doi:10.2498/cit.1002310

# Reusable Prime Number Labeling Scheme for Hierarchical Data Representation in Relational Databases

Serhiy Morozov[1], Hossein Saiedian[2] and Hanzhang Wang[1]

[1] University of Detroit Mercy, Detroit MI, USA
[2] University of Kansas, Lawrence KS, USA

Hierarchical data structures are important for many computing and information science disciplines including data mining, terrain modeling, and image analysis. There are many specialized hierarchical data management systems, but they are not always available. Alternatively, relational databases are far more common and offer superior reliability, scalability, and performance. However, relational databases cannot natively store and manage hierarchical data. Labeling schemes resolve this issue by labeling all nodes with alphanumeric strings that can be safely stored and retrieved from a database. One such scheme uses prime numbers for its labeling purposes, however the performance and space utilization of this method are not optimal. We propose a more efficient and compact version of this approach.

*Keywords:* hierarchical data structures, data labeling schemes, relational database systems, prime number labeling, performance evaluation

## 1. Background

Real-world information often consists of multiple pieces that can be grouped together. Usually, such an abstract relationship is modeled as a hierarchy, e.g., business organization, family ancestry, or military chain of command. The most common application of a hierarchical model is the file system on any modern operating system. It allows thousands of files to be neatly organized into appropriate folders, subfolders, etc. These hierarchical relationships provide an interesting insight into how information is organized. Therefore, a demand exists for data management systems that can store, retrieve, and search such data.

One way to supply this demand is to use a specialized hierarchical data management system. In fact, many of such systems successfully model complex trees and support a variety of query languages, e.g., XQuery, XPath, DOM, and SAX [19, 10]. Figure 1 shows an example XML document and the visualization of the hierarchy it represents. However, such specialized systems offer only hierarchical functionality and lack the maturity, scalability, and performance of relational database systems. Furthermore, relational databases are more popu-

```
<?xml version="1.0" encoding="UTF-8"?>
<a>
    <b>
        <c></c>
        <d></d>
    </b>
    <e></e>
    <f>
        <g></g>
    </f>
</a>
```

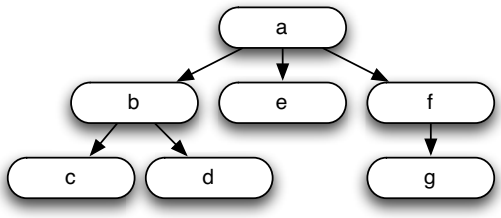


*Figure 1.* Example XML document.

lar, prevalent, and have a wider support base. Much of the existing data is stored in relational databases, so adding another data management system is often redundant.

A major distinction between hierarchical and relational data management is the way each method locates data. Hierarchical systems are best suited for gradual refinement of the search criteria or limiting the search to a specific category, subcategory, etc. Due to their advanced indexing ability, relational database systems excel at searches based on fixed criteria. However, no system can provide both kinds of functionality. In fact, this is why modern operating systems generate a file system index in addition to maintaining all files in a hierarchy. As an alternative, attempts have been made to add the hierarchical functionality to an existing relational database.

Many papers discuss the automation of creating a relational database schema to match the structure of a hierarchy. This provides the best utilization of resources, but it may not be applicable in cases where the structure of a tree is unknown [29, 11]. In fact, the limited acceptance of this approach has been due to the need to redesign the relational schema each time the hierarchy changes [26]. As a result, we focus on relational database hierarchical data management solutions that can accommodate trees of any shape.

### 1.1. Recursive Expansion Model

The recursive expansion model uses labels that point to each other to reveal the parent-child relationships of a tree. Each record contains a self-label and a label of a direct parent. It is the most natural way to store hierarchical data, especially for procedural programming language developers who are used to the concept of recursion. In fact, storing hierarchical data by shredding it into rows of a table is still a widely used technique [25, 7]. We refer to this model as Edge in our experiments.

The Edge model is compact because it uses consecutive integers as labels. Adding new and modifying existing nodes is simple as each node inherits the self-label of a direct parent as its parent-label. This allows quick sibling queries because all siblings share the same parent-label. However, reusing deleted labels is not common, so larger labels may be used unnecessarily in a frequently modified hierarchy.

Furthermore, the recursive nature of this model limits its applications. For instance, descendant searches in deep hierarchies require intermediate results to be stored in temporary tables for further recursive queries and multiple self-joins [27, 23]. Additionally, ancestor queries require a separate database request for each generation [21, 6]. Database communication is a major performance bottleneck, so a model that relies on multiple database requests to satisfy a single query is inefficient. Therefore, the Edge model is best suited for large and frequently changing trees.

### 1.2. Materialized Path Model

The materialized path model is designed specifically for ancestor determination queries. The ancestor information is encoded in the node's label, which eliminates the need to make additional database requests. The path from a node to the root is usually enumerated with numbers of specified length or encoded with delimited strings [24]. For instance, an absolute UNIX path `/var/log/error_log` shows that `error_log` file resides in log directory under var directory. We refer to this model as Path in our experiments.

The Path model performs sibling and descendant searches by comparing the common parts of node labels. It often uses pattern matching for descendant searches, which is slow and inefficient. Numerical, as opposed to character, labels are more effective because numeric comparisons are quicker and simpler than pattern matching. However, even with numeric labels, this model has difficulty distinguishing between direct children of a node and descendants of the same node because they all share a part of the label.

Path approach can quickly add new nodes to a tree, but doing so significantly increases label size. Each new node inherits the parent's path and appends its own label to it, thus making the label size grow linearly with the depth of the tree. This problem has been discussed in a

number of articles attempting to improve its performance and reduce label growth [18, 17, 15]. Furthermore, if a parent node changes its label, all descendants must be relabeled [9, 2]. Therefore, the applicability of this model is limited to shallow and static hierarchies.

