A Novel Method for Representing Hierarchies in a Relational Database Using Bignums and SQLite

Stephen Huntley
stephen.huntley@alum.mit.edu

Abstract

I introduce a method of using a rapidly-converging infinite series to generate integer values which, when stored in relational database table rows, act as tags allowing each row to be interpreted and queried as a node in a hierarchy. To overcome integer precision limitations, I use Tcl 8.5's Bignum feature and tcllib's math::bigfloat package. I use SQLite's ability to store arbitrary binary data in its BLOB data type to manage overflow precision digits. The resulting code provides a fast and efficient way to store and query tree-structure data of theoretically unlimited size.

1. Introduction

It is natural for beginners as well as for experienced computer programmers to wish to organize and store information in the form of hierarchies, or tree structures. The filesystem on every modern computer is the most straightforward and ubiquitous example. Most users grasp and appreciate the utility of hierarchical file storage immediately.

Power users are also generally familiar with the frustration of trying to find particular files or file types in a directory structure, only to be faced with long waits as the computer grinds through a recursive search of the directory space.

The tree-structure data type is widely used for a variety of computer data-processing tasks beyond file storage. LDAP, OLAP, XML, 3D scene graphs, and network spanning trees are a few examples of technologies which organize data into hierarchies. The sizes of the datasets utilized by means of these technologies have typically grown enormously over the past several years, a trend consistent with datasets of nearly every type. What has not grown is the efficiency of algorithms used to query and retrieve information in these datasets. The approach still used in the overwhelming majority of cases is recursive search. Recursive search is a viable method for querying small to medium-sized datasets, but the technique does not scale, and performance of such searches on very large databases is becoming unacceptable even on the most advanced hardware.

This paper introduces a new method for parametrizing, storing and searching hierarchical information that eliminates the need for recursive approaches for the most

common search query types applied to trees. I also present details of a prototype executed with the help of advantageous features of Tcl 8.5 and the relational database extension TclSQLite.

Certain techniques described in this paper are covered by US patent #7,769,781, granted to the author.[1]

2. Hierarchies and Relational Databases

Although the method herein described is generally applicable to any linear or tabular data storage method, this presentation and the prototype focus on application to the problem of storing tree-structure data in a relational database.

The conundrum of storage and querying of tree-structure data in RDBMS programs has been a topic of persistent interest for many years.[2] The relational database is, generally speaking, the most powerful and flexible tool available to the mainstream programmer for dealing with large datasets. The presumed advantages of using a SQL-powered RDB package in this field have seemed self-evident for decades, but implementation issues have bedeviled almost everyone who has tried it for sizable datasets.

The obvious approach is simply to assign a unique number to each record in a database table that represents a node in the hierarchy, and define a “parent” field in the table schema to contain the unique number of the node linked one level up in the hierarchy from the node represented by the record. Thus finding a node's “children” is simply a matter of querying all records whose “parent” field contains the identifying number of the node of interest.

The problem comes when one wants to search all the linked nodes on the levels below a given node, an entire sub-tree. In that case, it's necessary first to retrieve all of a node's children, then all of those children's children, and so on recursively. A single complete search of this type might require thousands of individual read actions on the database, which is likely to take an unacceptably long time.

To get around this problem, the concept of “nested sets” was devised in the early nineteen-nineties. To use this method, each node is assigned an “entry” integer and and “exit” integer. The range of these numbers defines the set of integers lying between them. Every descendant of a given node is assigned entry and exit numbers which lie within the range of the given node's set. With these parameters defined for each node, querying all descendants of a node is then a simple matter of finding all nodes whose entry and exit numbers lie within the given node's defined range, which can be done with a single properly-crafted SQL statement.

This approach proves impractical, however, unless the tree structure is completely defined in advance and is expected not to change, or change very little; because adding a node to the hierarchy requires recalculating and rewriting some of the other nodes' entry and exit numbers. In a worst case most or all of these parameters may need to be rewritten, and the performance cost of so many write actions to a database is likely to be unacceptable.

One may hit upon the solution of using non-consecutive integers in entry and exit integer numbering; e.g., using multiples of five. There would then be room to add up to three more children to any given node before forcing the need for a recalculation and

rewrite. But this simply delays the reckoning.

