-first order in a nearly ubiquitous tree display with levels laid out horizontally

Employee Name	Preorder#
KING	1
JONES	2
SCOTT	3
ADAMS	4
FORD	5
SMITH	6
BLAKE	7
ALLEN	8
WARD	9
MARTIN	10

Post-order traversal is a little bit less intuitive way of navigating a tree. The nodes are visited in the same order, but node's index number assignment is postponed until all node's children are indexed.

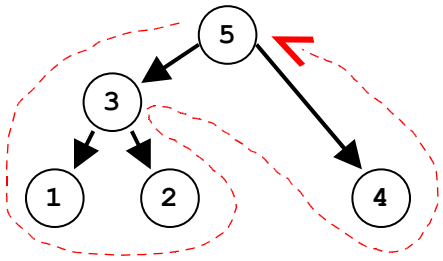


Figure 5.11: Post-order traversal.

Dietz tree encoding assigns a pair of indexes `(preorder#, postorder#)` to each node.

It is immediately evident that Dietz tree encoding is volatile. Inserting a new node disrupts existing encodings, both pre-order and post-order.

Querying a Dietz encoded tree is based upon the following ancestor criterion: node `x` is an ancestor of `y` if and only if `x.preorder# ≤ y.preorder#` and `y.postorder# ≤ x.postorder#`. This criterion appears to be identical to that of nested intervals, although unlike nested intervals neither `preorder# < postorder#,` nor `preorder# < postorder#` is universally true for all the nodes.

Let's look at two-dimensional picture of Dietz encoding. Let's assume that `preorder#` is the horizontal axis, and `postorder#` is the vertical one (fig 5.12).

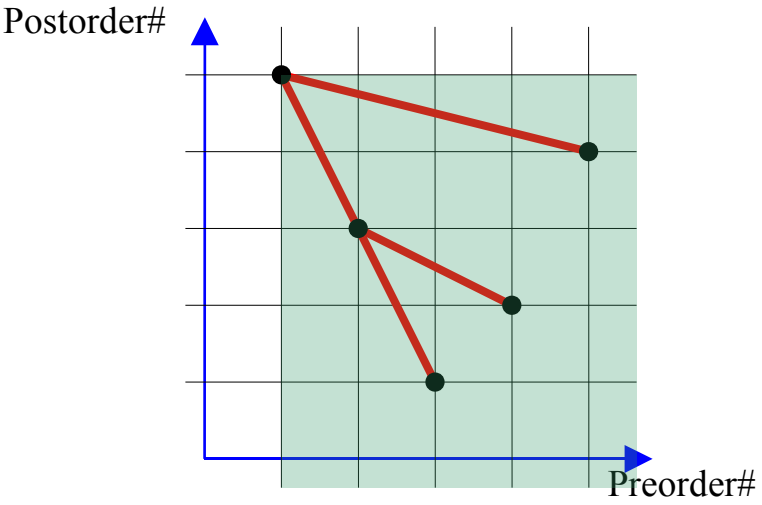


Figure 5.12: Two dimensional view of Dietz encoding. A root node preorder#=1, postorder#=5

has all it's descendants within the cone preorder#>1, postorder#<5.

Each node `x` has its descendant nodes `y` bounded within the two-dimensional cone defined by the two inequalities `x.preorder# ≤ y.preorder#` and `y.postorder# ≤ x.postorder#`. For nested intervals we would additionally have `preorder# < postorder#` or, in geometric terms have all the nodes above the diagonal `preorder# = postorder#`. If we move all the tree nodes above the diagonal somehow, then we'd succeed transforming Dietz encoding into nested intervals. Linear mapping

$$left = total\#nodes - postorder\# + 1$$

$$right = 2 \cdot total\#nodes - preorder\#$$

achieves that goal (fig 5.13).

