By Laurent Martin

A Static Relational Interval Tree

Suppose that a large car rental company needs to track each customer contract, which is represented by a row in a database table. Each row includes the customer's identifier and a pair of dates delimiting the rental time period. A SQL request to find all of the contracts effective between two dates might end up scanning a large portion of the table because interval intersection queries generally have no built-in support in a relational database management system (RDBMS).

In the year 2000, a group of German researchers—Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl from the Institute for Computer Science at the University of Munich—invented the Relational Interval Tree (RI-Tree), an extremely ingenious structure to efficiently handle interval queries in SQL. They wrote about it in the paper "Managing Intervals Efficiently in Object-Relational Databases."

However, one aspect of the RI-Tree is a bit tricky: There is no simple way to handle batched insertion (i.e., INSERT SELECT statements) because each inserted row might modify the value of one of the tree's four parameters. This article covers a variant of the RI-Tree, the *Static RI-Tree*, which supports batched insertion with excellent performance.

The original RI-Tree has four associated parameters: offset, leftRoot, rightRoot, and minstep. The function of these parameters is to save CPU time by controlling the tree's height, thereby avoiding many useless iterations while traversing the tree. Unfortunately, the insertion of an interval is dependent on these parameters, which in turn, might be modified by the insertion itself. This makes writing an INSERT SELECT statement difficult.

The Static RI-Tree does not use any of the original parameters. Instead, a newly inserted interval is dependent only on row data, which makes using an INSERT SELECT statement feasible. Consequently, this tree covers the entirety of the data space all of the time. No dynamic expansion takes place, hence the name Static RI-Tree. Because this new variant is static, it traverses the tree in a different way than the original RI-Tree to avoid the many iterations just mentioned.

The Static RI-Tree

The backbone of the Static RI-Tree is implemented the same as a binary tree: with its nodes labeled as integer numbers from 1 through 2^N-1 in order (where N is the size of the integer data type used, in bits). The value of the root is set to 2^{N-1}. You can use dates instead of integers by providing a simple mapping between dates and integers. Figure 1 on page 56 shows a sample binary tree in which N = 5.

The *fork node* plays a crucial role in the structure because it determines where an interval is inserted into the Static RI-Tree. Its value is used in the *node* column of the Intervals table, as shown

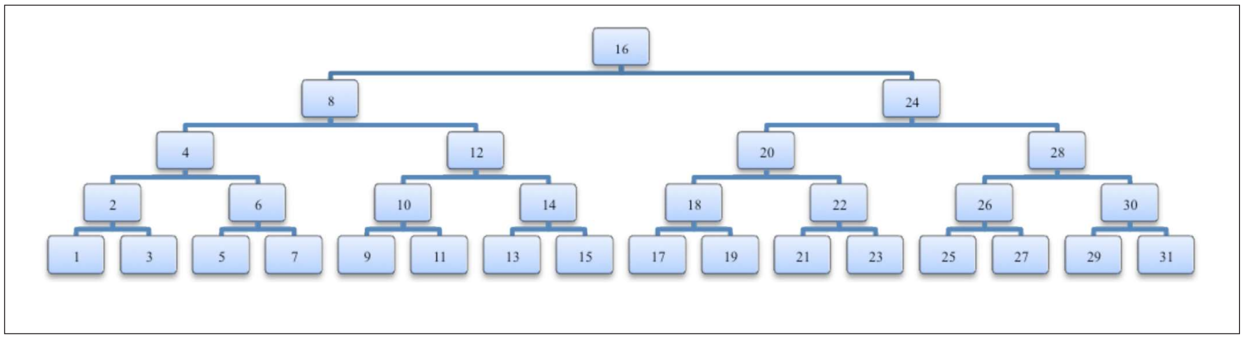

Figure 1: A sample binary tree with nodes labeled as integers

Listing 1: The SQL implementation of the Static RI-Tree

```
CREATE TABLE Intervals
(
  id    INT NOT NULL PRIMARY KEY,
  node  INT NOT NULL, -- Computed as forkNode(lower, upper)
  lower INT NOT NULL,
  upper INT NOT NULL
);

CREATE INDEX lowerIndex ON Intervals(node, lower);
CREATE INDEX upperIndex ON Intervals(node, upper);
```

in Listing 1. Also shown in this code are the two indexes built on this table, as in the original RI-Tree.