Over the past two decades a number of proposals have been made to generalize the nested sets approach with more sophisticated means of generating entry and exit intervals, using complex parametrizing equations. None has proved workable or popular in practice for a number of reasons; including insufficient capacity to describe very large sets of nodes within available precision of integers storable in database table fields, and difficulty in expressing the necessary math in the form of SQL queries.

3. Solution Parameters

Existing solutions impose performance and/or capacity limitations on the size of hierarchies that can be stored. The bottlenecks they impose may have been considered manageable with the small to moderate-size datasets typical of the past, but they quickly become unacceptable when trying to deal with contemporary data processing challenges involving very large dataset sizes.

Hardware and supporting software limitations will always make it impossible to store or process hierarchies of perfectly unlimited size, but a near-optimal improved method would impose minimal additional bottlenecks. The performance of the method would thus be close to the performance limitations of the underlying RDBMS itself.

An improved method would impose minimal performance penalties on adding nodes to a tree structure that will inevitably grow and change in the course of real-world use. It would also preferably be relatively simple to implement and to design SQL queries that put it into action.

The solution proposed here, in addition to approaching the above goals, has the additional advantages of not requiring complex schemas or extra record-keeping tables, and of employing simple integer parameters that can be indexed in a straightforward fashion using well-known database management practices.

4. An Infinite Series for Generating Hierarchy Tags

To tag records in a database table as nodes in a hierarchy, I employ an infinite series specially crafted to converge very quickly; by assigning a term of the series in increasing order to each node descending down the tree, each branch of the hierarchy defines a unique partial series subset of the infinite series, and each node can be assigned a value representing the sum of terms of itself plus its ancestors in the partial series it belongs to.

Since the infinite series is designed to converge very quickly, the sums assigned to all nodes in a given branch of the tree can be guaranteed to fall between all the sums of nodes in adjacent branches. The quick convergence ensures that the limits of partial sums of the series can be strictly ordered according to the size of the first term of the partial sum; that is, the sum of a partial series will never overlap any value of another series whose first term's sum is greater, no matter how many terms are added to the initially lesser sum.

The greatest difficulty in designing this method was finding an infinite series that converged fast enough to guarantee non-overlap of values in adjacent partial series. At the same time the series needed not to converge so fast that the precision of the sum parameter was exhausted before a sizable tree

could be defined. In the end I could not find a suitable simple series with a standard linear-progressing index value.

Ultimately I had to design a double-indexed infinite series and use traits of the nodes themselves as indexes for the element function. That is, one of the indexes of the series is the depth level of the node in the hierarchy, and the other is the node's place in the count of its "siblings" (nodes with the same parent).

This approach ensures that available precision is doled out suitably depending on whether a child or a sibling is being added to a given node, always allowing for appropriate room for growth of the tree overall.

As far as I am aware, incorporating actual traits of the node as inputs into the interval-generating function is an innovation unique in the field.

The equation, expressed in standard form, is shown in Equation 1:

$$\sum_{m=0}^{\infty}\sum_{n=1}^{\infty}\begin{cases} m = 0 & 0 \\ m >= n & 0 \\ m < n & 1/2^{(\sqrt{(n-1)}*\sqrt{(3m-2)})} \end{cases}$$

Eq. 1

In Equation 1, the index m represents the node's level, and n represents the node's place in the sibling count. (More precisely, n is an "inheritance count," the first child of a node gets the node's n value plus one, so the count always increases as children and additional descendents are added to the tree.)

Since there is no general method for calculating the limit of convergence for an infinite series with transcendental terms in the element function, the conclusion that Equation 1 will always converge with sufficient speed is purely heuristic. Extensive testing has shown this always to be the case in practice.

When a node is added to the hierarchy, Equation 1 is used to calculate a term value for the node. Neither index value need be globally unique, so the term value may not be either. What is unique for the node is the sum of its term value together with the values of its ancestor nodes. It is this sum that is stored in the database record as a numerical tag uniquely descriptive of the node's place in the hierarchy.

A column of hierarchy tags so generated in a database table makes searching a sub-tree quite simple. An ancestor node's descendents are identified simply as nodes whose tag value is greater than the ancestor and less than the ancestor's nearest older sibling ("older" meaning having a smaller inheritance index number). The SQL query to accomplish this is simply a single-pass search for numeric values that fall within a defined range. No special joins, views or caches need be employed. This is just about the fastest kind of search a relational database can perform, and of course the column of tag values can be indexed for maximum speed.