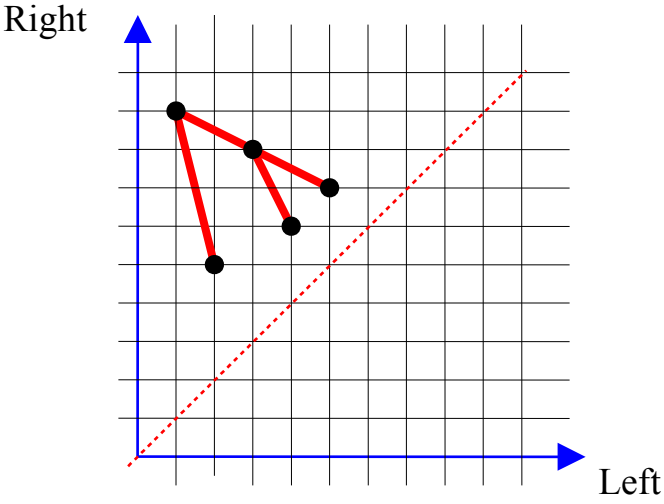


Figure 5.13: Dietz encoding linearly transformed to have all the tree nodes above the main diagonal – to nested interval encoding.

Pre-order – Depth Encoding

Yet another way to label the tree is storing combination of pre-order index number and level. Let's glance over the basic queries, though.

1. To find out node's parent, select all the nodes on the upper level and, then, filter out all the nodes beneath, and choose the one with maximum `preorder#` among the rest. In our familiar example

Employee Name	Preorder#
⊟ KING	1

⊟ JONES	2
⊟ SCOTT	3
⊟ ADAMS	4
⊟ FORD	5
⊟ SMITH	6
⊟ BLAKE	7
⊟ ALLEN	8
⊟ WARD	9
⊟ MARTIN	10

we search for `Smith`'s parent among the upper level nodes: `Scott`, `Ford`, `Allen`, `Ward` and `Martin`. We reject `Allen`, `Ward` and `Martin` as they all have `preorder#` greater than `Smith`. Between `Scott` and `Ford` the latter has greater `preorder#`.

2. To find out all the descendants we find the next node on the same level, and the next node on the parent level, choose the closer of the two and, then, select all the nodes between the given node and the chosen one.

For `Scott` we pick up `Ford` and `Blake`, and then select every node between `Scott` and `Ford` (exclusively).

3. Finding all of a node's children is just filtering out of case #2 the nodes with the proper level.

4. Finding a path to the root is just selecting all the predecessor nodes, grouping them by level, and extracting maximum sequence numbers per each group.

This is, again, is a volatile encoding, which kills our further interest in detailed exploration.

Reversed Nesting

Let's revise the nested sets example in fig. 5.2. This time instead of a parent set containing its children sets, we demand parent set to be contained in its children (fig 5.14).

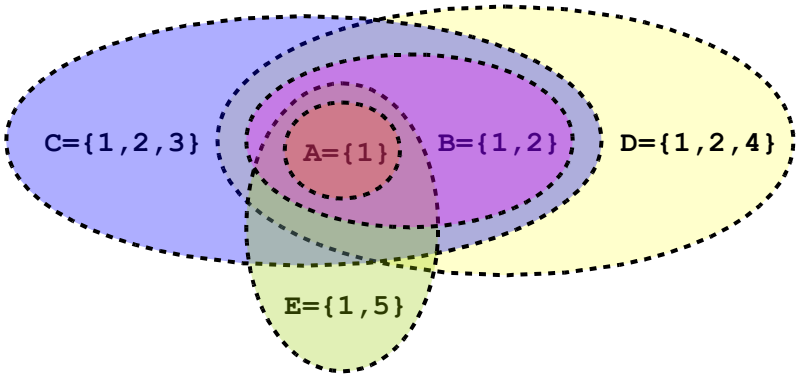


Figure 5.14: Nested sets structure for the tree in fig. 5.2a with set containment reversed. Now, a child set is required to contain its parent.

This variation of Nested Sets encoding is non-volatile. Once again, there is no direct support of a set datatype on most platforms, so we have to find a workaround. Armed with Boolean algebra we represent each set as a Boolean vector, the latter can be stored as plain strings. For example, `{0,1,4,6}` becomes `'1100101'`.

