Advanced interval queries with the Static Relational Interval Tree

In my previous article "[A Static Relational Interval Tree]", I described a new powerful structure, the Static Relational Interval Tree (or Static RI-Tree), efficiently handling interval intersection queries in SQL. The power of Static RI-Trees resides in their ability to efficiently partition a set of intervals so that queries can focus on just a subset of the intervals. In this article, I present other kinds of queries that are also nicely handled by a Static RI-Tree.

Remember the large car rental company that I mentioned in the introduction to the previous article? What if, instead of requesting the contracts effective *between* two dates, we wish to find those which started and ended between the two dates? Or those that started before a first date and ended after a second date? The Static Relational Interval Tree, with just a few additional tweaks, can handle these other interval relationships nicely.

Refining the intersection query

Let's begin with two improvements to the intersection query from the previous article. As a reminder, a sample binary tree for 5-bit positive integers is presented in Figure 1 below. This binary tree is also the virtual backbone of a Static RI-Tree.

Filtering out useless values from the BitMasks table

Cast your mind back to the queries used to fill the leftNodes and rightNodes tables (shown in Listing 1), and to the BitMasks table, a sample of which is shown in Table 1 for 5-bit integers. Filling leftNodes and rightNodes is a preliminary step in the execution of an intersection query. If you examine these queries, you may notice that they return some unnecessary values. Fortunately, these extra values don't change the result of an intersection query. They do, however, cause additional I/O.

Listing 1: SQL code to populate the leftNodes and rightNodes tables

```
-- Filling up the leftNodes table
SELECT :x & b1 FROM BitMasks
WHERE :x & b2 <> 0;

-- Filling up the rightNodes table
SELECT (:x & b1) | b3 FROM BitMasks
WHERE :x & b3 = 0;
```

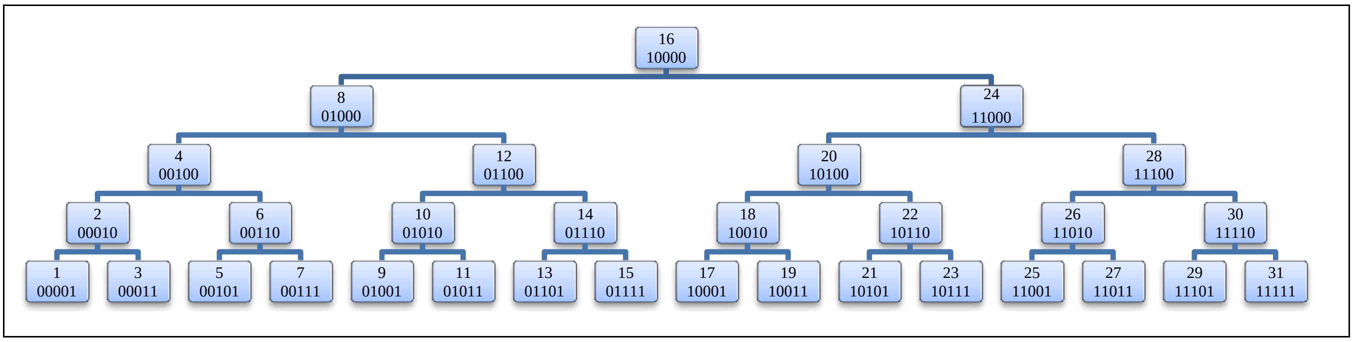

Figure 1: The virtual backbone of a Static RI-Tree. Node values are shown both in decimal and binary representations.

b1	b2	b3
11110	00001	00010
11100	00010	00100
11000	00100	01000
10000	01000	10000

Table 1: Sample BitMasks table for 5-bit integers

The query filling the leftNodes table may return the useless value 0. For instance, suppose the bound variable x has the binary value 00110, which is 6 in decimal. The query to fill the leftNodes table, because of its WHERE clause ":x & b2 <> 0", only retains the b1 values 11100 and 11000. For the b1 value 11100, the query selects 00110 & 11100 = 00100, which is a valid value for leftNodes. For the b1 value 11000, the query selects 00110 & 11000 = 00000, or 0 in decimal, which is useless for leftNodes.

The query filling the rightNodes table may return values from x's right subtree, which are useless. For instance, say variable x has the binary value 01100, which is 12 in decimal. The query to fill the rightNodes table, because of its WHERE clause ":x & b3 = 0", only retains the (b1, b3) pairs (11110, 00010) and (10000, 10000). For these pairs, the select expression "(:x & b1) | b3" yields (01100 & 11110) | 00010 = 01110 and (01100 & 10000) | 10000 = 10000 respectively. The former, 01110 (14 decimal), is part of x's right subtree, as you can see in Figure 1 above, so it's useless. In fact, all values of x to be collected into rightNodes should be found in the following way:

1. Looking at x's binary value, find the rightmost 0 bit *having at least one 1 bit to its right*.
2. If none is found, exit.
3. Set that rightmost 0 bit.
4. Clear all 1 bits to its right.
5. Retain the new value obtained and set x to this value.
6. Go to step 1.

Let's apply the algorithm above to our example, with x having the value 01100 (12 decimal). Step 1 tells us to find the rightmost 0 bit having at least one 1 bit to its right. Only the leftmost 0 bit of 01100 qualifies. Following steps 3 and 4, the resulting value is 10000. Applying step 1 again to 10000 yields no qualifying 0 bit, so we exit the algorithm. Thus, the only value produced for rightNodes is 10000, or 16 decimal.

The original queries to fill the leftNodes and rightNodes tables can easily be fixed to exclude all spurious values; you can find the corrected queries in Listing 2 below.

Listing 2: Corrected SQL code to populate the leftNodes and rightNodes tables without spurious values

```
-- Filling up the leftNodes table
SELECT :x & b1 FROM BitMasks
WHERE :x & b2 <> 0 AND :x & b1 <> 0;

-- Filling up the rightNodes table
SELECT (:x & b1) | b3 FROM BitMasks
WHERE :x & b3 = 0 AND :x & b1 <> :x;
```

Filtering out nodes outside the actual value range

Another interesting optimization, which we'll refer to as the *range optimization*, can further reduce the I/O of intersection queries when the actual values of the Interval table's node column don't spread over the full range of possible values for the integer data type, i.e., the minimum and/or maximum values are far beyond the integer type's minimum and maximum values. In order to implement this optimization, when computing node values to fill the leftNodes and rightNodes tables, we should only consider values within the minimum and maximum node values range.