### 1.3. Nested Set Model

The nested set model is designed specifically for descendant searches because node labels are chosen such that they can contain other labels. It is easy to determine the relationship between nodes by checking whether one label contains another [28, 16, 12]. For instance, nodes may be labeled with a number range expressed as two numbers. These numbers represent the beginning and the end of a range. The parent-child relationship is then determined by computing if one number range contains another. For example, the node with 3 : 8 range is considered a parent of all labels starting with 4 or more and ending with 7 or less. We refer to this model as Range in our experiments. The Range approach is the fastest way to do a descendant search, which is difficult for other schemes to accomplish. The performance advantage of this method is strictly in descendant searches, as it is computationally easy to locate all numbers within a range. However, there is no similar advantage in ancestor searches because it is difficult to search all the ranges that contain a specific number [5]. This disadvantage affects performance, but it is not the main reason why this model fails in many applications.

Frequently changing trees require a considerable number of label adjustments, as the changes stretch or shrink the number ranges. Adding a node to a densely populated tree requires adjusting all of its ancestors and possibly some of its siblings. Alternatively, failure to relabel a tree results in a label collision, when one label is supposed to represent two distinct nodes. A number of papers have addressed this issue through reusable labels, extra space for each label, and using numeric fractions/alphanumeric labels [30, 22, 14, 4]. Therefore, the applicability of this model is limited to deep and static hierarchies.

## 2. The Prime Number Model

Since classic approaches are limited in their applications, a lot of research has been directed toward creating a more universal labeling scheme. Wu et al. [3] introduce a new way to label a hierarchy with prime numbers. The prime number labeling scheme (PNL) labels each node with two numbers: a unique prime number called 'self-label' and another number called 'parent-label'. Each parent-label is divisible by all of its ancestors' self-labels because the label is a product of all ancestor self-labels. Adding a node to such a tree is simple. The self-label is assigned a value of any unused prime number and the parent-label is a product of this prime with a parent-label of the parent-node [3]. Figure 2 shows the prime number labeling table and the hierarchy it models.

This labeling scheme determines the relationship between two nodes by comparing their labels. If self-label of node X divides node Y's parent-label, then node X is considered to be a parent of node Y. Likewise, all nodes whose parent-labels are divisible by prime P are descendants of the node with P as a self-label. A modulo function can automate this process. It is not computationally intensive and can quickly operate on very large numbers.

The main disadvantage of the PNL scheme is that each prime number may only be used once. This helps establish the uniqueness of the labels,

```
<a>  self: 2   parent: 1
<b>  self: 3   parent: 2
<c>  self: 5   parent: 6
<d>  self: 7   parent: 6
<e>  self: 11  parent: 2
<f>  self: 13  parent: 2
<g>  self: 17  parent: 26
```

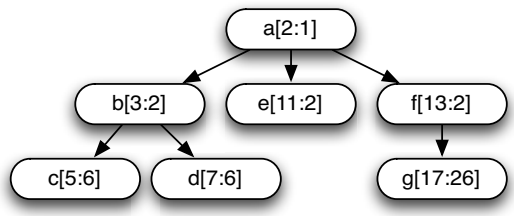


*Figure 2.* Prime number labeling scheme.

but also causes each subsequent parent-label to increase. This shortcoming is especially apparent in deep hierarchies, which require many prime numbers to be multiplied. Li et al. compared this scheme to two variants of the nested set model and a Dewey prefix scheme [13]. The PNL scheme required considerably more storage and had a much longer response time. Härder, et al. also performed benchmark experiments with PNL scheme modeling trees up to 37 levels deep and a maximum fanout of several millions [20]. The authors concluded that PNL scheme was not the most optimal solution for such complicated hierarchies. We propose a new version of this labeling scheme that solves its main drawback.

## 3. A Reusable Prime Number Labeling Scheme

The original prime number labeling scheme does not allow self-labels to be reused. This causes parent-labels to grow exponentially, which limits model capacity, increases model size, and reduces performance. In fact, label size limitations dictate the maximum possible depth and fanout of the model. This issue has been identified by the authors and confirmed by independent research [20, 13, 3]. A number of optimizations have been developed to improve the shortcomings of the original prime number labeling scheme. For instance, Preuveneers and Berbers recommended decreasing the label size by labeling each node with two different parent-labels that can be factorized into a single set of parent self-labels [8]. We propose a reusable prime number labeling scheme (rPNL) that natively recycles deleted labels and produces a more compact and responsive hierarchical model.

The proposed labeling scheme uses factorization to derive ancestor information. According to the Unique Factorization Theorem, every natural number greater than one can be written as a unique product of prime numbers. We use natural numbers as parent-labels and primes as self-labels. Given a node, factorization of its parent-label reveals the list of its ancestors. Decoding the rPNL label would take integer factorization, with an estimated time complexity between polynomial and exponential. However, local factorization of small numbers is considerably faster and more efficient than performing multiple database requests. Figure 3 shows the rPNL table as well as the hierarchy it models.

If the model reuses prime numbers throughout the hierarchy, repeating self-labels will appear. Therefore, our method records the order of the ancestors' self-labels using the simultaneous congruence (SC) number. The SC number encodes the order of primes such that $SC = i \mod p_i$. The Chinese Remainder Theorem states that there exists a number $n$ that satisfies $k$ simultaneous congruencies

$$\begin{aligned} n &= n_1 \mod m_1 \\ n &= n_2 \mod m_2 \\ &\dots \\ n &= n_k \mod m_k \end{aligned}$$

if and only if $n_i = n_j \mod gcd(m_i, m_j)$ for all $i$ and $j$. The solution $n$ is then congruent to the least common multiple of all $m_i$, $n = n \mod lcm(m_1, m_2, \dots m_k)$[1]. Because the self-labels are prime, the least common multiple is their product and their greatest common divisor is one. This means that a simultaneous congruence number is guaranteed to exist for any combination of primes.

```
<a>  self: 2 parent: 1  sc: 0
<b>  self: 3 parent: 2  sc: 0
<c>  self: 5 parent: 6  sc: 4
<d>  self: 7 parent: 6  sc: 4
<e>  self: 5 parent: 2  sc: 0
<f>  self: 7 parent: 2  sc: 0
<g>  self: 3 parent: 14 sc: 8
```

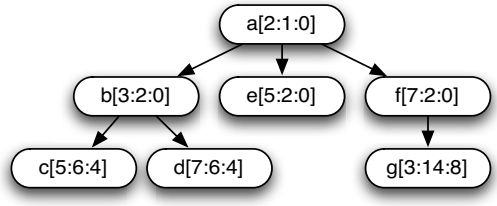


*Figure 3.* Reusable prime number labeling scheme.