Adding nodes to an already-established tree is straightforward as well. One simply needs to know the level of the parent to receive the new node as a child, and the inheritance number of the current youngest child of the parent. Equation 1 automatically produces a value which, when added to the parent's node sum, produces a new node sum that can be written directly to the database table and is guaranteed to conform to the existing

hierarchy scheme.

5. Prototype

In order to test the capabilities of this method, I developed a prototype program using Tcl 8.5 and the TclSQLite extension.[3] Tcl and SQLite were good complimentary choices to form a platform on which to build the prototype. SQLite is both easy to use and fast, and can handle very large datasets. SQLite also has the ability to store and process integer values of up to sixty-four bits in length -- that much available precision makes it possible for numbers generated by the method to describe very large sets of nodes. And given that calculation of numbers of such bit lengths made extra-precision mathematical calculations necessary, Tcl 8.5's new feature supporting native bigints in the core proved very useful, both directly for integer calculations and indirectly via its utilization in the tcllib::bigfloat package.

5.1 Implementation Example

Figure 1 illustrates a small sample hierarchy showing eight numbered nodes along with their level and inheritance number parameters in parentheses (m,n).

```
1. (0,1)
        2. (1,2)
                3. (2,3)
                4. (2,4)
                        5. (3,5)
        6. (1,3)
                7. (2,4)
                8. (2,5)
```

Fig. 1: Sample hierarchy

The process of preparing this hierarchy for storage in a SQLite database table starts with feeding each node's (m,n) parameters into Equation 1 to produce a term value to associate with the node. The term values clearly need not be unique.

```
t(1)= 0
t(2)= 0.5
t(3)= 0.1407857163281744654
t(4)= 0.0906152944101931834
t(5)= 0.0255328320537928796
t(6)= 0.3752142272464817736
t(7)= 0.0906152944101931834
t(8)= 0.0625
```

Fig. 2: Term values

Each node's term value is then added to the term values of its ancestors; e.g., node 5's value is added to the values of node 4 and node 2. The result is a unique numerical tag for each node which is unambiguously descriptive of its place in the hierarchy. For example, node 5 is known to be a child of node 4 because its node sum is greater than node 4's but less than node 3's. Because Equation 1 converges so rapidly, one could create unlimited descendents in this way for node 5, and those descendents' node sums would always be less than node 3's sum.

In order to take advantage of fast integer processing, the floating-point node sums are converted to integers by taking their fractional parts (with suitable precision-preserving zero-padding) and storing those in fields of a SQLite database table.

```
s(1)= 0
s(2)= 0.5
s(3)= 0.64078571632817447
s(4)= 0.59061529441019318
s(5)= 0.616148126463986
s(6)= 0.3752142272464817736
s(7)= 0.465829521656674957
s(8)= 0.43771422724648177
```

Fig. 3: Node sums

If then for example one wanted to retrieve all the descendents of node 6, one could use a simple SQL statement looking something like (sums truncated for clarity):

```
SELECT sum WHERE sum>3752
AND sum<5000
```

Clearly this query would return the sums associated with nodes 7 and 8, as desired.

6. Handling Node Distribution Limitations

With sixty-four bits of precision to work with, this method can easily be applied to hierarchies of tens of millions of nodes. It should be able to accommodate just about any data tree one is likely to come across in practice.

But the limited precision of integer storage in SQLite tables does impose some limitations in how nodes in a tree can be distributed. For example, no more than thirty-seven levels of depth can be described using this method before available precision runs out. In practice one is unlikely to encounter a tree with more than thirty-seven levels. But there may be pathological instances where this is the case. One would not wish to invest the time bringing this program into a real-world application only to find out in the midst of importing that ones dataset could not be accommodated. And what of the likely characteristics of the datasets of the next generation?

In order to eliminate inherent barriers to use of the prototype program for arbitrary hierarchies, I added a feature that makes it possible to encode and store any conceivable tree-structure dataset, up to the performance limitations of the database itself.

