By Laurent Martin

Using the Statistic Relational Interval Tree with time intervals

In my previous article, "A Static Relational Interval Tree" and "Advanced interval queries with the Static Relational Interval Tree", I explained how to manage intervals and query them in an efficient way. One fundamental property of the intervals I've been discussing so far is that they have integer boundaries. But what if in your situation, what you need to manage are time intervals? In this article, I present solutions to implement Static RI-Trees containing time intervals.

The Static RI-Tree is based on integers because its virtual backbone is a binary tree whose nodes and leaves are positive integers. While it would be possible to use another data type instead, in practice the advantage of our implementation lies in an efficient use of integer arithmetic, including bitwise operations, to enable fast bulk insertions and interval querying. In order to keep using integers while managing time intervals, what we need is to introduce a time-to-integer mapping, which we'll discuss in the first part of this article, with its constraints and complexities. Our focus will be on the DATE and DATETIME2 data types. We'll examine mappings implemented in T-SQL as well as in CLR user-defined functions. Next, we'll compare the performance of both implementations.

The second part of this article is dedicated to querying. It exposes the required adaptations to the interval queries and support objects in order to deal with DATE and DATETIME2 data types. Finally, we'll list the full query catalog for Static RI-Trees of time intervals.

Part 1: Time-to-integer mappings

Time-to-integer mapping properties

To define a time-to-integer mapping, we need to comply with the following rules:

1. Each time value must map to exactly one integer.
2. Each integer must map to exactly one time value.
3. For each pair of time values (t1, t2), such that t1 maps to i1 and t2 maps to i2, if t1 < t2 then i1 < i2, and conversely, if i1 < i2 then t1 < t2. This means that the mapping must conserve the order of values.

The implementation of such bijections depends on the DBMS. In the remainder of this article, we shall focus on time-to-integer mappings for Microsoft SQL Server 2008 and above.

A Static RI-Tree containing DATE values

In Microsoft SQL Server 2008, the DATE data type enables the user to represent dates ranging from January 1st, 0001 through December 31st, 9999.

Mapping Date to INT

Before defining a mapping, note that we cannot use 0 as the lowest mapped integer because the Static RI-Tree implementation does not support it: the fork node computation will report a floating point error.

To map dates to integers, let's decide we map January 1st, 0001 to 1, January 2nd, 0001 to 2, and so on. To implement this mapping, one natural solution is to use the following expression:

```
DATEDIFF(d, '00010101', @d) + 1
```

Here, @d is a variable of type DATE holding the date we wish to map. Note that the string '00010101' represents the lower bound January 1st, 0001, in ISO 8601 format (yyyymmdd). A good practice is to always specify date literals using this format, because SQL Server will always interpret them the same way, independently of the SET LANGUAGE and SET DATEFORMAT session settings. An alternative syntax for this format is yyyy-mm-dd. The DATEDIFF expression above computes the number of days between January 1st, 0001 and @d, which is exactly what we need for the mapping if we just add one to the result. When @d is set to '99991231', the upper bound for the DATE data type, the expression evaluates to 3652059, which can be represented with the INT data type. In fact, the DATEDIFF function returns an INT value. To map an INT whose value is less than or equal to 3652059 to a DATE, we can use the following expression:

```
DATEADD(d, @i-1, CAST('00010101' AS DATE))
```

Here, @i is the integer to map. Note that we cannot use:

```
DATEADD(d, @i-1, '00010101')
```

because SQL Server would try to implicitly convert the varchar value '00010101' to a DATETIME, which would result in an error since the lowest DATETIME value is January 1st, 1753.

Deterministic expressions

Unfortunately, things are a little bit more complicated for our mapping because what we need is an expression that we'll use as an inline formula for a persisted computed column, and SQL Server requires such an expression to be deterministic. This means that the expression must always return the same result every time it gets evaluated with a specific set of input values and given the same state of the database.

When parsing the expression:

```
DATEDIFF(d, '00010101', @d) + 1
```

SQL Server implicitly converts the varchar string '00010101' to a DATETIME, then to a DATE. This pair of implicit conversions is considered non-deterministic, even though we're using the ISO 8601 format. The expression can be made deterministic like so:

```
DATEDIFF(d, CONVERT(DATE, '00010101', 112),
@d) + 1
```

The style 112 explicitly states that the varchar string is to be interpreted as yyyymmdd.

Similarly, the deterministic equivalent of the INT-to-DATE mapping is:

```
DATEADD(d, @i-1, CONVERT(DATE, '00010101',
112))
```

To check whether an expression is deterministic, a simple way is to use it as the inline formula of a computed persisted column within a table variable:

```
DECLARE @T AS TABLE
(
  dt     DATE NOT NULL,
  int_dt AS DATEDIFF(d, '00010101', dt) + 1
    PERSISTED NOT NULL
);
```

When executed, the statement above fails with the following message: "Msg 4936, Level 16, State 1, Line 1. Computed column 'int_dt' in table '@T' cannot be persisted because the column is non-deterministic". The following, however, should succeed:

```
DECLARE @T AS TABLE
(
  dt     DATE NOT NULL,
  int_dt AS DATEDIFF(d,
    CONVERT(DATE, '00010101', 112), dt) + 1
    PERSISTED NOT NULL
);
```

The IntervalsDate table

Let's use our mapping to create the IntervalsDate table, which is a container of DATE intervals structured as a Static RI-Tree. As a reminder, here is the inline formula we used before for INT intervals (see my previous article "A Static Relational Interval Tree"):

```
upper - upper % POWER(2, FLOOR(
   LOG( (lower-1) ^ upper)/LOG(2)))
```

This formula computes the fork node of the interval [lower, upper]. Assuming that lower and upper are now DATE values, let's replace them by their mapping to INT values and map the result back to DATE, ensuring the complete expression is deterministic:

```
DATEADD(d,
  DATEDIFF(d, CONVERT(DATE, '00010101',
    112), upper)
- (DATEDIFF(d, CONVERT(DATE, '00010101',
  112), upper)+1) %
  POWER(2, FLOOR(LOG(
    DATEDIFF(d, CONVERT(DATE, '00010101',
       112), lower) ^
    (DATEDIFF(d, CONVERT(DATE, '00010101',
      112), upper)+1)) / LOG(2))),
  CONVERT(DATE, '00010101', 112))
```

Looks complex, doesn't it? Wait until you see the corresponding expression for the DATETIME2 data type, later in the article! Meanwhile, notice how this expression computes the fork node of the interval [lower, upper], as a DATE.