Fortunately, thanks to the lowerIndex and upperIndex indexes, computing the minimum and maximum value of the node column is very cheap and costs only 2 B-tree lookups. Each value filtered out with this idea saves one B-tree lookup. Listing 3 shows the final optimized queries to fill the leftNodes and rightNodes tables. Of course, if the range of actual values in use starts at, or close to, the minimum value, you can omit the minimum value test and save 1 B-tree lookup.

Listing 3: Optimized SQL code to populate the leftNodes and rightNodes tables without spurious or out-of-range values

```
-- Filling up the leftNodes table
SELECT :x & b1 FROM BitMasks
WHERE :x & b2 <> 0 AND :x & b1 <> 0
 AND :x & b1 >=
  ( SELECT MIN(node) FROM Intervals);

-- Filling up the rightNodes table
SELECT (:x & b1) | b3 FROM BitMasks
WHERE :x & b3 = 0 AND :x & b1 <> :x
 AND (:x & b1) | b3 <=
  (SELECT MAX(node) FROM Intervals);
```

Introducing other kinds of interval queries

Up to now, we've been focusing on intersection queries only. Other kinds of interval queries can also be useful, and the Static RI-Tree can handle them pretty well, even if it is not always as efficient as an intersection query. Let's examine these other kinds of queries.

A paper entitled "Object-Relational Indexing for General Interval Relationships", published in 2001 by Hans-Peter Kriegel, Marco Pötke and Thomas Seidl, from the University of Munich, in Germany, studies how the RI-Tree can be extended to handle the 13 general interval relationships defined by Allen (Allen J.F.: "Maintaining Knowledge about Temporal Intervals". Communications of the ACM 26(11): 832-843, 1983). The paper refines the partitioning of RI-Tree nodes in the neighborhood of the lower and upper bounds of the query interval: instead of merely relying on the leftNodes and rightNodes tables, it defines 12 classes of nodes, combined into 13 sets to support the 13 queries implementing Allen's interval relationships. The classes of nodes are named **topLeft**, **bottomLeft**, **innerLeft**, **topRight**, **bottomRight**, **innerRight**, **lower**, **fork**, **upper**, **allLeft**, **allInner** and **allRight**.

Computing the new node classes

Among the node classes listed above, some need to be computed, while others will not.

The allLeft, allInner and allRight classes don't need to be computed, because they can be represented by simple predicates. In addition, their computation could be too costly because of the potentially large number of nodes they could contain.

The lower, upper and fork classes don't need to be computed because they are singleton classes: each of them contains exactly one node.

Since bottomLeft is always used with topLeft and bottomRight with topRight, we don't need to compute bottomLeft nor bottomRight. Since bottomLeft and topLeft together form leftNodes and bottomRight and topRight together form rightNodes, we need to compute leftNodes and rightNodes instead, but we already know how to do it.

Consequently, we only need to compute the 6 classes topLeft, leftNodes, innerLeft, innerRight, topRight and rightNodes. Note that all of these classes never contain more nodes than the height of the RI-Tree.

Kriegel, Pötke and Seidl's paper uses integer arithmetic, applied in an iterative fashion, to compute these node classes. However, as explained in my previous article, while database engines are extremely efficient with set-based queries, they often perform poorly performance with language-based iterative and conditional logic. I therefore found it useful to turn this iterative logic into set-based queries. Later in this article, we'll examine the benefit of this approach in specific tests developed with Microsoft SQL Server 2008 R2.

topLeft

As the nodes in topLeft are all left ancestors of the fork node, topLeft can be easily computed by applying the leftNodes query to fork:

```
SELECT fork & b1
FROM   BitMasks
WHERE  fork & b2 <> 0
  AND  fork & b1 <> 0;
```

topRight

As the nodes in topRight are all right ancestors of the fork node, topRight can be computed by applying the rightNodes query to fork:

```
SELECT (fork & b1) | b3
FROM   BitMasks
WHERE  fork & b3 = 0
  AND  fork & b1 <> fork;
```

bottomLeft and bottomRight

Just to satisfy our curiosity, bottomLeft can be computed by adding a predicate to the leftNodes query, excluding nodes from topLeft:

```
SELECT lower & b1
FROM   BitMasks
WHERE  lower & b2 <> 0
  AND  lower & b1 <> 0
  -- Excluding nodes from topLeft:
  AND (lower & b1) | fork <> fork;
```

Similarly, bottomRight can be computed by adding a simple predicate to the rightNodes query, which excludes nodes from topRight:

```
SELECT (upper & b1) | b3
FROM   BitMasks
WHERE  upper & b3 = 0
  AND  upper & b1 <> upper
  -- Excluding nodes from topRight:
  AND  ((upper & b1) | b3) & fork = fork;
```

innerLeft

To compute innerLeft, we can apply the query to fill rightNodes to lower and set fork as an upper bound:

```
SELECT (lower & b1) | b3
FROM   BitMasks
WHERE  lower & b3 = 0
  AND  lower & b1 <> lower
  -- Excluding nodes to the right of
  -- the fork node:
  AND  (lower & b1) | b3 < fork;
```

innerRight

To compute innerRight, we can apply the query to fill leftNodes to upper and set fork as a lower bound:

```
SELECT upper & b1
FROM   BitMasks
WHERE  upper & b2 <> 0
  AND  upper & b1 <> 0
  -- Excluding nodes to the left of
  -- the fork node:
  AND  upper & b1 > fork;
```

Writing the queries for Allen's 13 interval relationships

Kriegel, Pötke and Seidl's paper summarized the SQL queries for Allen's 13 interval relationships. These queries use the pre-computed node classes. In this section, let's examine how to adapt these queries to use our set-based computation of the node classes. We'll refer to those rewritten queries as the *Static RI-Tree queries for interval relationships*, or *Static RI-Tree queries* for short.