In general, set containment doesn't correspond to any standard operation on strings. Sets of tree nodes are special, however (fig 5.15).

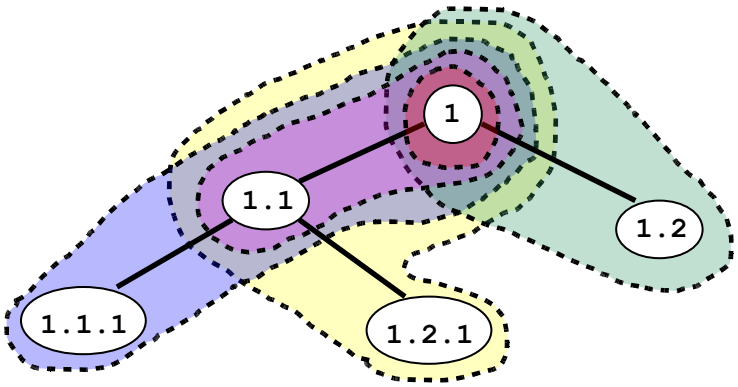


Figure 5.15: A different view of the reverse nested sets structure for the tree in fig. 5.14. A set is indistinguishable from path to the root.

Set elements correspond to tree nodes, and each set is associated with a path from the root to a node. Hence, set containment for reverse nested sets is the same as the substring operation. The

methods for querying materialized path encoded trees that we studied earlier must work for reverse nested sets as well.

Roji Thomas suggested yet another tree encoding, which is closely related to reverse nested sets. In his method each node is labeled in two steps. First, each node is designated a unique prime number. Then, each node is encoded with a number a product of the primes on the path from the node to the root. The node `A` is ancestor of the node `B` whenever encoding of `A` divides `B`.

By the **fundamental theorem of arithmetic**, every integer has a unique prime factorization:

$$N = 2^{\alpha_1} \cdot 3^{\alpha_2} \cdot 5^{\alpha_3} \cdot \ldots \cdot p_k^{\alpha_k}$$

It is immediate that the vector of prime orders $(\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_k)$ in Roji Thomas case is Boolean, and his encoding is essentially reverse nested sets. Without establishing this connection our investigation would not be complete.

When studying any encoding scheme, it is natural to start with the *expressive power*, verifying that any hierarchical query can be expressed in terms of new encoding. The second concern is efficiency, with emphasis on access path via index. It might be not obvious how to index tree nodes in Roji Thomas's encoding, and it is the connection to reverse nested sets that solves the problem.

Ordered Partitions

Given a positive integer `N`, in how many ways it can be expressed as a sum of smaller integers? Assume the order of the summands is important. Although this problem seems to be too distant from database practice, it nevertheless leads to yet another encoding method.

First of all, ordered integer partition is essentially materialized path. Let's enumerate all possible partitions by arranging them into the following table

Encoding	Partition	N
1	1	1
2	1+1	2
3	2	2
4	1+1+1	3
5	1+2	3

6	2+1	3
7	3	3
8	1+1+1+1	4
9	1+1+2	4
10	1+2+1	4
11	1+3	4
12	2+1+1	4
13	2+2	4
14	3+1	4
15	4	4
16	1+1+1+1+1	5

This enumeration is generated recursively. All the partitions of `N+1` are generated from the partitions of `N` in either way:

- putting an extra component with 1 in front of all of the other components of `N`, or
- incrementing the first component of each partition of `N` by `1`.

Then, it follows that partitions encoded with the even numbers have 1 as the last component. Indeed, as long as we have evenly encoded partitions of `N` to end up with 1, this property is carried over to the partitions of `N+1`. Since we associate partitions with materialized paths, this means the oldest child encoding is twice that of the parent!