Finally, here is the definition of the IntervalsDate table, along with the indexes enabling efficient

querying:

```
   CREATE TABLE dbo.IntervalsDate
   (
     id    INT   NOT NULL PRIMARY KEY,
     node  AS  DATEADD(d,
     DATEDIFF(d, CONVERT(DATE, '00010101',
       112), upper)
     - (DATEDIFF(d, CONVERT(DATE, '00010101',
     112), upper)+1) %
     POWER(2, FLOOR(LOG(
       DATEDIFF(d, CONVERT(DATE, '00010101',
           112), lower) ^
       (DATEDIFF(d, CONVERT(DATE, '00010101',
         112), upper)+1)) / LOG(2))),
     CONVERT(DATE, '00010101', 112))
       PERSISTED NOT NULL,
     lower DATE  NOT NULL,
     upper DATE  NOT NULL
   );
   CREATE INDEX IX_IntervalsDate_lower ON dbo.
IntervalsDate(node, lower, upper);
   CREATE INDEX IX_IntervalsDate_upper ON dbo.
IntervalsDate(node, upper, lower);
```

A Static RI-Tree containing DATETIME2 values

If you need more precise time intervals, including a date and a time, you should use a Static RI-Tree containing DATETIME or DATETIME2(n) values, where n is the precision, a number of digits between 0 and 7 for the fractional seconds. By default, DATETIME2 is equivalent to DATETIME2(7), and the rightmost digit represents the multiples of 100 nanoseconds (1 ns = 10-9 s). I'll present an implementation for DATETIME2. Once you get the ideas behind the solution, you can adapt the implementation if you need less precision, like DATETIME2(0) or DATETIME2(3).

Mapping DATETIME2 to BIGINT

The DATETIME2 data type enables the user to represent values ranging from January 1st, 0001 at midnight through December 31st, 9999 at 23:59:59.9999999. To map these values to integers, we'll decompose them into hours, minutes, seconds and fractional seconds, and express each component in multiples of 100 nanoseconds before summing them up:

- Fractional seconds are expressed in multiples of 100 nanoseconds.
- Seconds are multiplied by 107.
- Minutes are multiplied by 60 * 107.
- Hours are multiplied by 3600 * 107.
- Add 1 to map the lower bound value to 1 instead of 0.

Thus, the resulting integer represents the initial DATETIME2 value expressed in multiples of 100 nanoseconds, plus one. A variable @d of type DATETIME2 gets mapped to:

```
DATEDIFF(hh, '00010101', @d) * 36000000000
+ DATEPART(mi, @d) * 600000000
+ DATEPART(s, @d) * 10000000
+ DATEPART(ns, @d) / 100 + 1
```

Note that there are several problems with this expression. If you try to use it in a query or expression, you'll get an error for some of the values. First, the constant 36000000000 is greater than the maximum INT value (2147483647), so SQL Server automatically treats it as a NUMERIC value. Then, the constant 600000000 is smaller than the maximum INT value, so SQL Server treats it as an INT, but when it gets multiplied by DATEPART(mi, @d), which is considered as an INT, the result may well exceed the maximum INT value, so an arithmetic overflow error might be raised. Finally, the expression is non-deterministic.

Obviously, the integer to which the original DATETIME2 value is mapped cannot be an INT, but will a BIGINT suffice? The maximum DATETIME2 value is December 31st, 9999 at 23:59:59.9999999, which maps to 3,155,378,976,000,000,000. Since the maximum BIGINT value is 9,223,372,036,854,775,807, the answer is yes, a BIGINT is sufficient to hold the mapped integer value of any DATETIME2 value.

Here is the mapping expression, rewritten to

avoid the problems described above:

```
   ((CAST(DATEDIFF(hh, CONVERT(DATETIME2,
'00010101', 112), @d) AS BIGINT) * 60
   + DATEPART(mi, @d)) * 60
   + DATEPART(s, @d)) * 10000000
   + DATEPART(ns, @d) / 100 + 1
```

When computing this expression, SQL Server does the following (please pay attention to the parentheses):

- The result of the DATEDIFF function is an INT whose maximum value is 87649415. It gets cast to a BIGINT.
- The constant 60 is an INT which gets cast to a BIGINT because the left operand of the * operator is a BIGINT.
- DATEPART(mi, @d) returns an INT which gets cast to a BIGINT since the left operand of the first + operator is a BIGINT.
- The second constant 60 is an INT which gets cast to a BIGINT because the left operand of the second * operator is a BIGINT.
- DATEPART(s, @d) returns an INT which gets cast to a BIGINT since the left operand of the second + operator is a BIGINT.
- The constant 10000000 is an INT which gets cast to a BIGINT because the left operand of the third * operator is a BIGINT.
- DATEPART(ns, @d) returns an INT, the constant 100 is an INT, and the result of the / operator is an INT, which gets cast to a BIGINT since the left operand of the third + operator is a BIGINT.
- The constant 1 as an INT which gets cast to a BIGINT because the left operand of the fourth + operator is a BIGINT.

As you can see, the expression correctly handles conversions from INT to BIGINT, avoids overflows and remains reasonably compact. Also, it's deterministic because the varchar literal '00010101' is explicitly converted to a DATETIME2 with a style of 112. (Many thanks to Itzik Ben-Gan for helping me get this expression right!)

By the way, while searching for the best expression for the mapping, I was a bit frustrated that the DATEDIFF and DATEADD functions only work with INT values: DATEDIFF returns an INT and DATEADD's second argument is an INT. Having similar functions working with BIGINT values would have greatly simplified the mapping expressions, because the decomposition into hours, minutes, seconds and fractional seconds would have been unnecessary in the first place! If you agree, I invite you to vote for the Connect item that Itzik has posted on the subject.