The range optimization can be applied to all Static RI-Tree queries, using pre-computed node classes. Note that, in the following queries, we'll leave it out in the interest of clarity. However, this optimization should be considered in production queries, because it can reduce I/O.

In the following, lower and upper are the integers representing the bounds of the interval, and the interval table is defined as:

```
CREATE TABLE Intervals
(
  id    INT NOT NULL PRIMARY KEY,
  -- Computed as the fork node of
  -- the interval [lower, upper]:
  node  INT NOT NULL,
  lower INT NOT NULL,
  upper INT NOT NULL
);
```

Before

The original query is:

```
SELECT id FROM Intervals
WHERE  node  < :lower
  AND  upper < :lower
```

This query does not make use of the node classes, so there's nothing to rewrite; we'll therefore use the query as it is.

Meets

The original query is:

```
SELECT id
FROM   Intervals i
JOIN   :(topLeft U bottomLeft U lower) q
  ON   i.node = q.node
  AND  i.upper = :lower
```

Let's replace the joined table by a table expression using our precomputed node class queries. Since topLeft and bottomLeft together form leftNodes, the query is:

```
SELECT id
FROM   Intervals i
JOIN   (SELECT :lower & b1 AS node
        FROM   BitMasks
        WHERE  :lower & b2 <> 0
          AND  :lower & b1 <> 0
        UNION ALL
        SELECT :lower) q
  ON   i.node  = q.node
  AND  i.upper = :lower
```

Overlaps

The original query is:

```
SELECT id
FROM   Intervals i
JOIN   :(topLeft U bottomLeft) q
  ON   i.node = q.node
WHERE  i.upper > :lower
  AND  i.upper < :upper

UNION ALL

SELECT id
FROM   Intervals i
JOIN   :(innerLeft U lower U fork) q
  ON   i.node = q.node
WHERE  i.lower < :lower
  AND  i.upper > :lower
  AND  i.upper < :upper
```

Let's replace the joined table by a table expression, using our precomputed node class queries. Notice that topLeft and bottomLeft together form leftNodes. The query is:

```
SELECT id
FROM   Intervals i
JOIN   (SELECT :lower & b1 AS node
        FROM   BitMasks
        WHERE  :lower & b2 <> 0
          AND  :lower & b1 <> 0) q
  ON   i.node = q.node
WHERE  i.upper > :lower
  AND  i.upper < :upper

UNION ALL

SELECT id
FROM   Intervals i
JOIN   (SELECT (:lower & b1) | b3 AS node
        FROM   BitMasks
        WHERE  :lower & b3 = 0
          AND  :lower & b1 <> :lower
          AND  (:lower & b1) | b3 < :fork
        UNION ALL
        (SELECT :lower UNION SELECT :fork)
       ) q
  ON   i.node  = q.node
WHERE  i.lower < :lower
  AND  i.upper > :lower
  AND  i.upper < :upper
```

FinishedBy

The original query is:

```
SELECT id
FROM   Intervals i
JOIN   :(topLeft U fork) q
  ON   i.node  = q.node
WHERE  i.upper = :upper
  AND  i.lower < :lower
```

Let's replace the joined table by a table expression using our precomputed node class queries. The query is:

```
SELECT id
FROM   Intervals i
JOIN   (SELECT :fork & b1 AS node
        FROM   BitMasks
        WHERE  :fork & b2 <> 0
          AND  :fork & b1 <> 0
        UNION ALL
        SELECT :fork) q
  ON   i.node  = q.node
WHERE  i.upper = :upper
  AND  i.lower < :lower
```

Starts

The original query is:

```
SELECT id
FROM   Intervals i
JOIN   :(innerLeft U lower U fork) q
  ON   i.node  = q.node
WHERE  i.lower = :lower
  AND  i.upper < :upper
```

Let's replace the joined table by a table expression using our precomputed node class queries. The query is:

```
SELECT id
FROM   Intervals i
JOIN   (SELECT (:lower & b1) | b3 AS node
        FROM   BitMasks
        WHERE  :lower & b3 = 0
          AND  :lower & b1 <> :lower
          AND  (:lower & b1) | b3 < :fork
        UNION ALL
        (SELECT :lower
         UNION
         SELECT :fork)
       ) q
  ON   i.node  = q.node
WHERE  i.lower = :lower
  AND  i.upper < :upper
```

Contains

The original query is:

```
SELECT id
FROM   Intervals i
JOIN   :(topRight U fork) q
  ON   i.node = q.node
WHERE  i.lower < :lower
  AND  i.upper > :upper
UNION ALL
SELECT id
FROM   Intervals i
JOIN   topLeft q
  ON   i.node = q.node
WHERE  i.upper > :upper
```

Let's replace the joined table by a table expression using our precomputed node class queries. The query is:

```
SELECT id
FROM   Intervals i
JOIN   (SELECT (:fork & b1) | b3 AS node
        FROM BitMasks
        WHERE  :fork & b3 = 0
          AND  :fork & b1 <> :fork
        UNION ALL
        SELECT :fork) q
  ON   i.node = q.node
WHERE  i.lower < :lower
  AND  i.upper > :upper
UNION ALL
SELECT id
FROM   Intervals i
JOIN   (SELECT :fork & b1 AS node
        FROM   BitMasks
        WHERE  :fork & b2 <> 0
          AND  :fork & b1 <> 0
       ) q
  ON   i.node = q.node
WHERE  i.upper > :upper
```

Equals

The original query is:

```
SELECT id
FROM   Intervals i
WHERE  i.node  = :fork
  AND  i.lower = :lower
  AND  i.upper = :upper
```

This query does not make use of the node classes, so there's nothing to rewrite; we'll therefore use the query as it is.

During

The original query is:

```
SELECT id
FROM   Intervals i
WHERE  i.node  > :lower
  AND  i.node <= :fork
  AND  i.lower > :lower
  AND  i.upper < :upper
UNION ALL
SELECT id
FROM   Intervals i
WHERE  i.node  > :fork
  AND  i.node  < :upper
  AND  i.upper < :upper
```

This query does not make use of the node classes, so there's nothing to rewrite; we'll therefore use the query as it is.