For odd encodings, let's decrement them by `1`, first. This would produce an even encoding – the case that we have studied already. Compared to the former odd encoded partition, the latter even encoded partition has an extra component – the trailing `1`. Both are partitions of the same number `N`, however. This can be achieved only if the last component of the odd encoding is the sum of the last component and the last but one component of the even encoding. For example, the partition encoded as `11`, i.e. `1+3`, has the last component `3` to be represented as `2+1` in the partition number `10`, i.e. `1+2+1`. Then, by halving the even encoding, we'll get the encoding of the younger sibling (relative to the original odd encoded node)!

Although these discoveries allow defining an ancestor relation which could serve as a basis for hierarchical queries, we must not forget about performance. It is unclear if the integer partition based encoding schema admits an efficient querying node's descendants. Once again, our benchmark is nested intervals

encoding, where all the descendants can be accessed via index range scan. It turns out that integer partition encoding is very closely related to nested intervals encoding with dyadic rational numbers; in fact, there is a transparent mapping between them. I won't pursue this venue here any longer, however, and refer an interested reader to a series of my papers published by dbazine.com. Once more, from practical perspective Farey fractions provide more economical opportunity to organize a system of nested intervals.

Case Study: Homegrown C function call profiler

A profiler is a common tool for performance analysis. It determines how long certain parts of the program take to execute, how often they are executed, and generates the tree (or graph) of function calls. Typically this information is used to identify the portions of the program that take the longest to complete. Then, the time-consuming parts are expected to be optimized to run faster.

If the reader begins to suspect that there are plenty of tools doing the job, then (s)he's absolutely right. Those tools, however, are just programs. They build canned reports. We'll approach the problem from a database-centric angle. Let's place the profiling data into the database, and then we can query it any way we like!

First, we need to gather the data. Linux/Unix pstack is a primitive utility that does the job. We can query the calls stack repeatedly with a shell script like this

```
integer i=0
while ((i <= 999));
do
  pstack -F 25672 | tee -a pstack.trc;
  (( i = i + 1));
done
```

where the magic number 25672 is the process number to attach to.

The output is streamed into a file with the content like this:

```
25672:     oracleEMR920U3 (DESCRIPTION=(LOCAL=YES)
(ADDRESS=(PROTOCOL=beq)))
 009d5930 appdrv   (ffbe82d8, 180000, ffbe82d8, feb5e794, 0, ffbe8351)
 009fd85c kkofmx   (0, 0, 0, 0, fe7bebb4, 2a38a43e) + 624
 009f834c kkonxc   (fe7bebb4, fe4b3124, …) + 8c
 009fc65c kkotap   (200000, 0, 0, 0, 1, 4000) + 1444
 009f1154 kkojnp   (0, 0, 0, 0, 108, 0) + 14dc
 009ef9e8 kkocnp   (fe7bebb4, 1, 0, 0, 1f, 7c) + f0
…
```

```
 008f4b5c opidrv    (3c, 4, ffbef504, 2f6c6f67, 0, 2f) + 1ec
 001d666c sou2o     (ffbef514, 3c, 4, ffbef504, 0, 0) + 10
 001cf59c main      (2, ffbef5dc, ffbef5e8, 30d2000, 0, 0) + ec
 001cf488 _start    (0, 0, 0, 0, 0, 0) + 108
25672:     oracleEMR920U3 (DESCRIPTION=(LOCAL=YES)
(ADDRESS=(PROTOCOL=beq)))
 00a1c6a4 kkorbp    (ffbe8e3c, 0, ffffffff, 0, 0, ffbe8e61) + 808
 00a1f230 kkobrfak  (ffbe8e3c, ffbe8e60, fe39a42c, 0, 0, 1) + 8c
 00a1d92c kkofbp    (2a3bef38, 0, 2a3bef38, 0, 0, 1) + 584
 00a22ddc kkobmp    (10, fe4b39e4, 23, fe4df610, 0, fe4b3ac4) + bc
…
 008f4b5c opidrv    (3c, 4, ffbef504, 2f6c6f67, 0, 2f) + 1ec
 001d666c sou2o     (ffbef514, 3c, 4, ffbef504, 0, 0) + 10
 001cf59c main      (2, ffbef5dc, ffbef5e8, 30d2000, 0, 0) + ec
 001cf488 _start    (0, 0, 0, 0, 0, 0) + 108
25672:     oracleEMR920U3 (DESCRIPTION=(LOCAL=YES)
(ADDRESS=(PROTOCOL=beq)))
 00a1f218 kkobrf    (ffbe8e3c, ffbe8e60, 0, 0, 0, 0) + 74
 00a1d690 kkofbp    (2a3bef38, 0, 2a3bef38, 0, 0, 1) + 2e8
 00a22ddc kkobmp    (10, fe4b39e4, 23, fe4ff610, 0, fe4b3ac4) + bc
 009fc678 kkotap    (200000, 0, 0, 0, 1, 4000) + 1460
…
```

