



Trees in SQL

SQL Tips by Burleson Consulting

Graphs and trees are ubiquitous data structures. They do not easily fit into a Relational model; therefore querying them requires a little bit more ingenuity than the 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 a daily basis.

Most of this chapter will focus on tree encodings. The rest is dedicated to smaller problems such as node ordering by ad-hoc criteria. Yet, several problems are postponed until the next chapter, wherein hierarchical aggregate queries and tree comparisons will be studied. A reader who is primarily looking to develop some intuition with vendor specific hierarchical SQL extensions (be that the *connect by*, or the recursive *with* operator) is advised to proceed to the next chapter.

Materialized Path

Tree is a subclass of graph. However, since graphs are more complex entities with their own set of problems, they will not be explored in depth in this chapter.

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 to 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, typical queries such as the following can be answered:

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: a UNIX directory structure. Each file location in the hierarchy is defined by an absolute pathname -- a chain of directories that a user has to navigate from root to the leaf of the 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 the node's children. Then, list all the ancestor unique keys as the node's encoding. This list can be represented as a string (if so, there must be agreement upon a string delimiter), or as a collection datatype. This encoding will be referred to 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 thus this dynamically generated encoding is referred to as simply *path*.

At this moment, there is enough expressive power for basic queries:

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'
```


An employee FORD and the 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 in which 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. The matching a string prefix is roughly equivalent to a range check where `chr(255)` is the last ASCII code:

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

A reasonable execution strategy would be finding the unique employee record *e2* matching *ename='JONES'*, first. Finding a unique record is typically done via an 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 does not contain too many paths, then the best way to find them is to iterate via the index range scan. The more subordinates JONES has, the longer it will take to output them. In other words, the speed of this query is determined by the 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, now not only is the interval of paths known, but also the path *e2.path* itself, which will be matched against all the intervals of the *e1* table. Certainly, there would not 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 the *e1.path* column could be leveraged while processing the first predicate only, and it would have to scan half of the records on average.

Finding a Set of Intervals Covering a Point

Querying ranges is asymmetric from a 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, there will be difficulty in answering queries about the node s ancestors.

The critical observation is that a chain of ancestors is encoded in the node s materialized path encoding. The database does not have to be accessed in order to tell that the ancestors of nodes 1.5.3.2 are nodes 1.5.3, 1.5, and 1. A simple function could parse the materialized path. This function s natural habitat  is the client side. There a dynamic SQL query is built:

```
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 more elegant encodings than the materialized path will be studied.

This section concludes with a materialized path tree encoding schema design:

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

This schema leaves the structure of the *TreeNodes.path* column unspecified. Ideally, some constraints could be added, but once again, a much nicer development that does not require string parsing techniques awaits us ahead.