Mapping BIGINT back to DATETIME2

To map a BIGINT to a DATETIME2, let's decompose the integer into days after January 1st, 0001, seconds after midnight and fractional seconds, and then reassemble the components with the DATEADD function. Assuming the variable @i is a BIGINT value, the expression is

```
DATEADD(ns,
  ((@i-1) % 10000000) * 100,
  DATEADD(s,
    ((@i-1) / 10000000) % 86400,
    DATEADD(d,
     (@i-1) / cast(864000000000 AS BIGINT),
     CONVERT(DATETIME2, '00010101', 112)
    )))
```

In this expression, (@i-1) % 10000000 computes the fractional seconds component, in multiples of 100 nanoseconds, ((@i-1) / 10000000) % 86400 extracts the seconds since midnight component and (@i-1) / cast(864000000000 AS BIGINT) calculates the days component.

The last step is to replace lower and upper in the original inline formula computing the fork node by our DATETIME2-to-BIGINT mapping expression, and then to map the result back to a DATETIME2. On the next page, you'll find the final expression to compute the fork node as a DATETIME2 for an interval [lower, upper] where lower and upper are DATETIME2 values. Hold your breath!

```
DATEADD(ns,
  ((((CAST(DATEDIFF(hh, CONVERT(DATETIME2, '00010101', 112), upper) AS BIGINT) * 60
    + DATEPART(mi, upper)) * 60
    + DATEPART(s,  upper)) * 10000000
    + DATEPART(ns, upper)  / 100
    -
    (((CAST(DATEDIFF(hh, CONVERT(DATETIME2, '00010101', 112), upper) AS BIGINT) * 60
      + DATEPART(mi, upper)) * 60
      + DATEPART(s,  upper)) * 10000000
      + DATEPART(ns, upper)  / 100 + 1
      ) % POWER(CAST(2 AS BIGINT),
          FLOOR(LOG(
           (((CAST(DATEDIFF(hh, CONVERT(DATETIME2, '00010101', 112), lower) AS BIGINT) * 60
             + DATEPART(mi, lower)) * 60
             + DATEPART(s,  lower)) * 10000000
             + DATEPART(ns, lower)  / 100
             ) ^
             (((CAST(DATEDIFF(hh, CONVERT(DATETIME2, '00010101', 112), upper) AS BIGINT) * 60
               + DATEPART(mi, upper)) * 60
               + DATEPART(s,  upper)) * 10000000
               + DATEPART(ns, upper)  / 100 + 1)) / LOG(2)))) % 10000000) * 100,
        DATEADD(s,
          ((((CAST(DATEDIFF(hh, CONVERT(DATETIME2, '00010101', 112), upper) AS BIGINT) * 60
            + DATEPART(mi, upper)) * 60
            + DATEPART(s,  upper)) * 10000000
            + DATEPART(ns, upper)  / 100
            -
            (((CAST(DATEDIFF(hh, CONVERT(DATETIME2, '00010101', 112), upper) AS BIGINT) * 60
              + DATEPART(mi, upper)) * 60
              + DATEPART(s,  upper)) * 10000000
              + DATEPART(ns, upper)  / 100 + 1
              ) % POWER(CAST(2 AS BIGINT),
                  FLOOR(LOG(
                   (((CAST(DATEDIFF(hh, CONVERT(DATETIME2, '00010101', 112), lower) AS BIGINT)
                     * 60
                     + DATEPART(mi, lower)) * 60
                     + DATEPART(s,  lower)) * 10000000
                     + DATEPART(ns, lower)  / 100
                     ) ^
                     (((CAST(DATEDIFF(hh, CONVERT(DATETIME2, '00010101', 112), upper) AS
                       BIGINT) * 60
                       + DATEPART(mi, upper)) * 60
                       + DATEPART(s,  upper)) * 10000000
                       + DATEPART(ns, upper)  / 100 + 1)) / LOG(2)))) / 10000000) % 86400,
                 DATEADD(d,
                   (((CAST(DATEDIFF(hh, CONVERT(DATETIME2, '00010101', 112), upper) AS BIGINT)
                     * 60
                     + DATEPART(mi, upper)) * 60
                     + DATEPART(s,  upper)) * 10000000
                     + DATEPART(ns, upper)  / 100
```

```
-
(((CAST(DATEDIFF(hh, CONVERT(DATETIME2, '00010101', 112), upper) AS
    BIGINT) * 60
  + DATEPART(mi, upper)) * 60
  + DATEPART(s,  upper)) * 10000000
  + DATEPART(ns, upper)  / 100 + 1
  ) % POWER(CAST(2 AS BIGINT),
      FLOOR(LOG(
       (((CAST(DATEDIFF(hh, CONVERT(DATETIME2, '00010101', 112), lower) AS
          BIGINT) * 60
        + DATEPART(mi, lower)) * 60
        + DATEPART(s,  lower)) * 10000000
        + DATEPART(ns, lower)  / 100
        ) ^
        (((CAST(DATEDIFF(hh, CONVERT(DATETIME2, '00010101', 112), upper)
          AS BIGINT) * 60
          + DATEPART(mi, upper)) * 60
          + DATEPART(s,  upper)) * 10000000
          + DATEPART(ns, upper)  / 100 + 1)) / LOG(2)))
    ) / CAST(864000000000 AS BIGINT),
    CONVERT(DATETIME2, '00010101', 112))))
```

This expression could have been greatly simplified by using scalar user-defined functions or subqueries. But unfortunately, scalar used-defined functions have a negative impact on performance and subqueries are forbidden in a computed column's inline formula. And until now, I haven't found a simpler yet well performing expression! SQL Server 2008 and 2012 aren't very helpful with such expressions for computed columns. I wish there were a way to use aliases as in CTEs.