StartedBy

The original query is:

```
SELECT id
FROM   Intervals i
JOIN   :(topRight U fork) q
  ON   i.node  = q.node
WHERE  i.lower = :lower
  AND  i.upper > :upper
```

Let's replace the joined table by a table expression using our precomputed node class queries. The query is:

```
SELECT id
FROM   Intervals i
JOIN   (SELECT (:fork & b1) | b3 AS node
        FROM   BitMasks
        WHERE  :fork & b3 = 0
          AND  :fork & b1 <> :fork
        UNION ALL
        SELECT :fork
       ) q
  ON   i.node  = q.node
WHERE  i.lower = :lower
  AND  i.upper > :upper
```

Finishes

The original query is:

```
SELECT id
FROM   Intervals i
JOIN   :(innerRight U upper U fork) q
  ON   i.node = q.node
WHERE  i.upper = :upper
  AND  i.lower > :lower
```

Let's replace the joined table by a table expression using our precomputed node class queries. The query is:

```
SELECT id
FROM   Intervals i
JOIN   (SELECT :upper & b1 AS node
        FROM   BitMasks
        WHERE  :upper & b2 <> 0
          AND  :upper & b1 <> 0
          AND  :upper & b1 > :fork
         UNION ALL
         (SELECT :upper
          UNION
          SELECT :fork
         )
       ) q
  ON   i.node  = q.node
WHERE  i.upper = :upper
  AND  i.lower > :lower
```

OverlappedBy

The original query is:

```
SELECT id
FROM   Intervals i
JOIN   :(topRight U bottomRight) q
  ON   i.node = q.node
WHERE  i.lower > :lower
  AND  i.lower < :upper
UNION ALL
SELECT id
FROM   Intervals i
JOIN   :(innerRight U upper U fork) q
  ON   i.node = q.node
WHERE  i.upper > :upper
  AND  i.lower > :lower
  AND  i.lower < :upper
```

Let's replace the joined table by a table expression using our precomputed node class queries. Note that topRight and bottomRight together form rightNodes. The query is:

```
SELECT id
FROM   Intervals i
JOIN   (SELECT (:upper & b1) | b3 AS node
        FROM   BitMasks
        WHERE  :upper & b3 = 0
          AND  :upper & b1 <> :upper
       ) q
  ON   i.node = q.node
WHERE  i.lower > :lower
  AND  i.lower < :upper
UNION ALL
SELECT id
FROM   Intervals i
JOIN   (SELECT :upper & b1 AS node
        FROM   BitMasks
        WHERE  :upper & b2 <> 0
          AND  :upper & b1 <> 0
          AND  :upper & b1 > :fork
         UNION ALL
         (SELECT :upper
          UNION
          SELECT :fork
         )
       ) q
  ON   i.node  = q.node
WHERE  i.upper > :upper
  AND  i.lower > :lower
  AND  i.lower < :upper
```

MetBy

The original query is:

```
SELECT id
FROM   Intervals i
JOIN   :(topRight U bottomRight U upper) q
  ON   i.node  = q.node
WHERE  i.lower = :upper
```

Let's replace the joined table by a table expression using our precomputed node class queries. Since topRight and bottomRight together form rightNodes, the query is:

```
SELECT id
FROM   Intervals i
JOIN   (SELECT (:upper & b1) | b3 AS node
        FROM   BitMasks
        WHERE  :upper & b3 = 0
          AND  :upper & b1 <> :upper
        UNION ALL
        SELECT :upper
       ) q
  ON   i.node = q.node
WHERE  i.lower = :upper
```

After

The original query is:

```
SELECT id
FROM   Intervals
WHERE  node  > :upper
  AND  lower > :upper
```

This query does not make use of the node classes, so there's nothing to rewrite; we'll therefore use the query as it is.

About query performance

The performance of the Static RI-Tree queries, although generally very good, is not excellent for all query types.

Some queries exhibit excellent performance, similar to that of the intersection query, because they only scan useful portions of lowerIndex or upperIndex (the indexes we had created for the intersection query). Kriegel, Pötke and Seidl call this behavior *blocked* index scan and they call the opposite behavior, where useless portions of the index are being traversed, *non-blocked* index scan.

Other queries can achieve blocked index scans, provided lowerIndex and upperIndex are turned into the richer indexes upperLowerIndex and lowerUpperIndex, as described in Kriegel, Pötke and Seidl's paper.

Other queries expose non-blocked index scans, whatever index you create.

Let's create the richer indexes upperLowerIndex and lowerUpperIndex, as shown in listing 4, in order to optimize as many of the Static RI-Tree queries as we can.

Listing 4: Creating the richer indexes upperLowerIndex and lowerUpperIndex

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

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

The well-optimized queries are: meets, finishedBy, starts, equals, startedBy, finishes and metBy. Among these, some don't need the rich indexes: meets only needs upperIndex and metBy only needs lowerIndex. But it's unlikely that your application only uses the meets and metBy relationship, so you're better off sticking with the rich indexes.

The other queries only partly benefit from the indexes, with non-blocked index scans. However, this partial use, combined with node class selectivity, should generally provide good performance.

In the next section, we discuss the implementation of the Static RI-Tree queries for Microsoft SQL Server 2005 and above. The significant performance gains obtained with our set-based approach versus the iterative approach are illustrated by a performance comparison test script.

Note that the number of iterations in the original RI-Tree can be reduced by the mechanism of dynamic expansion, but we're really moving away from the iterative approach because we don't want to manage the RI-Tree's four parameters - offset, leftRoot, rightRoot and minstep - for the reasons explained in my previous article, and we don't want an iterative approach with loops. Fortunately, the range optimization presented above is a good alternative to dynamic expansion.

Writing the Static RI-Tree queries for Microsoft SQL Server

In this section, let's write the Static RI-Tree queries for Microsoft SQL Server.

We'll assume that the actual intervals do not cover the full range of values. Therefore, we'll use the full range optimization.

Note that, once created and filled, the BitMasks table is always accessed in a read-only fashion, so we can safely use a NOLOCK table hint.

Creating the sample Intervals table

The Intervals table can be created, along with its indexes, as shown in Listing 5. It can be filled with sample data as was demonstrated in my previous article.