[1] mod refers to the modulus operator, often represented as %; gcd stands for the greatest common divisor; lcm refers to the least common multiple.

There is an interesting pattern in rPNL node labels. The CS number of the child nodes is congruent to the parent's CS number modulo the parent's parent-label. For example, a node with a self-label=7 and ancestor path of 2.3.5 has a parent-label=30 and CS number=23. The child of this node, with a path 2.3.5.7 would have a parent-label=210 and CS number=53. It is evident that $53 \mod 30 = 23$, which is also the parent node's CS number. The proof of this pattern is simple. Assuming that $XC$ is the CS label of the child, $XP$ is the CS label of the parent, and $i$ is the generation of the node, the following must be true for all prime numbers on the parent's path, $XC \mod p_i = i = XP \mod p_i$ or $XC \mod p_i = XP$. Since the last formula holds true for all prime numbers on the parent's path, it must hold true for their product as well, $XC \mod \prod p_i = XP$. This property of rPNL labels may be used for both direct child and descendant node searches.

To avoid label collision, three rules must be enforced. First, only unique prime numbers may be allowed on each individual path, because it is impossible to have two simultaneous congruencies with the same modulo and different remainders. Second, only unique self-labels may be allowed among siblings, so we can uniquely identify them. Third, each self-label must be larger than the level at which it resides, because otherwise the SC number would have multiple meanings. For each node in a hierarchy, self-labels and parent-labels are assigned similarly to the original PNL scheme. However, according to the three rules, self-labels are not required to be globally unique primes. In fact, this scheme ensures that each self-label is the smallest possible prime that is bigger than its position on the path, unique within the given path, and unique among the siblings (see Appendix 1 listing for `get_next_rpnl_prime`). These rules reduce the label size growth that is so problematic with the PNL scheme. They also force deleted labels to be automatically reused, which makes this labeling scheme more efficient.

## 4. PNL and rPNL Label Size

The major advantage of the proposed scheme is that it may reuse labels. In fact, the number of possible reusable labels is close to $n!$, where n is hierarchy depth. Because a different prime is used for a node at each level, there will be n! possible combinations that result in the same parent-label. The actual number will be slightly less because certain small primes may not be used at the level that is greater than the prime itself.

A typical relational database can store integers up to 64 bits long and record numbers as big as 1.84E+19. This influences the number of generations any branch may have. The biggest primorial (n#), a product of all prime numbers less than or equal to n, that fits into the allocated space is 47#=6.15E+17. This means that there are 15 primes that may be used to describe the longest branch. Table 1 outlines the availability of the 15 positions relative to each prime.

Assuming that the prime position must be less than the prime itself, there are nine primes that are greater than 15. They may be organized in any order, which results in $9! = 362,880$ combinations. The first six primes are smaller, so some positions may be unavailable. There are $1*1*2*3*6*7*9! = 91,445,760$ total possible combinations. The difference between the PNL and rPNL schemes is that the PNL scheme may have only one 15 level path. It also must be the first path, assuming depth-first approach. The rPNL scheme may have over 91 million of such paths. Figures 4 and 5 shows the label growth of both schemes. The proposed scheme produces smaller labels and is less affected by the depth and fanout of the tree.

Table 2 shows the total label size of the different labeling schemes for the running example. The example tree contains only seven nodes so

| Prime | 2 | 3 | 5 | 7 | 11 | 13 | 17 | 19 | 23 | 29 | 31 | 37 | 41 | 43 | 47 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Possible Positions | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| Unsuitable Positions | 14 | 13 | 11 | 9 | 5 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Available Positions | 1 | 1 | 2 | 3 | 6 | 7 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

*Table 1.* rPNL self-label availability.

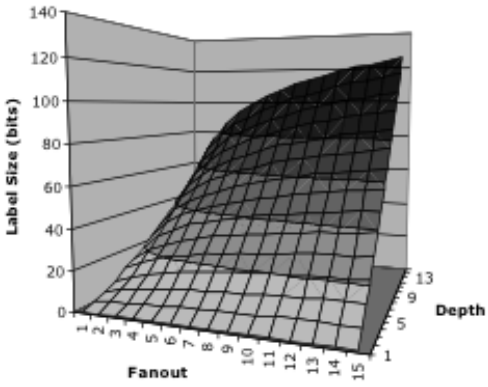


*Figure 4.* PNL label size.

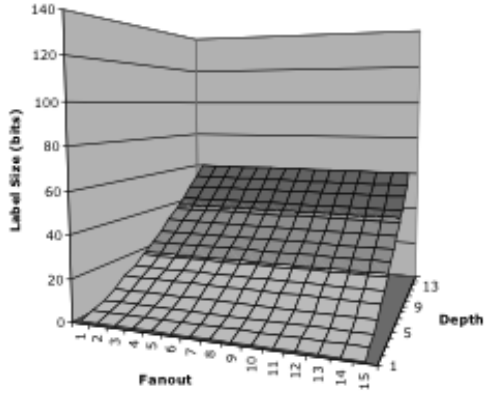


*Figure 5.* rPNL label size.

the difference between labeling schemes is not obvious. The rPNL scheme uses three labels, while PNL uses only two. However, the parent label of the PNL scheme grows significantly faster in a larger hierarchy because the self labels are not reused. As a result, we anticipate our model to be more compact than PNL.

| | Edge | Path | PNL | Range | rPNL |
|---|---|---|---|---|---|
| Size in Bits | 28 | 34 | 41 | 41 | 49 |

*Table 2.* Example tree total label size comparison.