6.1 Overflow Precision Storage

SQLite has a BLOB (Binary Large Object) datatype which allows storage of arbitrary binary data. In order to accommodate trees of theoretically any size or node distribution, the prototype program adds a field to its table schema of the BLOB type, which is used to store extra precision digits in the form of binary data where necessary, without limitation as to length.

The Tcl code, when calculating the node sum for a new child, detects whether 64-bit precision has been exhausted by checking if the child's node sum is identical to the parent's. If this is the case, a global precision parameter is increased and the node sum is recalculated. The sum is divided into a part which can be stored using 64 bits, and a part containing all excess digits. The excess digits are converted into hexadecimal form as SQLite prefers them and are written into the BLOB-format field at the same time the integer part is stored in the integer sum field as described above.

Thenceforth, search queries which potentially require the extra precision to give complete results are done with a slightly more complex

SQL statement that incorporates comparison of the BLOB fields alongside integer value comparison of the sum fields. SQLite does comparisons of BLOB fields via binary byte-by-byte comparisons from the beginning of the field value to the end (analogous to Tcl's `[string compare]` command option). So precision of a calculated sum can be extended without limit by appending extra digits to a parent's overflow value stored in its BLOB field; and if use of overflow precision grows by multiple increments, binary values of varying lengths can be meaningfully compared just as varying length string comparisons are done.

I anticipate that in practice overflow precision storage will be rarely needed and employed chiefly in pathological situations, so impact on performance is expected to be minimal.

6.2 Separating Branch and Leaf Nodes

In the great majority of tree datasets, there will be many more leaf nodes (nodes with no children of their own, which terminate a branch) than branch nodes (which have one or more children). For example, in a hierarchy in which each node is assigned eight children up to a limit of a million nodes, only 62,500 branch nodes are required.

In practice, there is no reason to expend available precision and CPU resources calculating node sum values for leaf nodes. For querying purposes, leaf nodes can share the node sums of their parent branch nodes, as long as there is some established means of identifying the leaf nodes as such.

In the prototype program, a separate table is created for storage of leaf nodes solely. This table defines fields for a unique node ID, the parent node sum, and the parent overflow BLOB value in case it's necessary.

When a leaf is added to the tree, the node ID and parent sum information are written to a row in the leaf table. If a leaf node subsequently acquires a child of its own, Tcl code is first called to calculate a unique node sum of the leaf's own, and the node with its new sum is migrated to the branch table. Then the new child node is added to the leaf table complete with the reference to the newly-created branch's node sum value.

By granting unique node sums only to branch nodes, the capacity of the node sum-calculating method to describe and store large hierarchies is greatly increased. Splitting the total data into two tables also helps keep table sizes tractable, deterring the onset of any database-related maximum table-size capacity issues. It also helps SQLite maintain efficient caching and indexing states. I believe these advantages outweigh the performance penalty of requiring two separate queries on the database to ensure a complete search of a given sub-tree.

7. Performance

In truth it has been difficult to test the maximum capacity of the prototype program. It handles queries on databases containing in the tens of millions of nodes with little difficulty, even though minimal performance tuning has been done.

By way of comparison, probably the most widely-used tool for storing and querying large hierarchies is OpenLDAP, which in its most common implementations utilizes a Berkeley DB (BDB) backend for storage. Discussion in online forums suggests that the maximum capacity limit for practical operation of an OpenLDAP server with a

BDB backend (after extensive expert configuration tuning) is on the order of ten to fifteen million records.[4]

The chief performance difficulty is in initially populating the database from a large tree-structure dataset given for input. Calculating node sums and writing them to table rows can take hours for hierarchies containing millions of nodes. This of course would be impracticable in applications requiring close to real-time loading of data; such as, for example, reading and examining large XML files in an XML editor. In such cases the performance difficulties could be partially overcome by pre-calculating large template hierarchies with node sums already included. A suitable template hierarchy could be matched with an input dataset and imported with it, leaving custom calculations only for instances where the node distribution of the dataset of interest does not fit within the template exactly.

8. Future Developments

The prototype program was successful in demonstrating the basic validity of this novel method for encoding hierarchies, and in producing evidence that the limitations of the method are bounded chiefly by the inherent limitations of the underlying tools used to construct the program rather than by newly-introduced bottlenecks. Tcl and SQLite proved very useful in developing the prototype.