Listing 5: Creating the Intervals table

```
CREATE TABLE dbo.Intervals
(
  id INT NOT NULL PRIMARY KEY,
  node AS upper - upper % POWER(2, FLOOR(
   LOG( (lower-1) ^ upper)/LOG(2)))
   PERSISTED NOT NULL,
  lower INT NOT NULL,
  upper INT NOT NULL
);
CREATE UNIQUE INDEX lowerUpperIndex
ON dbo.Intervals(node, lower, upper, id);
CREATE UNIQUE INDEX upperLowerIndex
ON dbo.Intervals(node, upper, lower, id);
```

Creating the BitMasks table

To create and populate the BitMasks table, you can reuse the code from the previous article.

Creating functions for the node classes

Let's create functions to wrap the node classes. This will promote code reuse and simplify the final queries. The best choice for these functions is to implement them as inline table-valued, because SQL Server inlines their code upon execution, and thus we avoid the cost of invoking a function.

topLeft

The topLeft node class can be implemented by the following function:

```
CREATE FUNCTION dbo.TopLeft(@fork INT)
RETURNS TABLE
AS
RETURN
  SELECT  @fork & b1 AS node
  FROM    dbo.BitMasks WITH (NOLOCK)
  WHERE   @fork & b2 <> 0
    AND   @fork & b1 <> 0;
```

topRight

The topRight node class can be implemented by the following function:

```
CREATE FUNCTION dbo.TopRight(@fork INT)
RETURNS TABLE
AS
RETURN
  SELECT  (@fork & b1) | b3 AS node
  FROM    dbo.BitMasks WITH (NOLOCK)
  WHERE   @fork & b3 = 0
    AND   @fork & b1 <> @fork;
```

innerLeft

The innerLeft node class can be implemented by the following function:

```
CREATE FUNCTION dbo.InnerLeft(@lower INT,
  @fork INT)
RETURNS TABLE
AS
RETURN
  SELECT  (@lower & b1) | b3 AS node
  FROM    dbo.BitMasks WITH (NOLOCK)
  WHERE   @lower & b3 = 0
    AND   @lower & b1 <> @lower
    -- Excluding nodes to the right of the
    -- fork node:
    AND   (@lower & b1) | b3 < @fork;
```

innerRight

The innerRight node class can be implemented by the following function:

```
CREATE FUNCTION dbo.InnerRight(@upper INT,
  @fork INT)
RETURNS TABLE
AS
RETURN
  SELECT  @upper & b1 AS node
  FROM    dbo.BitMasks WITH (NOLOCK)
  WHERE   @upper & b2 <> 0
    AND   @upper & b1 <> 0
    -- Excluding nodes to the left of the
    -- fork node:
    AND   @upper & b1 > @fork;
```

leftNodes

The leftNodes node class represents the union of the topLeft and bottomLeft node classes.

It can be implemented by the following function:

```
CREATE FUNCTION dbo.LeftNodes(@lower INT)
RETURNS TABLE
AS
RETURN
  SELECT  @lower & b1 AS node
  FROM    dbo.BitMasks WITH (NOLOCK)
  WHERE   @lower & b2 <> 0
    AND   @lower & b1 <> 0;
```

rightNodes

The rightNodes node class represents the union of the topRight and bottomRight node classes.

It can be implemented by the following function:

```
CREATE FUNCTION dbo.RightNodes(@upper INT)
RETURNS TABLE
AS
RETURN
  SELECT  (@upper & b1) | b3 AS node
  FROM    dbo.BitMasks WITH (NOLOCK)
  WHERE   @upper & b3 = 0
    AND   @upper & b1 <> @upper;
```

fork

The fork node class can be implemented by the following function:

```
CREATE FUNCTION dbo.Fork(@lower INT,
  @upper INT)
  RETURNS INT
AS
BEGIN
  RETURN @upper - @upper %
    POWER(2, FLOOR(LOG((@lower - 1) ^
     @upper) / LOG(2)));
END
```

Notice that this is a scalar function and not a table-valued function, as are the other functions.

As I explained in my previous article, the Fork function should not be called from the definition of the computed node column in the Intervals table, because this would significantly impact performance. In this particular situation, we prefer an inline formula. However, it's perfectly OK to invoke the function once during the execution of one of the Static RI-Tree queries.

Writing the Static RI-Tree queries

In this section, we'll use the node class functions to write the Static RI-Tree queries.

Notes:

- In the following queries, you can add a MAXDOP query hint to help save some precious CPU cycles when a parallel plan is unnecessary. Just append “OPTION (MAXDOP 1)” to the query.
- Full range optimizations are used when appropriate, via the @min and @max variables.
- The intersection query, although not strictly part of Allen’s 13 interval relationships, is one of the most useful.

Relationship: Intersection

Query:

```
DECLARE @lower INT = 826216,
        @upper INT = 826254,
        @min   INT,
        @max   INT;
SELECT  @min = MIN(node), @max = MAX(node) FROM dbo.Intervals;

SELECT  id
FROM    dbo.Intervals i
JOIN    dbo.LeftNodes(@lower) ln
    ON  i.node    = ln.node
    AND i.upper >= @lower
WHERE   ln.node >= @min

UNION ALL

SELECT  id
FROM    dbo.Intervals i
JOIN    dbo.RightNodes(@upper) rn
    ON  i.node = rn.node
    AND i.lower <= @upper
WHERE   rn.node <= @max

UNION ALL

SELECT  id
FROM    dbo.Intervals
WHERE   node BETWEEN @lower AND @upper;
```

Relationship: Before

Query:

```
DECLARE @lower INT = 826217,
        @min   INT = (SELECT MIN(node) FROM dbo.Intervals);

SELECT  id
FROM    dbo.Intervals
WHERE   node  < @lower
    AND upper < @lower
    AND node >= @min;
```

Relationship: Meets

Query:

```
DECLARE @lower INT = 826217,
        @upper INT = 826253,
        @min   INT = (SELECT MIN(node) FROM dbo.Intervals);

SELECT  *
FROM    dbo.Intervals i
JOIN    (
            SELECT  node
            FROM    dbo.LeftNodes(@lower)
            WHERE   node >= @min
            UNION ALL
            SELECT  @lower
        ) q
  ON    i.node  = q.node
  AND   i.upper = @lower;
```