The reciprocal function essentially finds a solution to the equation $a * x = 1 \mod b$, given a and b. We have implemented a less efficient version of this algorithm for simplicity. The current version has a time complexity of O(b). However, a more optimal version exists with a time complexity of $O(\log a)$ [1]. The simultaneous congruence function accepts $d$ primes, computes their product, and a reciprocal for each prime. The entire operation would have a time complexity of $O(\sum \log(p_i))$ operations. Furthermore, the number of primes is bound by the depth of the hierarchy $d$. Therefore, the time complexity of labeling a single node is $O(\sum_{i=0}^{d} \log(p_i))$ and $O(n * \sum_{i=0}^{d} \log(p_i))$ for $n$ nodes. The summation will have at most $d$ components with all primes $p_i$ less than or equal to $p_n$. All primes $p_i$ are bound by $p_n \leq n * \ln(n * \ln(n))$ for $n \geq 6$. Therefore, $\log(p_i) \leq \log(n * \ln(n * \ln(n)))$ and the time complexity of labeling a tree with $n$ nodes on $d$ levels will be $O(n * d * \log(n * \log(n * \log(n))))$. We know that $\log(n) \leq n$ and $n * \log(n) \leq n^2$, therefore $\log(n * \log(n)) \leq \log(n^2) \leq n$. Applying the same argument again, we get $\log(n * \log(n * \log(n))) \leq \log(n^2)$ which makes the time complexity of our algorithm $O(n * d * \log(n))$.

## 5. PNL and rPNL Benchmarks

To evaluate the proposed method, we used a hierarchy with considerable depth and fanout. The XML source of the CNN homepage has multiple nested tables, which add to the depth of the tree, and multiple links, which add to the fanout. It is a complex tree with an unbalanced structure, i.e., body tag contains the majority of the data whereas head element has only a few children. The CNN homepage contained 840 elements with 444 paths. Maximum depth and fanout were 14 and 20 respectively. Figure 6 shows the hierarchy we modeled.

*Figure 6.* CNN tree visualization.

Hardware resources also play an important role in a model's performance. The experimental system configuration is outlined in the table below.

| CPU | Intel Pentium D 2.66GHz |
|---|---|
| RAM | 2GB |
| Database | MySQL v5.0.45 |
| Code | PHP v.5.2.3 DOM/XML enabled |

*Table 3.* System configuration.

Additionally, network connection speed may have a great impact on the performance of certain schemes. If a scheme relies on client side computation, which results in very few queries being sent to the database, then network latency effects are minimal. For example, the proposed scheme will produce a single query for most tasks, provided that a small list of prime numbers is available to the client. Otherwise, network delay will occur with every query. For instance, Edge approach relies on recursive queries being sent to the server. If the network connection is slow, this model's performance will suffer. Finally, the database scheme itself affects the performance of all labeling schemes. We did not implement any indexing or other optimizations in order to test the true performance of each labeling scheme.

## 5.1. Model Size

When labeling the CNN tree, our scheme generated parent-labels that were 15 orders of magnitude smaller and took up less than half the space of PNL labels. The PNL scheme created labels up to 9.91E+25 that would require as many as 87 bits. The biggest parent-label in our reusable prime number labeling scheme was only 9.91E+10, which takes up only 37 bits. In addition to a much smaller parent-label that is much easier to factor, the rPNL scheme used only 27 prime numbers to label the entire tree instead of 840 unique primes required by the PNL scheme.

Table 4 compares model sizes of different labeling schemes and the original XML document. The size is measured in the minimum number of bytes required by each model. The Edge approach produced the most compact representation, followed by the Range labeling scheme. The rPNL scheme was better than the PNL method. Our method produces a model size that is comparable to traditional approaches, yet is smaller than the original PNL scheme.

| | XML | Edge | Path | PNL | Range | rPNL |
|---|---|---|---|---|---|---|
| CNN | 41,550 | 82,376 | 98,760 | 115,144 | 82,376 | 98,760 |

*Table 4.* CNN tree table sizes.

Note that we measured size of the hierarchy by comparing the size of the entire database table to size of the raw XML file. All tables have the same number of rows, with some columns smaller than others. MySQL storage engine may have padded/modified the data for more optimal retrieval, which is why different labeling schemes take up the same amount of space for different labels.

## 5.2. Tree Labeling Time

We recorded the time it took each scheme to label the CNN tree. Figure 7 shows that Range labeling scheme was the slowest of all models. The Edge and Path models work quickly because new label generation does not require any computation or database requests. The rPNL scheme was a little behind the Edge and Path models because it had to request a new label from the database before each new node was

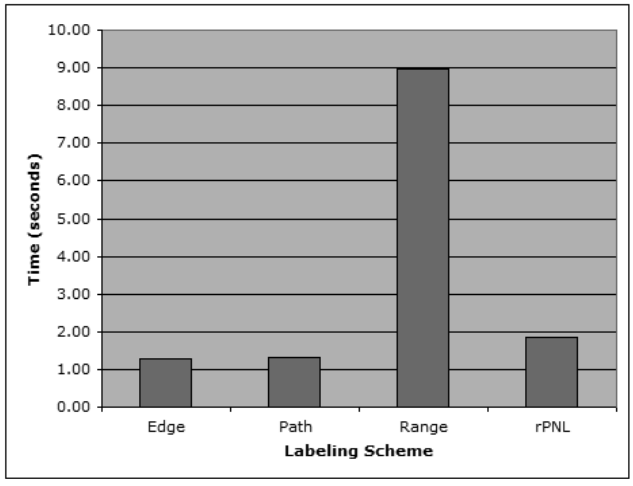


*Figure 7.* Overall tree labeling time.

saved. However, considering that it used only 27 primes to label the entire tree, the database requests could be replaced by a cached list of primes, which would dramatically improve performance. The PNL scheme took 48 seconds, and was removed from the graph to preserve the scale. Our scheme is comparable to traditional approaches and is over 20 times faster than the original PNL method.