Assuming an interval [lower, upper], with lower and upper being integers, the fork node is defined as the topmost node *w* in the following relation: *lower <= w <= upper.*

Listing 2: The forkNode function

```
FUNCTION int forkNode(int lower, int upper) {
  int node = 2N-1;
  for(int step = node/2; step >= 1; step /= 2)
    if upper < node
      node -= step;
    else if node < lower
      node += step;
    else
      break;

  return node;
}
```

The forkNode function iterates through the tree to the fork node. The fork node plays a crucial role because it determines where an interval is inserted into the tree.

As described in the original paper, the forkNode function proceeds by iterating through the tree from the root down to the fork node, as shown in Listing 2.

The Problem

With this simplified implementation, there is a problem: The number of iterations needed in a call to forkNode depends on the tree's height and can waste many CPU cycles. For instance, suppose we are using 32-bit signed integers that are positive numbers ranging from 1 through 2^{31}-1 = 2147483647. If we call forkNode with the interval [734288, 734317], the function starts at the root, whose value is 2^{30} = 1073741824, then steps through the following nodes:

```
536870912, 268435456, 134217728, 67108864,
33554432, 16777216, 8388608, 4194304, 2097152,
1048576, 524288, 786432, 655360, 720896, 753664,
737280, 729088, 733184, 735232, 734208, 734720,
734464, 734336, 734272, 734304
```

That's 26 iterations to reach the fork node!

A New Way of Finding the Fork Node

For the remainder of this article, let n_x represent a node of the tree with value x. Let us assume we are searching the fork node for an interval [lower, upper] in which n_{lower} is the node with the value of *lower* and n_{upper} is the node with the value of *upper*. Looking closely at the sample binary tree in Figure 1, notice that the fork node can be determined from *lower* and *upper* alone. Instead of starting from the root, we can walk up the tree from both n_{lower} and n_{upper} until we reach their lowest common ancestor. This will be the fork node.

To demonstrate that the lowest common ancestor of n_{lower} and n_{upper} is indeed their fork node, consider the following cases:

- **Case 1:** If n_{lower} and n_{upper} have a lowest common ancestor of n_x, which is neither n_{lower} nor n_{upper}, then n_{lower} will be in n_x's left subtree and n_{upper} will be in n_x's right subtree. As such, x is in the interval [lower, upper]. Because the value of any ancestor of n_x is outside this interval, n_x is the fork node.
- **Case 2:** If n_{lower} = n_{upper}, the fork node is that node. It is also the lowest common ancestor.
- **Case 3:** If n_{lower} is an ancestor of n_{upper}, the fork node will be n_{lower} because it is the highest node whose value is contained in the interval [lower, upper], while all of its ancestors are outside this interval. Also, n_{lower} is the lowest common ancestor.
- **Case 4:** If n_{upper} is an ancestor of n_{lower}, the fork node will be n_{upper} for reasons symmetric to those of Case 3.

We now need a method to compute the fork node efficiently from *lower* and *upper*.

The sample binary tree in Figure 2 (page 58) has its nodes labeled with their decimal and binary values. Looking at the binary values of the integers in the tree, we can make several observations:

- **Observation 1:** For any non-leaf node possessing an even value, the binary representation of this value will have a non-empty sequence of trailing 0 bits, whose number depends on the height of the node in the tree. Let us call *L* the sequence of leading bits before these trailing 0 bits. L uniquely identifies a non-leaf node.
- **Observation 2**: Assume we have a non-leaf node n_x with a leading bit sequence of L. In n_x's left subtree, the prefix of the value of any node is L with its rightmost 1 bit cleared. In n_x's right subtree, the prefix of the value of any node is L.
- **Observation 3**: Assume we have a non-leaf node n_x. The node n_{x-1} belongs to the left subtree of n_x (assuming that the value 0 is allowed in the tree). Because the fork node of interval [x, y] (where x <= y) is n_x or an ancestor of n_x, the interval [x-1, y] has the same fork node. This allows the replacement of x with x-1 without changing the resulting fork node.
- **Observation 4**: Assume we have a leaf node n_z. The last bit of z is set to 1 because it is the value of a leaf node. In this case, z-1 will have the exact same bit representation as z, except for the last bit, which is set to 0.