The IntervalsDateTime2 table

Here is the IntervalsDateTime2 table along with the indexes for efficient querying:

```
CREATE TABLE dbo.IntervalsDateTime2
(
  id    INT NOT NULL PRIMARY KEY,
  node  AS … inline forknode formula goes
            here…
        PERSISTED NOT NULL,
  lower DATETIME2 NOT NULL,
  upper DATETIME2 NOT NULL
);
CREATE INDEX IX_IntervalsDateTime2_lower ON
dbo.IntervalsDateTime2(node, lower, upper);
CREATE INDEX IX_IntervalsDateTime2_upper ON
dbo.IntervalsDateTime2(node, upper, lower);
```

Computing the fork node with CLR functions

If using CLR code in the database is an option for you, you can easily implement the fork node computation as a CLR user-defined function. The performance could be better than with SQL code (see next section). Here is the C# code for the DATE and DATETIME2 versions of the fork node function:

```
using System;
using Microsoft.SqlServer.Server;

namespace DateRITree
{
  public static class RITree
  {
    [SqlFunction(IsDeterministic = true,
     DataAccess = DataAccessKind.None,
     IsPrecise = true)]
    public static DateTime ForkDateCLR(
      DateTime lower, DateTime upper)
    {
      // Convert the parameters to 32-bit
      // integers.
      Int32 lowerInt = (lower.Date -
        DateTime.MinValue).Days + 1;
      Int32 upperInt = (upper.Date -
        DateTime.MinValue).Days + 1;

      // Compute the 32-bit fork node
Int32 node = ((lowerInt - 1) ^
        upperInt) >> 1;
      node |= node >> 1;
      node |= node >> 2;
      node |= node >> 4;
      node |= node >> 8;
      node |= node >> 16;
      node = upperInt & ~node;

      // Convert the result to DateTime
      return new DateTime((node - 1) *
        864000000000L);
    }

    [SqlFunction(IsDeterministic = true,
     DataAccess = DataAccessKind.None,
     IsPrecise = true)]
    public static DateTime
    ForkDateTime2CLR(DateTime lower,
      DateTime upper)
    {
      // Convert the parameters to 64-bit
      // integers.
      Int64 lowerInt = lower.Ticks + 1L;
      Int64 upperInt = upper.Ticks + 1L;

   // Compute the 64-bit fork node
      Int64 node = ((lowerInt - 1) ^
        upperInt) >> 1;
      node |= node >> 1;
      node |= node >> 2;
      node |= node >> 4;
      node |= node >> 8;
      node |= node >> 16;
      node |= node >> 32;
      node = upperInt & ~node;

      // Convert the result to DateTime
      return new DateTime(node - 1L);
    }
  }
}
```

Personally, I find this C# code so much simpler and clearer than the equivalent SQL!

Note that the Ticks property of the System.DateTime structure returns the same value as our DATETIME2-to-BIGINT mapping, minus one.

One interesting thing to try is have the intervals table contain 2 fork node columns: one computed with the large SQL expression and one computed via a call to the C# function. It is then easy to check that both columns always have the same value.

Measuring bulk insertion performance

In this section, let's examine the fork node computation cost (the computed column in the intervals table) at insertion time. For this, we'll generate random intervals into a staging table, and then compare the time needed to perform a bulk insertion of 10,000,000 rows into the target interval table in the 3 following situations:

1. The node column is initialized to a default value
2. The node column is computed as the interval's fork node with an inline formula in SQL
3. The node column is computed as the interval's fork node with a call to a CLR function written in C#

The GetNums function

Let's first see a handy utility function. The GetNums function, written by Itzik Ben-Gan, returns the set of integers included within a range specified by its boundaries, with an excellent performance and without needing to read from a database table. The code for GetNums is:

```
   CREATE FUNCTION dbo.GetNums(@low AS BIGINT, @
high AS BIGINT) RETURNS TABLE
   AS
   RETURN
     WITH
       L0   AS (SELECT c FROM (SELECT 1 UNION ALL
SELECT 1) AS D(c)),
       L1   AS (SELECT 1 AS c FROM L0 AS A CROSS
JOIN L0 AS B),
       L2   AS (SELECT 1 AS c FROM L1 AS A CROSS
JOIN L1 AS B),
       L3   AS (SELECT 1 AS c FROM L2 AS A CROSS
JOIN L2 AS B),
       L4   AS (SELECT 1 AS c FROM L3 AS A CROSS
JOIN L3 AS B),
       L5   AS (SELECT 1 AS c FROM L4 AS A CROSS
JOIN L4 AS B),
      Nums AS (SELECT ROW_NUMBER() OVER(ORDER BY
(SELECT NULL)) AS rownum
               FROM L5)
     SELECT TOP(@high - @low + 1) @low +
       rownum - 1 AS n
     FROM Nums
     ORDER BY rownum;
```

For instance, here's how to select all integers between (and including) 1 and 10000:

```
   SELECT n FROM dbo.GetNums(1, 10000);
```

I like to call GetNums the "harp" function. If you wonder why, take a look at the execution plan of the preceding query.

Inserting into the IntervalsDate table

The code to create and fill the staging table with DATE intervals is the following:

```
CREATE TABLE dbo.StagingDate
(
  id    INT  NOT NULL PRIMARY KEY,
  lower DATE NOT NULL,
  upper DATE NOT NULL
);
DECLARE @num_intervals INT = 10000000,
        @max_interval_length_days INT = 30;
DECLARE @max_lower INT =
  (SELECT DATEDIFF(d, CONVERT(DATE,
   '00010101', 112), DATEADD(d,
   -@max_interval_length_days, '99991231'
  )));

WITH T AS
(
  SELECT n, DATEADD(D,
   1 + ABS(CHECKSUM(NEWID())) % @max_lower,
   CONVERT(DATE, '00010101', 112)) AS lower
  FROM dbo.GetNums(1, @num_intervals)
)
INSERT dbo.StagingDate WITH(TABLOCK)
       (id, lower, upper)
SELECT n,
       lower,
       DATEADD(d, ABS(CHECKSUM(NEWID())) %
        (@max_interval_length_days + 1),
        lower)
FROM   T;
```

In the code above, @num_intervals is the number of intervals to insert, @max_lower is the maximum lower bound for an interval and @max_interval_length_days is the maximum length of an interval in days.