Relationship	Query

Overlaps

```
DECLARE @lower INT = 826217,
        @upper INT = 826253,
        @min   INT = (SELECT MIN(node) FROM dbo.Intervals);
DECLARE @fork  INT = dbo.Fork(@lower, @upper);

SELECT  id
FROM    dbo.Intervals i
JOIN    dbo.LeftNodes(@lower) q
    ON  i.node = q.node
WHERE   i.upper > @lower
    AND i.upper < @upper
    AND q.node >= @min

UNION ALL

SELECT  id
FROM    dbo.Intervals i
JOIN    (
            SELECT  node
            FROM    dbo.InnerLeft(@lower, @fork)
            UNION ALL
            (
                SELECT  @lower
                UNION
                SELECT  @fork
            )
        ) q
    ON  i.node  = q.node
WHERE   i.lower < @lower
    AND i.upper > @lower
    AND i.upper < @upper;
```

FinishedBy

```
DECLARE @lower INT = 826240,
        @upper INT = 826253,
        @min   INT = (SELECT MIN(node) FROM dbo.Intervals);
DECLARE @fork  INT = dbo.Fork(@lower, @upper);

SELECT  id
FROM    dbo.Intervals i
JOIN    (
            SELECT  node
            FROM    dbo.TopLeft(@fork)
            WHERE   node >= @min
            UNION ALL
            SELECT  @fork
        ) q
    ON  i.node  = q.node
WHERE   i.upper = @upper
    AND i.lower < @lower;
```

Starts

```
DECLARE @lower INT = 826240,
        @upper INT = 826253;
DECLARE @fork  INT = dbo.Fork(@lower, @upper);

SELECT  id
FROM    dbo.Intervals i
JOIN    (
            SELECT  node
            FROM    dbo.InnerLeft(@lower, @fork)
            UNION ALL
            (
                SELECT  @fork
                UNION
                SELECT @lower
            )
        ) q
  ON   i.node  = q.node
WHERE  i.lower = @lower
  AND  i.upper < @upper;
```

Relationship	Query

Contains

```
DECLARE @lower INT = 826240,
        @upper INT = 826253,
        @min   INT,
        @max   INT;
DECLARE @fork  INT = dbo.Fork(@lower, @upper);
SELECT  @min = MIN(node), @max = MAX(node) FROM dbo.Intervals;

SELECT  id
FROM    dbo.Intervals i
JOIN    (
            SELECT  node
            FROM    dbo.TopRight(@fork)
            WHERE   node <= @max
            UNION ALL
            SELECT  @fork
        ) q
    ON  i.node = q.node
WHERE   i.lower < @lower
    AND i.upper > @upper
UNION ALL
SELECT  id
FROM    dbo.Intervals i
JOIN    dbo.TopLeft(@fork) q
    ON  i.node  = q.node
WHERE   i.upper > @upper
    AND q.node  >= @min;
```

Equals

```
DECLARE @lower INT = 826240,
        @upper INT = 826253;
DECLARE @fork  INT = dbo.Fork(@lower, @upper);

SELECT  id
FROM    dbo.Intervals i
WHERE   i.node  = @fork
    AND i.lower = @lower
    AND i.upper = @upper;
```

During

```
DECLARE @lower INT = 826240,
        @upper INT = 826253,
        @min   INT,
        @max   INT;
DECLARE @fork  INT = dbo.Fork(@lower, @upper);
SELECT  @min = MIN(node), @max = MAX(node) FROM dbo.Intervals;

SELECT  id
FROM    dbo.Intervals i
WHERE   i.node  > @lower
    AND i.node >= @min
    AND i.node <= @fork
    AND i.lower > @lower
    AND i.upper < @upper
UNION ALL
SELECT  id
FROM    dbo.Intervals i
WHERE   i.node  > @fork
    AND i.node <= @max
    AND i.node  < @upper
    AND i.upper < @upper;
```

<table>
<tr><th>Relationship</th><th>Query</th></tr>
<tr><td>StartedBy</td><td><pre>DECLARE @lower INT = 826240,
        @upper INT = 826253,
        @max   INT = (SELECT MAX(node) FROM dbo.Intervals);
DECLARE @fork INT = dbo.Fork(@lower, @upper);

SELECT id
FROM   dbo.Intervals i
JOIN   (
            SELECT node
            FROM   dbo.TopRight(@fork)
            WHERE   node <= @max
            UNION ALL
            SELECT @fork
) q
    ON i.node = q.node
WHERE   i.lower = @lower
    AND i.upper > @upper;</pre></td></tr>
<tr><td>Finishes</td><td><pre>DECLARE @lower INT = 826240,
        @upper INT = 826253;
DECLARE @fork INT = dbo.Fork(@lower, @upper);

SELECT id
FROM   dbo.Intervals i
JOIN   (
            SELECT node
            FROM   dbo.InnerRight(@upper, @fork)
            UNION ALL
            (
                SELECT @upper
                UNION
                SELECT @fork
)
) q
    ON i.node = q.node
WHERE   i.upper = @upper
    AND i.lower > @lower;</pre></td></tr>
<tr><td>OverlappedBy</td><td><pre>DECLARE @lower INT = 826240,
        @upper INT = 826253,
        @max   INT = (SELECT MAX(node) FROM dbo.Intervals);
DECLARE @fork INT = dbo.Fork(@lower, @upper);

SELECT id
FROM   dbo.Intervals i
JOIN   (
            SELECT node
            FROM   dbo.RightNodes(@upper)
            WHERE   node <= @max
) q
    ON i.node = q.node
WHERE   i.lower > @lower
    AND i.lower < @upper
UNION ALL
SELECT id
FROM   dbo.Intervals i
JOIN   (
            SELECT node
            FROM   dbo.InnerRight(@upper, @fork)
            UNION ALL
            (
                SELECT @upper
                UNION
                SELECT @fork
)
) q
    ON i.node = q.node
WHERE   i.upper > @upper
    AND i.lower > @lower
    AND i.lower < @upper;</pre></td></tr>
</table>