The detailed file structure is not important. What is essential for our analysis is the sequence of function calls (emphasized in bold) within each section (demarcated with italic delimiters). Specifically, we would like to move the data to the database with the following schema:

```
table function_sequences (
    stack_id integer,       -- section
    id       integer,       -- sequence #
    func     varchar2(100)  -- function name
);
```

Our sample data snippet is now a part of the database table

```
select * from function_sequences
```

STACK_ID	ID	FUNC
1	1	appdrv
1	2	kkofmx
1	3	kkonxc
1	4	kkotap
1	5	kkojnp
1	6	kkocnp
…	…	…
1	28	opidrv
1	29	sou2o
1	30	main
1	31	_start
2	1	kkorbp
2	2	kkobrfak
2	3	kkofbp
2	4	kkobmp
…	…	…
…	…	…
2	29	opidrv
2	30	sou2o
2	31	main

2	32	_start
3	1	kkobrfak
3	2	kkofbp
3	3	kkobmp
3	4	kkotap
...	...	...

The data mover implementation is rather boring. We read the input file line by line. If the line starts with the magic number 25672, then we are parsing a section delimiter string so we increment the `stack_id` counter, and initialize the `id` counter. Otherwise, we read the function name and increment the `id` counter.

It was easy to implement the function `id`s increasing with each line scanned, but their order is inconsistent with stack slot numbering. It is convenient to relabel the functions, so that the root function `_start` is always labeled with `id = 1`. The new view/table name – `Stacks` – reflects the fact that our data is now conventionally aligned with the stack data structure:

```
create table Stacks as
select s.stack_id id, height - s.id + 1 pos, func
   from function_sequences s, (
   select stack_id, max(id) height from function_sequences
   group by stack_id
) ss
where s.stack_id = ss.stack_id
;
```

We have finally arrived at an interesting aspect of the problem, the reporting. How do we combine all these stacks into a meaningful call graph? In particular, what defines the function location in the call graph? Having learned so much about materialized path encoding already, we'll find the answer hardly surprising. It is a path assembled from of all the names on the call stack that we are after.

Technically, functions are concatenated with the `list`[8] aggregate function

```
select id, list('.'||func)
           over (partition by id order by pos) path,
       pos, func
from stacks;
```

ID	PATH	FUNC
1	._start	_start
1	._start.main	main
1	._start.main.sou2o	sou2o

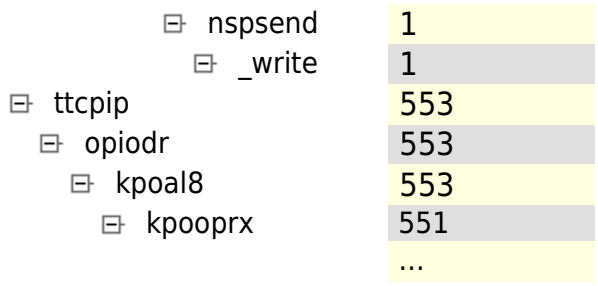
[8] Reminder: the `list` string concatenation function was defined in the section dedicated to user-defined aggregates of chapter 3.