Now, let's define the 3 test tables:

```
CREATE TABLE dbo.IntervalsDateRaw
(
  id    INT   NOT NULL PRIMARY KEY,
  node  DATE  NOT NULL DEFAULT
    CONVERT(DATE, '00010101', 112),
  lower DATE  NOT NULL,
  upper DATE  NOT NULL
);
CREATE INDEX IX_IntervalsDateRaw_lower ON dbo.
IntervalsDateRaw(node, lower, upper);
CREATE INDEX IX_IntervalsDateRaw_upper ON dbo.
IntervalsDateRaw(node, upper, lower);
GO
CREATE TABLE dbo.IntervalsDate
(
  id    INT   NOT NULL PRIMARY KEY,
  node  AS   …inline forknode formula goes
             here…
    PERSISTED NOT NULL,
  lower DATE  NOT NULL,
  upper DATE  NOT NULL
);
CREATE INDEX IX_IntervalsDate_lower ON dbo.
IntervalsDate(node, lower, upper);
CREATE INDEX IX_IntervalsDate_upper ON dbo.
IntervalsDate(node, upper, lower);
GO
CREATE TABLE dbo.IntervalsDateCLR
(
  id    INT   NOT NULL PRIMARY KEY,
  node  AS dbo.ForkDateCLR(lower, upper)
PERSISTED NOT NULL,
  lower DATE  NOT NULL,
  upper DATE  NOT NULL
);
CREATE INDEX IX_IntervalsDateCLR_lower ON dbo.
IntervalsDateCLR(node, lower, upper);
CREATE INDEX IX_IntervalsDateCLR_upper ON dbo.
IntervalsDateCLR(node, upper, lower);
```

Finally, below is the code to measure the bulk insertion time in seconds. Execute it once for each table, replacing IntervalsDateX by IntervalsDateRaw, IntervalsDate then IntervalsDateCLR.

```
DECLARE @t0 DATETIME = CURRENT_TIMESTAMP;
INSERT dbo.IntervalsDateX(id, lower, upper)
SELECT id, lower, upper
FROM   dbo.StagingDate;
SELECT DATEDIFF(s, @t0, CURRENT_TIMESTAMP);
```

On my desktop, the results are as follows:

Computed node column	Bulk insert time (s)
Default value	216
Inline formula	231
CLR function	227

It appears that, for the DATE data type, a CLR function does not offer any significant performance advantage over an inline formula.

Inserting into de IntervalsDateTime2 table

The code to create and fill the staging table with DATETIME2 intervals is the following:

```
CREATE TABLE dbo.StagingDateTime2
(
  id    INT  NOT NULL PRIMARY KEY,
  lower DATETIME2 NOT NULL,
  upper DATETIME2 NOT NULL
);

DECLARE @num_intervals INT = 10000000,
        @max_interval_length_days INT = 2;
DECLARE @max_lower INT =
  (SELECT node FROM dbo.MapDateToInt(
   DATEADD(d, -@max_interval_length_days,
    '99991231')));
WITH T1 AS
(
  SELECT n, 1 + ABS(CHECKSUM(NEWID())) %
    @max_lower AS lower
  FROM dbo.GetNums(1, @num_intervals)
),
T2 AS
(
  SELECT n, lower, lower +
    ABS(CHECKSUM(NEWID())) %
```

```
@max_interval_length_days AS upper
  FROM T1
),
T3 AS
(
  SELECT n,
DATEADD(ns, (ABS(CHECKSUM(NEWID())) %
       10000000) * 100,
     DATEADD(s, ABS(CHECKSUM(NEWID())) %
          86400,
      DATEADD(d, lower,
       CONVERT(DATETIME2, '00010101', 112))
    )) AS lower,
    DATEADD(ns, (ABS(CHECKSUM(NEWID())) %
        10000000) * 100,
     DATEADD(s, ABS(CHECKSUM(NEWID())) %
          86400,
      DATEADD(d, upper,
       CONVERT(DATETIME2, '00010101', 112))
    )) AS upper
    FROM T2
)
INSERT dbo.StagingDateTime2 WITH(TABLOCK)
       (id, lower, upper)
SELECT  n, lower,
        CASE WHEN upper < lower
          -- May happen on the same day
          THEN DATEADD(S,
            ABS(CHECKSUM(NEWID()))
              % 36000, lower)
          ELSE upper
        END AS upper
FROM   T3;
```

In the code above, @num_intervals is the number of intervals to insert, @max_lower is the maximum lower bound (the day part) for an interval and @max_interval_length_days is the maximum length of an interval in days.

Now, let's define the 3 test tables:

```
    CREATE TABLE dbo.IntervalsDateTime2Raw
    (
      id    INT       NOT NULL PRIMARY KEY,
      node  DATETIME2 NOT NULL
        DEFAULT CONVERT(DATETIME2, '00010101',
          112),
    lower DATETIME2 NOT NULL,
      upper DATETIME2 NOT NULL
    );
    CREATE  INDEX IX_IntervalsDateTime2Raw_lower
ON dbo.IntervalsDateTime2Raw(node, lower, upper);
    CREATE  INDEX IX_IntervalsDateTime2Raw_upper
ON dbo.IntervalsDateTime2Raw(node, upper, lower);
    GO
    CREATE TABLE dbo.IntervalsDateTime2
    (
      id    INT       NOT NULL PRIMARY KEY,
      node  AS  …inline forknode formula goes
                here…
            PERSISTED NOT NULL,
      lower DATETIME2 NOT NULL,
      upper DATETIME2 NOT NULL
    );
    CREATE  INDEX IX_IntervalsDateTime2_lower  ON
dbo.IntervalsDateTime2(node, lower, upper);
    CREATE  INDEX IX_IntervalsDateTime2_upper  ON
dbo.IntervalsDateTime2(node, upper, lower);
    GO
    CREATE TABLE dbo.IntervalsDateTime2CLR
    (
      id    INT   NOT NULL PRIMARY KEY,
      node  AS dbo.ForkDateTime2CLR(lower, upper)
            PERSISTED NOT NULL,
    lower DATETIME2 NOT NULL,
      upper DATETIME2 NOT NULL
    );
    CREATE INDEX IX_IntervalsDateTime2CLR_lower ON
dbo.IntervalsDateTime2CLR(node, lower,
      upper);
    CREATE INDEX IX_IntervalsDateTime2CLR_upper ON
dbo.IntervalsDateTime2CLR(node, upper,
      lower);
```

Finally, below is the code to measure the bulk insertion time in seconds. Execute it once for each table, replacing IntervalsDateTime2X by IntervalsDateTime2Raw, IntervalsDateTime2 then IntervalsDateTime2CLR.

```
DECLARE @t0 DATETIME = CURRENT_TIMESTAMP;
INSERT dbo.IntervalsDateTime2X(id, lower,
  upper)
SELECT id, lower, upper
FROM   dbo.StagingDateTime2;
SELECT DATEDIFF(s, @t0, CURRENT_TIMESTAMP);
```

On my desktop, the results are as follows:

Computed node column	Bulk insert time (s)
Default value	216
Inline formula	231
CLR function	227

With the DATETIME2 data type, the performance is much better with a CLR function.