Relationship	Query

MetBy	`DECLARE @lower INT = 826240,`
	`        @upper INT = 826253,`
	`        @max   INT = (SELECT MAX(node) FROM dbo.Intervals);`
	
	`SELECT  id`
	`FROM    dbo.Intervals i`
	`JOIN    (`
	`            SELECT  node`
	`            FROM    dbo.RightNodes(@upper)`
	`            WHERE   node <= @max`
	`            UNION ALL`
	`            SELECT  @upper`
	`        ) q`
	`    ON  i.node  = q.node`
	`WHERE   i.lower = @upper;`

After	`DECLARE @lower INT = 826240,`
	`        @upper INT = 826253,`
	`        @max   INT = (SELECT MAX(node) FROM dbo.Intervals);`
	
	`SELECT  id`
	`FROM    dbo.Intervals`
	`WHERE   node  > @upper`
	`    AND lower > @upper`
	`    AND node <= @max;`

Performance comparison test

In order to compare the performance of queries written with iterative node class logic with those written with set-based node class logic, let's write iterative versions of the node class functions.

topLeft

The topLeft node class can be implemented by the following function:

```
CREATE FUNCTION dbo.TopLeftIterative(@lower
    INT, @upper INT)
  RETURNS @topleft TABLE(node INT NOT NULL
    PRIMARY KEY)
AS
BEGIN
  -- root = 2 ^ 30:
  DECLARE @n    INT = 1073741824;
  DECLARE @step INT = @n/2;

  -- Descend from the root node to the
  -- fork node
  WHILE @step >= 1
  BEGIN
    IF @n < @lower
    BEGIN
      INSERT @topleft(node) VALUES(@n);
      SET @n += @step;
    END
    ELSE IF @upper < @n
      SET @n -= @step;
    ELSE
      BREAK; -- fork node
    SET @step /= 2;
  END
  RETURN;
END
```

topRight

The topRight node class can be implemented by the following function:

```
CREATE FUNCTION dbo.TopRightIterative(
    @lower INT, @upper INT)
  RETURNS @topright TABLE(node INT NOT NULL
    PRIMARY KEY)
AS
BEGIN
  -- root = 2 ^ 30:
  DECLARE @n    INT = 1073741824;
  DECLARE @step INT = @n/2;

  -- Descend from the root node to the
  -- fork node
  WHILE @step >= 1
  BEGIN
    IF @upper < @n
    BEGIN
      INSERT @topright(node) VALUES(@n);
      SET @n -= @step;
    END
    ELSE IF @n < @lower
      SET @n += @step;
    ELSE
      BREAK; -- fork node

    SET @step /= 2;
  END

  RETURN;
END
```

innerLeft

The innerLeft node class can be implemented by the following function:

```
CREATE FUNCTION dbo.InnerLeftIterative(
    @lower INT, @upper INT)
  RETURNS @innerleft TABLE(node INT
    NOT NULL PRIMARY KEY)
AS
BEGIN
  -- root = 2 ^ 30:
  DECLARE @n    INT = 1073741824;
  DECLARE @step INT = @n/2;
  DECLARE @fork INT;

  -- Descend from the root node to the
  -- fork node
  WHILE @step >= 1
  BEGIN
    IF @upper < @n
      SET @n -= @step;
    ELSE IF @n < @lower
      SET @n += @step;
    ELSE
    BEGIN
      SET @fork = @n;
      BREAK; -- fork node
    END

    SET @step /= 2;
  END

  -- Descend from the fork node to lower
  IF @lower < @fork
  BEGIN
    SET @n = @fork - @step;
    DECLARE @lstep INT = @step / 2;

    WHILE @lstep >= 1
    BEGIN
      IF @n > @lower
      BEGIN
        INSERT @innerleft(node) VALUES(@n);
        SET @n -= @lstep;
      END
      ELSE IF @n < @lower
        SET @n += @lstep;
      ELSE
        BREAK; -- lower node

      SET @lstep /= 2;
    END
  END
  RETURN;
END
```

innerRight

The innerRight node class can be implemented by the following function:

```
CREATE FUNCTION dbo.InnerRightIterative(
    @lower INT, @upper INT)
  RETURNS @innerright TABLE(node INT
    NOT NULL PRIMARY KEY)
AS
BEGIN
  -- root = 2 ^ 30:
  DECLARE @n    INT = 1073741824;
  DECLARE @step INT = @n/2;
  DECLARE @fork INT;

  -- Descend from the root node to the
  -- fork node
  WHILE @step >= 1
  BEGIN
    IF @upper < @n
      SET @n -= @step;
```

```
      SET @n += @step;
    ELSE
    BEGIN
      SET @fork = @n;
      BREAK; -- fork node
    END

    SET @step /= 2;
  END

  -- Descend from the fork node to upper
  IF @upper > @fork
  BEGIN
    SET @n = @fork + @step;
    DECLARE @rstep INT = @step / 2;

    WHILE @rstep >= 1
    BEGIN
      IF @n < @upper
      BEGIN
        INSERT @innerright(node)VALUES(@n);
        SET @n += @rstep;
      END
      ELSE IF @n > @upper
        SET @n -= @rstep;
      ELSE
        BREAK; -- upper node

      SET @rstep /= 2;
    END
  END
  RETURN;
END
```

leftNodes

The leftNodes node class represents the union of the topLeft and bottomLeft node classes.

It can be implemented by the following function:

```
CREATE                                  FUNCTION
dbo.LeftNodesIterative(@lower INT,
    @upper INT)
  RETURNS @leftnodes TABLE(node INT
    NOT NULL PRIMARY KEY)
AS
BEGIN
  -- root = 2 ^ 30:
  DECLARE @n    INT = 1073741824;
  DECLARE @step INT = @n/2;
  DECLARE @fork INT;

  -- Descend from the root node to the
  -- fork node:
  WHILE @step >= 1
  BEGIN
    IF @n < @lower
    BEGIN
      INSERT @leftnodes(node) VALUES(@n);
      SET @n += @step;
    END
    ELSE IF @upper < @n
      SET @n -= @step;
    ELSE
    BEGIN
      SET @fork = @n;
      BREAK; -- fork node
    END
    SET @step /= 2;
  END
```

```
  -- Descend from the fork node to lower
  IF @lower < @fork
  BEGIN
    SET @n = @fork - @step;
    DECLARE @lstep INT = @step / 2;

    WHILE @lstep >= 1
    BEGIN
      IF @n < @lower
      BEGIN
        INSERT @leftnodes(node) VALUES(@n);
        SET @n += @lstep;
      END
      ELSE IF @n > @lower
        SET @n -= @lstep;
      ELSE
        BREAK; -- lower node

      SET @lstep /= 2;
    END
  END

  RETURN;
END
```

rightNodes

The rightNodes node class represents the union of the topRight and bottomRight node classes.