In fact, it can model complex hierarchies, without slowing down. Figure 8 shows a more detailed view of the time it takes to label the CNN tree. Both PNL and rPNL schemes have strict rules about label uniqueness. The PNL scheme requires global uniqueness, which results in increasing delays, as the tree gets bigger. The rPNL scheme requires only local uniqueness, so very few records are referenced with every new node. The two labeling schemes are very close at the beginning because only a few labels must be checked for uniqueness. However, as the model grows, the number of labels increases and the PNL scheme must check them all. The rPNL scheme checks only the nodes on the path to the root and sibling nodes. As a result, the delay is relatively constant.

Figure 9 shows a similar comparison between rPNL and Edge scheme. The Edge labeling scheme has no drawbacks that would slow it down as it progresses down the tree, there-

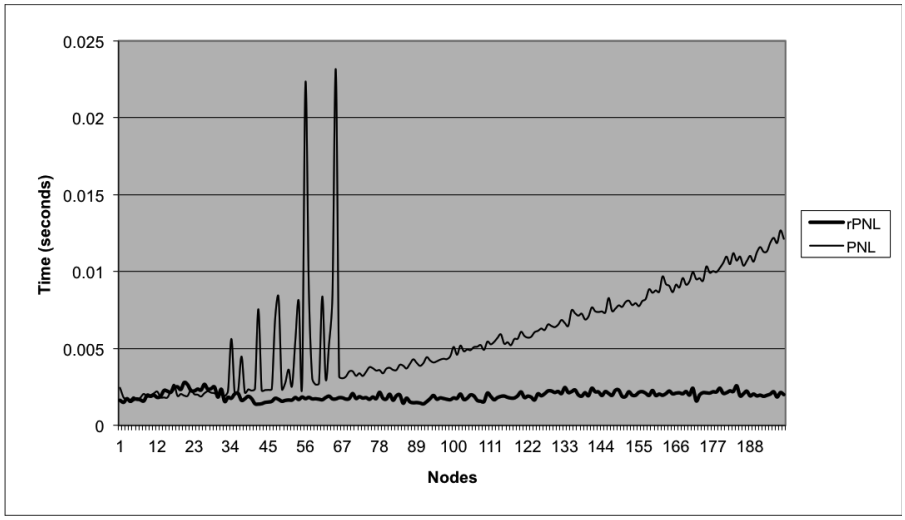


*Figure 8.* PNL vs. rPNL CNN tree modeling.

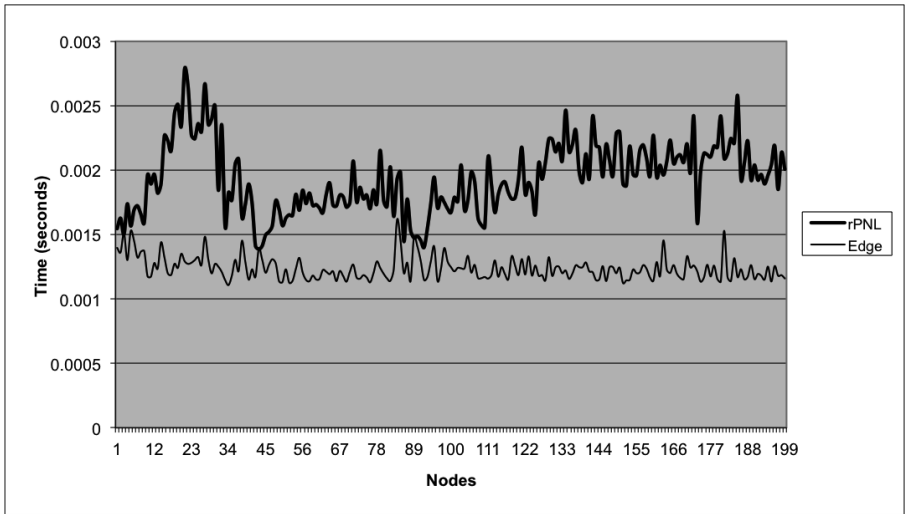


*Figure 9.* Tree labeling with rPNL and edge.

fore each node insertion should take the same amount of time and result in a straight-line graph. The small discrepancy in the Edge graph is due to physical interference, hardware resource availability, hard disk access times, etc. The proposed scheme differs from the Edge approach only slightly whenever the tree's depth increases. For example, depth of the tree goes from 4 to 13 and back to 4 between the 10th and 40th nodes. In fact, on average rPNL is 68% slower than the fastest possible labeling scheme.

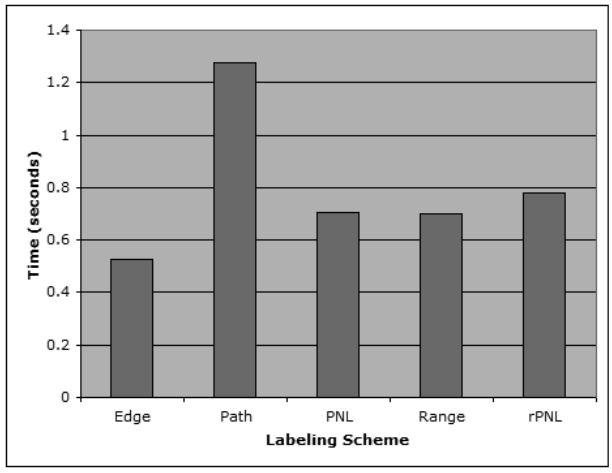


*Figure 10.* Top-Down tree traversal.