Part 2: Query Static BI-Trees holding time intervals

Let's examine what's involved to make the original intersection queries and Allen queries work with Static RI-Trees holding DATE or DATETIME2-based intervals.

Time interval query support objects

The Static RI-Tree interval queries use a BitMasks table and a couple of user-defined functions. These need to be adapted to the DATE and DATETIME2 data types. In addition, we need to wrap time-to-integer mappings into dedicated functions.

The BitMasksDate and BitMasksDateTime2 tables

The BitMasksDate and BitMasksDateTime2 tables are the adaptations of the BitMasks table to respectively the DATE and DATETIME2 data types.

Since the maximum DATE value, '99991231', maps to the integer value 3652059, the BitMasksDate table should be initially filled in such a way that b3 is less than or equal to 3652059, to ensure that ancestors returned never exceed this limit. In practice, we need b3 <= 221, because 222 equals 4194304. Thus, the code to create and initialize BitMasksDate is:

```
CREATE TABLE dbo.BitMasksDate
(
  b1     INT NOT NULL,
  b3     INT NOT NULL
);
INSERT dbo.BitMasksDate(b1, b3)
SELECT -POWER(2, n),
        POWER(2, n)
FROM   dbo.GetNums(1, 21);
```

Note that the b2 column of the original BitMasks table is no longer used because the Ancestors function (explained shortly) doesn't require it.

The Static RI-Tree for DATETIME2 internally uses BIGINT values, so we must use a 64-bit version of the BitMasks table. Since the maximum DATETIME2 value, '99991231 23:59:59.9999999', maps to the integer value 3,155,378,976,000,000,000, the BitMasksDateTime2 table should be initially filled in such a way that b3 is less than or equal to this integer, to ensure that ancestors returned never exceed the limit. In practice, we need b3 <= 261, because 262 equals 4,611,686,018,427,387,904. The code below creates the BitMasksDateTime2 table and populates it.

```
CREATE TABLE dbo.BitMasksDateTime2
(
  b1     BIGINT NOT NULL,
  b3     BIGINT NOT NULL
);
INSERT dbo.BitMasksDateTime2(b1, b3)
SELECT -POWER(CAST(2 AS BIGINT), n),
        POWER(CAST(2 AS BIGINT), n)
FROM   dbo.GetNums(1, 61);
```

The time-to-integer and integer-to-time mapping functions

We need to be able to easily map time values to

integers and vice versa. To do this, let's wrap the mapping expressions we've discussed above into functions. However, let's use inline table-valued functions to ensure the cost in execution time is always minimal. The DATE-to-INT mapping function is presented below:

```
CREATE FUNCTION dbo.MapDateToInt(@d
  DATE)
RETURNS TABLE
AS
RETURN
  SELECT DATEDIFF(d,
    CONVERT(DATE, '00010101', 112), @d) + 1
    AS node;
```

The DATETIME2-to-BIGINT mapping function is:

```
CREATE FUNCTION dbo.MapDateTime2ToBigInt(
  @d DATETIME2)
RETURNS TABLE
AS
RETURN
  SELECT
   ((CAST(DATEDIFF(hh,
    CONVERT(DATETIME2, '00010101', 112),
    @d) AS BIGINT) * 60
    + CAST(DATEPART(mi, @d) AS BIGINT)) * 60
    + CAST(DATEPART(s,  @d) AS BIGINT))
      * 10000000
    + CAST(DATEPART(ns, @d) AS BIGINT)
      / 100 + 1 AS node;
```

And here are the integer-to-time mapping functions. First, the INT-to-DATE mapping function:

```
CREATE FUNCTION dbo.MapIntToDate(@i INT)
RETURNS TABLE
AS
RETURN
  SELECT DATEADD(d, @i - 1,
   CONVERT(DATE, '00010101', 112)) as node;
```

And second, the BIGINT-to-DATETIME2 mapping function:

```
CREATE FUNCTION dbo.MapBigIntToDateTime2(@i
  BIGINT)
RETURNS TABLE
AS
RETURN
  SELECT DATEADD(ns,
   ((@i - 1) % 10000000) * 100,
   DATEADD(s,
     ((@i - 1) / 10000000) % 86400,
     DATEADD(d,
       (@i - 1) / cast(864000000000
          AS BIGINT),
       CONVERT(DATETIME2, '00010101', 112)
     ))) AS node;
```

The Fork function

Next, we need time versions of the Fork function, to compute the fork node of a time interval. As a reminder, the original implementation for INT values is:

```
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
```

To derive the DATE and DATETIME2 versions, we need to map the arguments to integers, feed these integers into the fork expression and map the result back to a DATE or DATETIME2. Here is the DATE version of the Fork function:

```
CREATE FUNCTION dbo.ForkDate(@lower DATE,
  @upper DATE)
  RETURNS DATE
AS
BEGIN
```

```
  DECLARE @lowerInt INT, @upperInt INT,
          @forkInt  INT, @fork DATE;

  SELECT @lowerInt = node FROM
    dbo.MapDateToInt(@lower);
  SELECT @upperInt = node FROM
    dbo.MapDateToInt(@upper);
  SET @forkInt = @upperInt - @upperInt %
    POWER(2, FLOOR(LOG((@lowerInt - 1) ^
      @upperInt) / LOG(2)));
  SELECT @fork = node FROM
    dbo.MapIntToDate(@forkInt);

    RETURN @fork;
END
```