It can be implemented by the following function:

```
CREATE FUNCTION
dbo.RightNodesIterative(@lower INT,
    @upper INT)
  RETURNS @rightnodes TABLE(node INT
    NOT NULL PRIMARY KEY)
AS
BEGIN
  -- root = 2 ^ 30:
  DECLARE @n    INT = 1073741824;
  DECLARE @step INT = @n/2;
  DECLARE @fork INT;

  -- Descend from the root node to the
  -- fork node:
  WHILE @step >= 1
  BEGIN
    IF @n < @lower
      SET @n += @step;
    ELSE IF @upper < @n
    BEGIN
      INSERT @rightnodes(node) VALUES(@n);
      SET @n -= @step;
    END
    ELSE
    BEGIN
      SET @fork = @n;
      BREAK; -- fork node
    END

    SET @step /= 2;
  END

  -- Descend from the fork node to upper
  IF @fork < @upper
  BEGIN
    SET @n = @fork + @step;
    DECLARE @rstep INT = @step / 2;

    WHILE @rstep >= 1
    BEGIN
      IF @n < @upper
        SET @n += @rstep;
      ELSE IF @upper < @n
      BEGIN
        INSERT @rightnodes(node)VALUES(@n);
        SET @n -= @rstep;
      END
      ELSE
        BREAK; -- upper node

      SET @rstep /= 2;
    END
  END

  RETURN;
END
```

fork

The fork node class can be implemented by the following function:

```
CREATE FUNCTION dbo.ForkIterative(@lower
    INT, @upper INT)
  RETURNS INT
AS
BEGIN
  -- root = 2 ^ 30:
  DECLARE @n    INT = 1073741824;
  DECLARE @step INT = @n/2;

  -- Descend from the root node to the
  -- fork node:
  WHILE @step >= 1
  BEGIN
    IF @upper < @n
      SET @n -= @step;
    ELSE IF @n < @lower
      SET @n += @step;
    ELSE
      BREAK; -- fork node reached

    SET @step /= 2;
  END

  RETURN @n;
END
```

Test results

The test script executes each Static RI-Tree query involving the node class functions in both the iterative and set-based versions. To magnify the measures, it runs each query 1,000 times in a loop. Also, instead of returning the results, it just counts the rows, in order to avoid overloading the query editor window with result sets.

Below is an excerpt of the script involving the StartedBy query:

```
-- Iteration-based StartedBy
DECLARE @lower INT = 826240,
        @upper INT = 826253;
DECLARE @max   INT = (SELECT MAX(node)
  FROM dbo.Intervals);

DECLARE @i INT = 1000, @cnt INT;
WHILE @i > 0
BEGIN
  SELECT @cnt = COUNT(*)
  FROM
  (
    SELECT  id
    FROM    dbo.Intervals i
    JOIN    (SELECT  node
             FROM dbo.TopRightIterative
               (@lower, @upper)
             UNION ALL
             SELECT dbo.ForkIterative
               (@lower, @upper)
            ) q
      ON    i.node  = q.node
    WHERE   i.lower = @lower
      AND   i.upper > @upper
      -- Range optimization:
      AND q.node <= @max
  ) T;

  SET @i -=1;
END
GO

-- Set-based StartedBy
DECLARE @lower INT = 826240,
        @upper INT = 826253;
DECLARE @max   INT = (SELECT MAX(node)
  FROM dbo.Intervals);
DECLARE @fork INT = dbo.Fork(@lower,
  @upper);

DECLARE @i INT = 1000, @cnt INT;
WHILE @i > 0
BEGIN
  SELECT @cnt = COUNT(*)
  FROM
  (SELECT  id
   FROM    dbo.Intervals i
   JOIN    (SELECT  node
            FROM    dbo.TopRight(@fork)
            UNION ALL
            SELECT  @fork
           ) q
     ON    i.node  = q.node
   WHERE   i.lower = @lower
     AND   i.upper > @upper
     -- Range optimization:
     AND   q.node <= @max
  ) T;

  SET @i -=1;
END
GO
```

Note that for some queries, I had to add a MAXDOP 1 query option, in order to prevent SQL Server from using a parallel query plan when it turned out to be more costly in CPU time.

I ran this test script on my laptop, which is equipped with an Intel® Core™ 2 Duo P7450 / 2.13 GHz processor and 4GB of RAM. The SQL Server version is 2008 R2 SP1 Developer Edition 64-bit, running on Windows 7 Family Edition Premium SP1 64-bit. The results are shown in Table 2. As you can see, using the set-based approach is significantly more efficient.

Query	Type	Cpu	Elapsed	Logical Reads
Meets	Iterative	390	387	57000
	Set	62	76	37000
Overlaps	Iterative	593	816	79000
	Set	187	188	63000
FinishedBy	Iterative	546	542	46000
	Set	266	263	32000
Starts	Iterative	171	169	6000
	Set	47	45	6000
Contains	Iterative	733	768	92003
	Set	172	167	53003
StartedBy	Iterative	499	503	53003
	Set	62	65	21003
Finishes	Iterative	250	244	19000
	Set	62	59	15000
OverlappedBy	Iterative	718	746	89003
	Set	156	165	54003
MetBy	Iterative	530	556	73003
	Set	110	99	33003

Table 2: Results of the performance comparison test script between iterative and set-based interval queries

About the Author

Laurent Martin (laurent.martin741@yahoo.fr) is a software architect working in Paris, France, for StatPro (www.statpro.com), a leading portfolio analysis and asset valuation provider. Laurent has been working in the software industry for over 20 years, specializing in Microsoft technologies.