## 5.3. Direct Children Lookup

The goal of this experiment was to traverse the entire tree from top to bottom, one level at a time. For each node in a tree, only its direct children were located. The results are displayed in Figure 10. The Edge model performed best as it involved fast number lookups. The PNL scheme also used numerical lookups, but it took some time to generate the necessary labels. The Range approach performed well because it is best suited for descendant searches with a small correction for depth in this case. The Path method was the slowest because regular expression matching is complex and resource intensive. The rPNL scheme was only 10% behind PNL. Unlike PNL, rPNL parent-labels are not globally unique, so fast number lookups were not possible. Direct child lookups are usually singular, which means that, on average, the proposed method will be 48% behind the fastest available method.

## 5.4. Descendant Search

The descendant search is the most common application of labeling schemes. In fact, it is a frequent practice to search only a specific branch of the hierarchy. Figure 11 shows the results of the descendant search experiment. The desired nodes were located on different depths between 5 and 11 generations. Each algorithm inspected 71 paths and located 135 matching records. The Range descendant search produced the best results because the only function needed to iden-

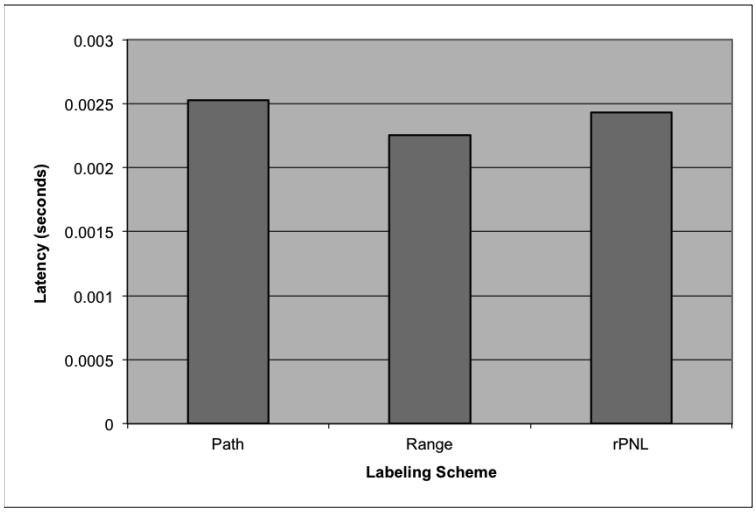


*Figure 11.* Descendant search.

tify the resulting nodes was a simple number comparison. The Path scheme was also able to successfully locate all matching nodes, but it was the slowest one. The rPNL scheme located the same nodes in considerably less time. Both prime number based models had similar results on smaller depths. However, the PNL scheme was not able to successfully complete this experiment, as it failed to compute the relationships among nodes at such extreme depths. The Edge implementation is not present on the graph to preserve scale. It used recursive queries to traverse the sub-tree, analyzing each node individually to determine if it matched the criteria. As a result, it took 0.09471 seconds to complete. This is almost forty times as slow as the rPNL scheme.

### 5.5. Update Flexibility

Tree updates are one of the most complex operations available. They usually require a number of nodes to be relabeled in order to reflect ancestor removal/addition and branch movement. The Edge labeling scheme is best suited for this purpose because only a single node needs to be updated. The Range model is also quite flexible because it assumes frequent updates. For the Path labeling scheme, each individual update is not very complex. Even though string operations are resource intensive, they are guaranteed to work at all depths of the tree. The PNL scheme can accommodate various types of updates as well. However, due to extremely large labels that are common in deep hierarchies, some of the updates might not be able to propagate through the entire sub-tree. Our scheme cannot move an entire sub-tree within a hierarchy because there is no reliable way to ensure that the prime numbers used in the destination branch are not also used in the sub-tree being moved. The only way to accomplish this would be through individual node relabeling. Therefore, our scheme is best suited for trees that do not move its branches.

## 6. Conclusions

The purpose of hierarchical data modeling is a quick determination of relationships among the nodes in a tree. Many specialized hierarchical data management solutions exist, but they may not be always available. On the other hand, relational databases are more common and are extensively used in many organizations. We researched the different ways to model hierarchical data in relational databases. To do so efficiently, a labeling scheme that supports fast and computationally light queries must be in place. Many labeling schemes exist, but no one is best for all applications.

We propose a labeling scheme that utilizes the unique characteristics of prime numbers to encode the node position in a hierarchy. Our scheme allows labels to be reused throughout the tree. This keeps the label size minimal and improves performance. Table 5 outlines the results of the experiments performed (lower number represent better ranking). Overall, rPNL scheme is not the best performer. However, it is better than PNL in four out of six categories. Furthermore, rPNL lacks the disadvantages that make traditional approaches specialized, e.g., the relabeling issue with nested sets model, the intermediate results issue with recursive expansion model, and the update performance issue of materialized path model. The experiments show that our scheme is capable of quickly and efficiently modeling complex hierarchies as well as searching them effectively. The only disadvantage of our scheme is the difficulty of moving branches within a tree. However, this kind of functionality is rare. In fact, the Edge method is designed specifically for such hierarchies. Therefore, our labeling scheme is a good general-purpose approach that performs well in most common applications.

| | Edge | Path | PNL | Range | rPNL |
|---|---|---|---|---|---|
| Model Size | 1 | 3 | 5 | 2 | 4 |
| Insert Performance | 1 | 2 | 5 | 4 | 3 |
| Direct Children Lookup | 1 | 5 | 3 | 2 | 4 |
| Descendant Search | 4 | 3 | 5 | 1 | 2 |
| Ancestor Determination | 3 | 2 | 5 | 1 | 4 |
| Update Flexibility | 1 | 3 | 4 | 2 | 5 |

*Table 5.* Experiment summary.

## References


[1] D. E. KNUTH, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*. Addison-Wesley, 1997.

[2] M. YOSHIKAWA, T. AMAGASA, T. SHIMURA, S. UEMURA, XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Transactions on Internet Technology*, **1**(1) (2001), 110–141.

[3] X. WU, M.-L. LEE, W. HSU, A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *Proceedings of the 20th International Conference on Data Engineering (ICDE'04)*, (2004), pp. 66–78.