And here is the DATETIME2 version of the Fork function:

```
CREATE FUNCTION dbo.ForkDateTime2(@lower
  DATETIME2, @upper DATETIME2)
  RETURNS DATETIME2
AS
BEGIN
  DECLARE @lowerInt BIGINT,
          @upperInt BIGINT,
          @forkInt  BIGINT,
          @fork     DATETIME2;

  SELECT @lowerInt = node FROM
    dbo.MapDateTime2ToBigInt(@lower);
  SELECT @upperInt = node FROM
    dbo.MapDateTime2ToBigInt(@upper);
  SET @forkInt = @upperInt - @upperInt %
    POWER(CAST(2 AS BIGINT),
      FLOOR(LOG((@lowerInt - 1) ^
        @upperInt) / LOG(2)));
SELECT @fork = node FROM
    dbo.MapBigIntToDateTime2(@forkInt);

  RETURN @fork;
END
```

The Ancestors function

While I was researching solutions to efficiently implement time-based Static RI-Trees, Itzik Ben-Gan came up with a beautiful alternative to my LeftNodes and RightNodes functions: the Ancestors function, which computes the set of ancestors of a node in the RI-Tree. I love this function because it's simpler, less abstract and much clearer than LeftNodes and RightNodes. In addition, it performs equally well. The integer form of Ancestors is as follows:

```
    CREATE FUNCTION dbo.Ancestors(@node AS INT)
RETURNS TABLE
    AS
    RETURN
      SELECT @node & b1 | b3 as node
      FROM dbo.BitMasks
      WHERE b3 > @node & -@node;
```

The magical expression @node & -@node computes the rightmost set bit in the binary form of @node's value. To achieve the same results as LeftNodes, you simply need to query the Ancestors function and restrict the output to values strictly less than @node. Conversely, just restrict the output to values strictly greater than @nodes to obtain the same results as RightNodes.

The DATE and DATETIME2 versions of the function are obtained by mapping the argument to an integer, then feeding this integer into the ancestors expression and mapping the result back to a DATE or DATETIME2. The DATE version of the Ancestors function is:

```
    CREATE FUNCTION dbo.AncestorsDate(@d DATE)
    RETURNS TABLE
    AS
    RETURN
      SELECT  D.node
      FROM    dbo.BitMasksDate AS BM WITH(NOLOCK)
      CROSS JOIN  dbo.MapDateToInt(@d) AS I
      CROSS APPLY dbo.MapIntToDate(
                I.node & BM.b1 | BM.b3) AS D
      WHERE   BM.b3 > I.node & -I.node;
```

The DATETIME2 version of the Ancestors function is:

```
    CREATE FUNCTION dbo.AncestorsDateTime2(@d
DATETIME2)
    RETURNS TABLE
    AS
    RETURN
      SELECT  D.node
      FROM    dbo.BitMasksDateTime2 AS BM
              WITH(NOLOCK)
      CROSS JOIN dbo.MapDateTime2ToBigInt(@d)
                 AS I
      CROSS APPLY dbo.MapBigIntToDateTime2(
          I.node & BM.b1 | BM.b3) AS D
      WHERE   BM.b3 > I.node & -I.node;
```

Putting it all together: the interval query catalog

All interval queries can be written simply with the Fork and Ancestors functions. No need to keep the TopLeft, TopRight, InnerLeft, InnerRight, LeftNodes, RightNodes, BottomLeft and BottomRight functions!

The table below lists the complete interval query catalog for DATE-based Static RI-Trees.
Notes:

- The queries are written for DATE intervals. They can be transposed to DATETIME2 intervals by replacing ForkDate by ForkDateTime2, IntervalsDate by IntervalsDateTime2, AncestorsDate by AnctestorsDateTime2, DATE by DATETIME2 and DATE constants by DATETIME2 constants.
- To get the best query plans by sniffing the current values of local variables, you can append the OPTION (RECOMPILE) query hint.
- The @min and @max variables are used to implement the range optimization (see my previous article "Advanced interval queries with the Static Relational Interval Tree").
- The Intersects relationship is not part of the 13 interval relationships defined by Allen. Its query is equivalent to the union of all Allen queries, except those corresponding to Before and After. Its semantics are: all intervals intersecting the interval [@lower, @upper].

Relationship	Query
Intersects	DECLARE @lower DATE = '20130410', @upper DATE = '20130425', @min DATE, @max DATE; SELECT @min = MIN(node), @max = MAX(node) FROM dbo.IntervalsDate; SELECT id, lower, upper FROM dbo.IntervalsDate i JOIN dbo.AncestorsDate(@lower) ln ON i.node = ln.node AND i.upper >= @lower WHERE ln.node < @lower AND ln.node >= @min UNION ALL SELECT id, lower, upper FROM dbo.IntervalsDate i JOIN dbo.AncestorsDate(@upper) rn ON i.node = rn.node AND i.lower <= @upper WHERE rn.node > @upper AND rn.node <= @max UNION ALL SELECT id, lower, upper FROM dbo.IntervalsDate WHERE node BETWEEN @lower AND @upper;
Before	DECLARE @lower DATE = '00010410', @min DATE; SELECT @min = MIN(node) FROM dbo.IntervalsDate; SELECT id, lower, upper FROM dbo.IntervalsDate WHERE node < @lower AND upper < @lower AND node >= @min;
Meets	DECLARE @lower DATE = '20130410', @min DATE; SELECT @min = MIN(node) FROM dbo.IntervalsDate; SELECT id, lower, upper FROM dbo.IntervalsDate i JOIN (SELECT node FROM dbo.AncestorsDate(@lower) WHERE node < @lower AND node >= @min UNION ALL SELECT @lower) q ON i.node = q.node AND i.upper = @lower;

Relationship	Query

Overlaps	`DECLARE @lower DATE = '20130410',`
	`  @upper DATE = '20130421',`
	`  @min  DATE = (SELECT MIN(node) FROM`
	`  dbo.IntervalsDate);`
	`DECLARE @fork  DATE = dbo.ForkDate(@lower, @upper);`
	