These observations enable us to define a method to compute the fork node directly from n_{lower} and n_{upper}. Using these observations, let us examine the four cases previously mentioned and then incorporate the ideas into one general method.

In Case 2, when n_{lower} is a non-leaf node, Observation 3 tells us that the leading sequence L can be computed as the leftmost matching bits of *lower-1* and *upper* with a "1" bit appended. When n_{lower} is a leaf node, consider the following: Because Observation 4 applies and *lower* = *upper*, *lower-1* differs from *upper* on the last bit only. Therefore, the leftmost matching bits of *lower-1* and *upper* with a "1" bit appended equals *lower* and *upper*, which is also the fork node.

In Cases 1, 3, and 4, when n_{lower} is a non-leaf node, Observation 3 tells us that the leading sequence L can be computed as the leftmost matching bits of *lower-1* and *upper* with a "1" bit appended. If n_{lower} is a leaf node, consider the following: Because Observation 4 applies, *lower-1* is the only value differing from *lower* exclusively on the last bit. Also, because *lower-1* < *lower*, then *lower-1* ≠ *upper*. Further, because *lower* < *upper*, *lower* differs from *upper* on another bit than the last. As a consequence, the last bit of *lower* is irrelevant in the determination of the fork node, so *lower* can be replaced by *lower-1*.

To conclude, the computation of the fork node boils down to:

1. Finding the sequence of leftmost matching bits between *lower-1* and *upper*.
2. Appending a "1" bit.
3. Filling the remaining right bits with 0's.

Let us investigate a couple of examples:

- **Example 1:** In the tree represented in Figure 2, we are looking for the fork node of the interval [5, 10]. This is Case 1, and the fork node is 8. Let us replace 5 with 4, examine the binary representations of 4 and 10, and search for the leftmost matching bits. The search reveals a sequence reduced to one 0 bit. The leading sequence of the fork node is thus L = 01. This leading sequence uniquely identifies the node labeled 8.
- **Example 2:** Let us find the fork node of the interval [12, 15]. This is Case 3, and the fork node is 12. Replace 12 with 11, examine the binary representations of 11 and 15, and search for the leftmost matching bits. The search reveals the sequence 01. After appending a "1" bit, we get the leading sequence L = 011, which corresponds to the node labeled 12.
- **Example 3:** We are now looking for the fork node of the interval [21, 24]. This is Case 4, and the fork node is 24. Replace 21 with 20, examine the binary representations of 20 and 24, and search for the leftmost matching bits. The search reveals the sequence 1. After appending a "1" bit, we get the leading sequence L = 11, which corresponds to the node labeled 24.
- **Example 4:** Let us find the fork node of the interval [21, 21]. This is Case 2, and the fork node is 21. Replace 21 with 20, examine the binary representations of 20 and 21, and search for the leftmost matching bits.

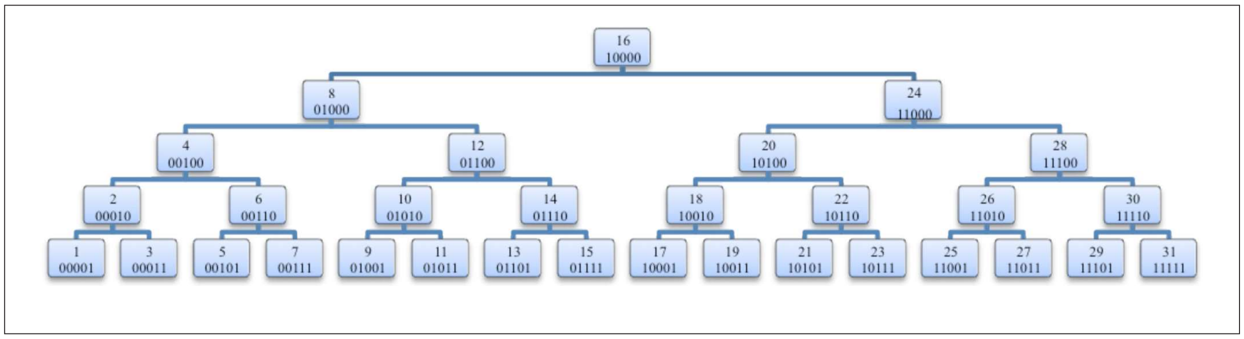

Figure 2: The virtual backbone of a Static RI-Tree