[4] F. WEIGEL, K. U. SCHULZ, H. MEUSS, The BIRD Numbering Scheme for XML and Tree Databases – Deciding and Reconstructing Tree Relations Using Efficient Arithmetic Operations. In *Proceedings of the 3rd International Conference on Database and XML Technologies*, (2005), pp. 49–67.

[5] V. TROPASHKO, Nested Intervals Tree Encoding in SQL. *ACM Special Interest Group on Management of Data*, **34**(2) (2005), 47–52.

[6] W. M. SHUI, F. LAM, D. K. FISHER, R. K. WONG, Querying and Maintaining Ordered XML Data Using Relational Databases. In *Proceedings of the 16th Australasian Database Conference*, (2005), pp. 85–94.

[7] J. SHANMUGASUNDARAM, K. TUFTE, C. ZHANG, G. HE, D. DEWITT, J. F. NAUGHTON, Relational Databases for Querying XML Documents: Limitations and Opportunities. In Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99), (1999), pp. 302–314.

[8] D. PREUVENEERS, Y. BERBERS, Prime Numbers Considered Useful: Ontology Encoding for Efficient Subsumption Testing, 2006.

[9] P. E. O'NEIL, E. J. O'NEIL, S. PAL, I. CSERI, G. SCHALLER, N. WESTBURY, ORDPATHs: Insert-Friendly XML Node Labels. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, (2004), pp. 903–908

[10] C. MATHIS, T. HÄRDER, K. SCHMIDT, Storing and Indexing XML Documents Upside Down. *Computer Science – R&D*, **24**(1-2) (2009), 51–68.

[11] M. MANI, D. LEE, XML to Relational Conversion Using Theory of Regular Tree Grammars. In *Proceedings of the VLDB 2002 Workshop EEXTT and CAiSE 2002 Workshop DTWeb on Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web-Revised Papers*, (2002), pp. 81–103.

[12] Q. LI, B. MOON, Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, (2001), pp. 361–370.

[13] C. LI, T. W. LING, M. HU, Reuse or Never Reuse the Deleted Labels in XML Query Processing Based on Labeling Schemes. In *Proceedings of the 11th International Conference on Database Systems for Advanced Applications (DASFAA'06)*, (2006), pp. 659–673.

[14] C. LI, T. W. LING, *QED*: A Novel Quaternary Encoding to Completely Avoid Re-Labeling in XML Updates. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, (2005) Bremen, Germany, pp. 501–508.

[15] C. WALLACE, Strategies for Encoding XML Documents in Relational Databases: Comparisons and Contrasts, 2006.

[16] A. A. KHAING, N. L. THEIN, A Persistent Labeling Scheme for Dynamic Ordered XML Trees. *Web Intelligence*, 2006, pp. 498–501

[17] H. JIANG, H. LU, W. WANG, J. X. YU, XParent: An Efficient RDBMS – Based XML Database System. In *Proceedings of the 18th International Conference on Data Engineering*, (2002), pp. 335–336.

[18] H. JIANG, H. LU, W. WANG J. X. YU, Path Materialization Revisited: An Efficient Storage Model for XML Data. *Australasian Database Conference*, (2002).

[19] T. HÄRDER, C. MATHIS, Key Concepts for Native XML Processing. *From Active Data Management to Event-Based Systems and More*, (2010), pp. 1–19.

[20] T. HÄRDER, M. P. HAUSTEIN, C. MATHIS, M. WAGNER, Node Labeling Schemes for Dynamic XML Documents Reconsidered. *Data Knowledge Engineering*, **60**(1) (2007), 126–149.

[21] D. FLORESCU, D. KOSSMANN, Storing and Querying XML Data Using an RDBMS. *Bulletin of the Technical Committee on Data Engineering*, **22**(3) (1999), 27–34.

[22] M. DUONG, Y. ZHANG, LSDX: A New Labeling Scheme for Dynamically Updating XML Data. In *Proceedings of the 16th Australasian Database Conference*, (2005) Darlinghurst, Australia, pp. 185–193.

[23] M. M. DAVID, ANSI SQL Hierarchical Processing Can Fully Integrate Native XML. *ACM Special Interest Group on Management of Data*, **32**(1) (2003), 41–46.

[24] E. COHEN, H. KAPLAN, T. MILO, Labeling Dynamic XML Trees. *Symposium on Principles of Database Systems*, (2002).

[25] J. CELKO, *Joe Celko's SQL for Smarties: Trees and Hierarchies*. Morgan Kaufmann Publishers Inc., San Francisco, 2004.

[26] R. CATHEY, S. M. BEITZEL, E. C. JENSEN, D. A. GROSSMAN, O. FRIEDER, Using a Relational Database for Scalable XML Search. *The Journal of Supercomputing*, **44**(2) (2008), 146–178.

[27] D. BRANDON, Recursive Database Structures. *Journal of Computing Sciences in Colleges*, **21**(2) (2005), 295–304.

[28] T. BÖHME, E. RAHM, Supporting Efficient Streaming and Insertion of XML Data in RDBMS. In *Proceedings of the Third International Workshop on Data Integration over the Web*, (2004) Riga, Latvia.

[29] M. ATAY, A. CHEBOTKO, D. LIU, S. LU, F. FOTOUHI, Efficient Schema-Based XML-to-Relational Data Mapping. *Information Systems*, **32**(3) (2007), 458–476.

[30] T. AMAGASA, M. YOSHIKAWA, S. UEMURA, QRS: A Robust Numbering Scheme for XML Documents. In *Proceedings of the 19th International Conference on Data Engineering*, (2003).