	`SELECT  id, lower, upper`
	`FROM  dbo.IntervalsDate i`
	`JOIN  dbo.AncestorsDate(@lower) q`
	`  ON  i.node = q.node`
	`WHERE  i.upper > @lower`
	`  AND i.upper < @upper`
	`  AND q.node  < @lower`
	`  AND q.node >= @min`
	`UNION ALL`
	`SELECT  id, lower, upper`
	`FROM  dbo.IntervalsDate i`
	`JOIN  (`
	`  SELECT  node`
	`  FROM  dbo.AncestorsDate(@lower)`
	`  WHERE  node > @lower`
	`  AND node < @fork`
	`  UNION ALL`
	`  (`
	`  SELECT  @lower`
	`  UNION`
	`  SELECT  @fork`
	`  )`
	`  ) q`
	`  ON  i.node  = q.node`
	`WHERE  i.lower < @lower`
	`  AND i.upper < @upper`

FinishedBy	`DECLARE @lower DATE = '20130410',`
	`  @upper DATE = '20130421',`
	`  @min  DATE = (SELECT MIN(node) FROM`
	`  dbo.IntervalsDate);`
	`DECLARE @fork  DATE = dbo.ForkDate(@lower, @upper);`
	
	`SELECT  id, lower, upper`
	`FROM  dbo.IntervalsDate i`
	`JOIN  (`
	`  SELECT  node`
	`  FROM  dbo.AncestorsDate(@fork)`
	`  WHERE  node >= @min`
	`  AND node < @fork`
	`  UNION ALL`
	`  SELECT  @fork`
	`  ) q`
	`  ON  i.node  = q.node`
	`WHERE  i.upper = @upper`
	`  AND i.lower < @lower;`

Starts

```
DECLARE @lower DATE = '20130410',
        @upper DATE = '20130421';
DECLARE @fork  DATE = dbo.ForkDate(@lower, @upper);

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

Contains

```
DECLARE @lower DATE = '20130410',
        @upper DATE = '20130421',
        @min   DATE = (SELECT MIN(node) FROM
                       dbo.IntervalsDate),
        @max   DATE = (SELECT MAX(node) FROM
                       dbo.IntervalsDate);
DECLARE @fork  DATE = dbo.ForkDate(@lower, @upper);

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

Equals	DECLARE @lower DATE = '20130410',
	  @upper DATE = '20130421';
	DECLARE @fork DATE = dbo.ForkDate(@lower, @upper);
	
	SELECT id, lower, upper
	FROM dbo.IntervalsDate i
	WHERE i.node = @fork
	  AND i.lower = @lower
	  AND i.upper = @upper;

During	DECLARE @lower DATE = '20130410',
	  @upper DATE = '20130421',
	  @min DATE = (SELECT MIN(node) FROM
	  dbo.IntervalsDate),
	  @max DATE = (SELECT MAX(node) FROM
	  dbo.IntervalsDate);
	DECLARE @fork DATE = dbo.ForkDate(@lower, @upper);
	
	SELECT id, lower, upper
	FROM dbo.IntervalsDate 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, lower, upper
	FROM dbo.IntervalsDate i
	WHERE i.node > @fork
	  AND i.node < @upper
	  AND i.node <= @max
	  AND i.upper < @upper;

StartedBy	DECLARE @lower DATE = '20130410',
	  @upper DATE = '20130421',
	  @max DATE = (SELECT MAX(node) FROM
	  dbo.IntervalsDate);
	DECLARE @fork DATE = dbo.ForkDate(@lower, @upper);
	
	SELECT id, lower, upper
	FROM dbo.IntervalsDate i
	JOIN (
	  SELECT node
	  FROM dbo.AncestorsDate(@fork)
	  WHERE node > @fork
	  AND node <= @max
	  UNION ALL
	  SELECT @fork
	  ) q
	  ON i.node = q.node
	WHERE i.lower = @lower
	  AND i.upper > @upper;

Finishes

```
DECLARE @lower DATE = '20130410',
        @upper DATE = '20130421';
DECLARE @fork  DATE = dbo.ForkDate(@lower, @upper);

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

OverlappedBy

```
DECLARE @lower DATE = '20130410',
        @upper DATE = '20130421',
        @max   DATE = (SELECT MAX(node) FROM dbo.IntervalsDate);
DECLARE @fork  DATE = dbo.ForkDate(@lower, @upper);

SELECT  id, lower, upper
FROM    dbo.IntervalsDate i
JOIN    (
            SELECT  node
            FROM    dbo.AncestorsDate(@upper)
            WHERE   node > @upper
                AND node <= @max
        ) q
    ON  i.node = q.node
WHERE   i.lower > @lower
    AND i.lower < @upper
UNION ALL
SELECT  id, lower, upper
FROM    dbo.IntervalsDate i
JOIN    (
            SELECT  node
            FROM    dbo.AncestorsDate(@upper)
            WHERE   node < @upper
                AND node > @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	DECLARE @lower DATE = '20130410', @upper DATE = '20130421', @max DATE = (SELECT MAX(node) FROM dbo.IntervalsDate); SELECT id, lower, upper FROM dbo.IntervalsDate i JOIN (SELECT node FROM dbo.AncestorsDate(@upper) WHERE node > @upper AND node <= @max UNION ALL SELECT @upper) q ON i.node = q.node WHERE i.lower = @upper;
Alter	DECLARE @lower DATE = '20130410', @upper DATE = '20130421', @max DATE = (SELECT MAX(node) FROM dbo.IntervalsDate); SELECT id, lower, upper FROM dbo.IntervalsDate WHERE node > @upper AND lower > @upper AND node <= @max;

Conclusion - Microsoft could make a difference

In this article, I exposed techniques to implement Static RI-Trees containing time intervals. The heart of the implementation consists of time-to-integer mappings, which enable seamless integration of time values within the interval queries.

However, the table definition is complex, especially for DATETIME2, and the insertion overhead is far from negligible. The interval queries themselves can be a bit complex, too. In March 2013, Itzik Ben-Gan posted a Connect item suggesting that Microsoft add Static RI-Tree features to the SQL Server database engine, to hide the complexity and reduce the performance penalty to a minimum, like only native C++ code can do. If you find the idea useful, you're welcome to vote for it. ■

About the Author

Laurent Martin. *(Laurent.martin741@yahoo.fr) s 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.*