1	._start.main.sou2o.opidrv	opidrv
1	...	kglobld
1	._start.main.sou2o.opidrv. ... .kkoqbc.kkooqb.kkocnp	kkocnp
1	._start.main.sou2o.opidrv. ... .kkoqbc.kkooqb.kkocnp.kkojnp	kkojnp
1	._start.main.sou2o.opidrv. ... .kkoqbc.kkooqb.kkocnp.kkojnp.kkotap	kkotap
1	._start.main.sou2o.opidrv. ... .kkoqbc.kkooqb.kkocnp.kkojnp.kkotap.kkonxc	kkonxc
1	._start.main.sou2o.opidrv. ... .kkoqbc.kkooqb.kkocnp.kkojnp.kkotap.kkonxc.kkofmx	kkofmx
1	._start.main.sou2o.opidrv. ... .kkoqbc.kkooqb.kkocnp.kkojnp.kkotap.kkonxc.kkofmx .appdrv	appdrv
2	._start	_start
2	._start.main	main
2	._start.main.sou2o	sou2o
2	._start.main.sou2o.opidrv	opidrv
2	...	opiodr

Now that we have a materialized path, it can be used to group the stack tree nodes with identical paths together, and/or order the nodes to get a nice indented tree layout

```
select func, pos-1 depth, count(1) from (
  select id, list('.'||func)
             over (partition by id order by pos) path, pos, func
  from
  stacks
) group by path, pos, func
order by path;
```

FUNC	COUNT(1)
⊟ _start	632
⊟ main	632
⊟ sou2o	632
⊟ opidrv	632
⊟ opiodr	632
⊟ opiino	632
⊟ opitsk	632
⊟ opikndf2	79
⊟ nioqrc	79
⊟ nsdo	78
⊟ nsrdr	78
⊟ nsprecv	78
⊟ sntpread	78
⊟ read	78
⊟ nsdosend	1
⊟ nsdo	1
⊟ nsdofls	1



The `count` aggregate is proportional to the time the execution has spent on this particular stack tree node. From there you would typically search for hotspot nodes, where a significant part of the execution time is spent.

Summary

- Nested Sets encoding is volatile, and not efficient for finding the chain of node's ancestors.
- The Stern-Brokot tree provides the most economical way to split intervals. This idea leads to matrix encoding.
- Matrix encoding combines the two models: adjacency relations and nested intervals. It is an especially appealing alternative to materialized path encoding.

Exercises

1. Prove that in matrix encoding sibling node intervals are disjoint.
2. It is very tempting to write the ancestors query like this

```
select *
from MatrixTreeNodes
where a11 IN (19,7,2)
  and a12 IN (7,2,1)
  and a21 IN (…,…,…)
  and a22 IN (…,…,…)
```

   Explain why this query is flawed.

3. Implement the `Ancestors` table function in an RDBMS of your choice.
4. Adapt the descendants query to return the list of node's immediate children, that is

Select all the node's descendants, which are on the next level.

Compare it to the query that leverages informal referential integrity constraint.

5. Combine the two matrix encoding node insertion steps into a single `insert as select` SQL statement.

6. Explore matrix encoding with atomic matrices of a kind

$$\begin{bmatrix} n & 1 \\ 1 & 0 \end{bmatrix}$$

7. The atomic matrices from exercise 8 are **symmetric** -- they remain unchanged under transposition. Prove that matrix transposition of an arbitrary node encoding corresponds to its inversed materialized path. (Then, symmetric matrix encoding corresponds to palindrome path!) Hint: *matrix transposition law*

$$(A\ B)^T = B^T\ A^T$$

8. Explore matrix encoding with atomic matrices of a kind

$$\begin{bmatrix} n+1 & i \\ i & 0 \end{bmatrix}$$

9. Prove that linear transformation of Dietz encoding indeed meets all the nested interval constraints.

10. Consider the chain of ancestors of the node `A`, and the chain of ancestors of the node `B`. Among their common ancestors there exists the oldest one, which is called the *nearest common ancestor*. Write the nearest common ancestor query in matrix tree encoding.