*Received:* December, 2013
*Revised:* May, 2014
*Accepted:* May, 2014

*Contact addresses:*

Serhiy Morozov
University of Detroit Mercy
Detroit MI
USA
e-mail: morozov.sergey@gmail.com

Hossein Saiedian
University of Kansas
Lawrence KS
USA
e-mail: h.saiedian@ku.edu

Hanzhang Wang
University of Detroit Mercy
Detroit MI
USA
e-mail: wangha2@udmercy.edu

SERHIY MOROZOV is currently an assistant professor at the Mathematics, Computer Science, and Software Engineering department at the University of Detroit Mercy. He teaches undergraduate and graduate courses in software engineering and serves on the software engineering assessment committee. He is currently involved in the recommender systems research, but his other interests include data mining and Web development. Prior to his academic career, Serhiy worked as a Web developer for over 5 years. He received a BA degree from the Westminster College in 2005, MS degree from the University of Kansas in 2007, and PhD from the University of Kansas in 2011. Serhiy is a member of the Institute of Electrical and Electronics Engineers (IEEE) and a member of the Association for Computing Machinery (ACM).

HOSSEIN SAIEDIAN (Ph.D., Kansas State University, 1989) is currently the director of IT undergraduate and graduate degree programs and an associate chair and a professor of software engineering at the Department of Electrical Engineering and Computer Science at the University of Kansas (KU) and a member of the KU Information and Telecommunication Technology Center (ITTC). His career includes 27 years of research and teaching in software engineering, over 160 publications, several research fundings, as well as an array of industrial consulting and training courses. He has won a variety of research and teaching awards and was ranked among the top-10 software engineering scholars by the Journal of Systems and Software. He has served as the conference general, program chair, track chair and/or committee member for many of the prestigious IEEE-CS and ACM conferences. Saiedian has over 150 publications on a variety of topics on computing but primarily on software engineering and has supervised the work of more than 65 Ph.D. and Master's students.

HANZHANG WANG is a software engineering graduate student at the University of Detroit Mercy. He received his undergraduate degree in Computer Information Systems from the same school in 2011. Hanzhang is currently involved in testing and data management research projects. He also serves as a technical co-founder at a local start-up, specializing in mobile application development. In the past, Hanzhang received the 2014 Mathematics and Software Engineering Graduate Student of the Year award from the University of Detroit Mercy and won the 1st place at the Chinese National Olympiad in Informatics in 2006.

## Appendix 1: The Next-Prime Algorithm

Below please find various components of `get_next_rpnl_prime` algorithm.

Listing 1: PNL

```php
<?php
include "get_next_pnl_prime.php";

// load xml file into the DOM tree
$tree = new DOMDocument;
$tree->preserveWhiteSpace = false;
$tree->load('tree.xml');

pnl($tree->documentElement);

function pnl($node, $parent = 1)
{
    $self = get_next_pnl_prime();

    echo "<".$node->nodeName.">\tself: ".$self."\tparent: ".$parent."\n";

    // proceed with the first child (if any), with a longer parent label
    if($node->firstChild)
        pnl($node->firstChild, $parent*$self);

    // proceed with the next sibling, with the same parents label
    if($node->nextSibling)
        pnl($node->nextSibling, $parent);
}
?>
```

Listing 2: Get Next PNL Prime

```php
<?php
function get_next_pnl_prime()
{
    static $index = 0;
    $primes = array(2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79);

    return $primes[$index++];
}
?>
```

Listing 3: rPNL

```php
<?php
include "simultaneous_congruence.php";
include "get_next_rpnl_prime.php";

// load xml file into the DOM tree
$tree = new DOMDocument;
$tree->preserveWhiteSpace = false;
$tree->load('tree.xml');

rpnl($tree->documentElement);

function rpnl($node, $parents=array(), $sibling_offset = 0)
{
    $self = get_next_rpnl_prime($parents, $sibling_offset);

    $parent=1;
    foreach($parents as $prime)
        $parent=$parent*$prime;

    $sc = get_sc($parents);

    echo "<".$node->nodeName.">\tself: ".$self."\tparent: ".$parent."  sc: ".$sc."\
        n";

    // proceed with the first child (if any), with a longer parent list
    if($node->firstChild)
        rpnl($node->firstChild, array_merge($parents, array($self)), 0);

    // proceed with the next sibling, with the same parents list
    if($node->nextSibling)
        rpnl($node->nextSibling, $parents, $sibling_offset+1);
}
?>
```

Listing 4: Simultaneous Congruence Library

```php
<?php
// Calculate the multiplicative inverse of a number.
// i.e, find x in the equation a*x mod b = 1
// this is only possible when a and b are coprime
function reciprocal($a, $b)
{
    $answer = 0;
    $a = $a % $b; // reduce a mod b if necessary

    // find the smallest (positive or negative) x
    for($i=1; $i<=$b/2; $i++)
    {
        $product = $a * $i;
        if($product % $b == 1)
            return $i;
        elseif(-$product % $b == (-$b+1))
            return -$i;
    }
    return $answer;
}

//Calculate the simultaneous congruence number, given a sequence of primes
function get_sc($parents)
{
    $sc = 0; // Simultanous Congruence (SC)

    $N  = 1; // product of all parent labels
    foreach($parents as $prime)
        $N = $N * $prime;

    // apply Chinese Remainder theorem
    foreach($parents as $position=>$prime)
    {
        $n_i        = $N/$prime;
        $reciprocal = reciprocal($n_i, $prime);
        $sc         = $sc + $position*$n_i*$reciprocal;
    }

    $sc = $sc % $N; // reduce the number if necessary
    if($sc<0)
        $sc = $sc + $N; // make sure the answer is positive

    return $sc;
}
?>
```

Listing 5: Get Next rPNL Prime

```php
<?php
// get the next available prime that has not yet been used by any of
// the parents or siblings and is bigger than the current node's generation
function get_next_rpnl_prime($parents, $sibling_offset = 0)
{
    $primes = array(2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79);
    $index  = 0;

    // prime must be greater than its generation
    while(count($parents)>$primes[$index])
        $index++;

    // prime may not already be used as parent label
    while(in_array($primes[$index], $parents))
        $index++;

    // prime must be different from existing siblings
    $index = $index + $sibling_offset;

    return $primes[$index];
}
?>
```