But it is to be expected that the next generation of computing challenges will present even larger datasets and more complex computing environments, and I believe that it is in meeting future challenges that this new method, and the particular advantages of Tcl and SQLite, will prove exceptionally valuable.

8.1 Parallelization

The prototype, despite its early state of development and minimal performance tuning, already performs well enough to handle very large hierarchical datasets which are typically handled only with difficulty by existing solutions. As the next generation of larger datasets arrive, I believe it will be possible to expand the capacity of the prototype greatly by introducing the ability to execute queries in parallel.

A strong advantage of the method described above is that partitioning a tree by node sum ranges without foreknowledge of the structure of the tree is a conceptually straightforward task. Thus SQL statements could be designed in advance to search sub-sections of the tree.

SQLite has no built-in client-server or parallel query-processing features. But it does make use of shared memory on operating systems that offer it for loading tables into RAM.. Thus multiple independent processes or threads that attempt to open a single database are all accessing a single in-memory set of tables.

With that feature in mind, SQLite's lack of multi-processing features can be well-compensated by Tcl's advanced event looping, socket networking and threading features. These features would be well put to use by expanding the prototype to include the ability to execute separate sub-queries in independent processes or threads, and collecting results via event loop polling.

As growing dataset sizes push the limits of a computer's ability to host a single database containing an entire hierarchy, the ability to

partition trees and index the partitions by node sum also makes the concept of calving off sub-trees into separate tables appealing. These tables could be moved to separate computers, thus efficiently sharding that database. Tcl's networking features could be used to distribute and collate queries and their results across a cluster.

The rapid development of multi-core processors and clustering technology in the commodity computer market suggest almost unlimited scalability in application of this method to hierarchical search.

8.2 Disconnected Hierarchies

Related to the ability to partition a tree into sub-trees is the ability of the method to add nodes to a parent without global information about the tree: only the traits of the parent node itself are required to calculate values for child node values (namely parent node sum, level and inheritance number). Thus if a sub-tree is moved to a separate computer, it can be updated and grown independently, without loss of ability to coordinate, or even re-merge, with the original tree. This ability distinguishes the method from most competing approaches for handling large-size hierarchies.

This feature is potentially useful for scaling and sharding databases for a single server application. But it also makes possible the concept of distributed filesystems or similar hierarchical information systems. In short, node sums calculated via this method could be used as universally valid hierarchical position identifiers (UHI:// ?).

Whereas in the Internet Protocol the concept of hierarchy is imposed arbitrarily on an undifferentiated 32-bit range of numbers, node sums used for network host identification would be meaningful within themselves, and thus potentially make tasks like routing as well as searching more efficient (at the cost of maximum node capacity in a given number space).

The version control system git is a conceptual example of a tool that organizes project files into hierarchies, and lets individuals check out subsections of the main project for disconnected development, with changes re-merged to the main project later. If one were to imagine a future iteration of the git concept which managed thousands or millions of entities in a project (rather than the now-typical few dozen files), assigning node sums to each entity would be a useful way to ensure consistent classification and search capabilities throughout the development cycle.

Various other tools for sharing information in discontiguous and dispersed usage patterns continue to appear and evolve into widespread use, from the old (e.g. Usenet) to the new (BitTorrent).

Advances in mobile computing and the spread of computer networks into the less-developed parts of the world have spurred interest in ad hoc and disconnected networking.

These and many other use cases could conceivably benefit from a globally valid yet locally editable hierarchy tagging protocol. The great diversity of environments and platforms encompassed by these use cases make the portability, compactness and power of the combination of Tcl and SQLite highly attractive for future development of applications which make use of this method.

References

[1] Huntley, S. "Method for labeling data stored in sequential data structures with parameters which describe position in a hierarchy." US Patent 7,769,781, issued August 3, 2010.

[2] A comprehensive treatment of the state of the art can be found in: Celko, J. *Joe Celko's Trees and Hierarchies in SQL for Smarties,* Morgan-Kaufmann. San Francisco, CA, USA, 2004.

[3] TclSQLite - http://www.sqlite.org/tclsqlite.html

[4] See for example: http://www.openldap.org/lists/openldapsoftware/200611/msg00051.html