Self-maintaining, Contiguous Effective Dates in Temporal Tables

10 March 2015
by Dwain Camps

'Temporal' tables contain facts that are valid for a period of time. When they are used for financial information they have to be very well constrained to prevent errors getting in and causing incorrect reporting. This makes them more difficult to maintain. Is it possible to have both the stringent constraints and simple CRUD operations? Well, yes. Dwain Camps patiently explains the whole process.

Self-maintaining, Contiguous Effective Dates in Temporal Tables

In a database that is dealing with price information where a new price immediately supersedes the previous one, you can find the correct price for any particular date when all you have is an 'effective date' on each row. This is because the 'effective end date' is implied by the effective date on the subsequent row, and the current price is always the one with the latest date on it. It is a simple system that requires few constraints to implement.

Although this allows us to reproduce account information from the data and run historic financial reports, it is often a better fit for the business requirements to have both an 'effective start' date and an 'effective end' date on each row. This can handle a range of the more generic cases of a 'transaction time table', and it also is much quicker to determine the current value since it will have a NULL end-date.

Joe Celko's article Contiguous Time Periods gives a good summary of this type of table, and he described an ingenious way of enforcing the basic rules of this type of table solely via constraints, in what he called "Kuznetsov's History Table." This is a special version of the temporal price table which has the extra '**previous end-date**' included in the row. This allows us to use a foreign key constraint to prevent a row being inserted that doesn't have an existing row with an end-date matching the entry in the 'previous-end-date' column. While it is an interesting approach, it does have some issues that he and Alex both cover in good detail.

In his article "Modifying Contiguous Time Periods in a History Table" Alex Kuznetsov shows not only how to set this up but also how to maintain such a table, because with such tight constraints it is no mean task.

In this article I'll show how it is possible to design and use such a table, but without the complexity of the '**previous-end-date**' column, yet still using only constraints to enforce data integrity. Then I'll show you how to use the MERGE statement to update the table with new prices, while automatically maintaining the effective end date column. Finally, I'll then show how it is possible to do updates to this table using only simple CRUD (Create, Read, Update, Delete) operations, hiding the complexity of the actual MERGE process in a trigger.

Enforcing Effective End-to-start Dates Using Constraints without the Additional Column

In order to demonstrate this approach, and show how it is used, we'll use a simple example. Here is the Data Definition Language (DDL) to create two tables. The first is a simple Products table (which we will populate with some test data) and the second is a **ProductPrices** table with CONSTRAINTs sufficient to enforce this end-to-start date relationship.

```
CREATE TABLE dbo.Products
(
    ProductID           INT IDENTITY
    ,ProductName        VARCHAR(50) NOT NULL
    ,DiscontinuedDT     DATETIME    NOT NULL DEFAULT ('2099-12-31')
    ,CONSTRAINT p_pk    PRIMARY KEY (ProductID)
);

INSERT INTO dbo.Products (ProductName)
VALUES ('Mouse Trap'), ('Better Mouse Trap'), ('Rat Trap');

SELECT *
FROM dbo.Products;
```

```
CREATE TABLE dbo.ProductPrices
(
    ProductID           INT         NOT NULL
    ,EffectiveStartDT   DATETIME    NOT NULL
    ,EffectiveEndDT     DATETIME    NULL
    -- Time dependent attributes
    ,ProductPrice       MONEY       NOT NULL
    -- In PK, EffectiveStartDT should occur only once for each product
    ,CONSTRAINT pp_pk   PRIMARY KEY (ProductID, EffectiveStartDT)
    --
    -- EffectiveEndDT should always be greater than EffectiveStartDT (unless NULL)
    ,CONSTRAINT pp_ck1  CHECK (EffectiveEndDT > EffectiveStartDT)
    --
    -- ProductPrice should always be greater than zero
    ,CONSTRAINT pp_ck2  CHECK (ProductPrice > 0)
    --
    -- FK1: Each EffectiveEndDT should be linked to an EffectiveStartDT
    ,CONSTRAINT pp_fk1  FOREIGN KEY (ProductID, EffectiveEndDT)
                        REFERENCES dbo.ProductPrices(ProductID, EffectiveStartDT)
    --
    -- FK2: Products/Prices must reflect existing Products
    ,CONSTRAINT pp_fk2  FOREIGN KEY (ProductID)
                        REFERENCES dbo.Products(ProductID)
    --
    -- EffectiveEndDT must be unique for each row of a product
    ,CONSTRAINT pp_u1   UNIQUE (ProductID, EffectiveEndDT)
);
```

Let's describe the CONSTRAINTs we've defined on **ProductPrices**.

- **pp_pk** – This is the PRIMARY KEY CONSTRAINT that ensures that for any product, the effective start date must be unique across all rows. That is, the same product cannot have the same effective start date appearing more than once.
- **pp_ck1** – This is a rather simple CHECK CONSTRAINT that ensures that any effective end date (on a row) will always be greater than the effective start date on that row.
- **pp_ck2** – This is another fairly simple CHECK CONSTRAINT that ensures that a product's price is always greater than zero. Unless you are giving products away, it is unlikely the price would ever be zero, and quite unlikely that it would ever be negative. A product's price can also not be NULL (as defined by the column itself).
- **pp_fk1** – This is a self-referencing FOREIGN KEY (FK) CONSTRAINT, that says any row that has an effective end date defined, i.e., it is not NULL, must have a matching row where that row's effective start date matches the effective end date. This enforces the "contiguous" nature of the end-to-start date relationships in the table.
- **pp_fk2** – This is a FOREIGN KEY CONSTRAINT that ensures that any ProductID entered into the ProductPrices table exists as a product in the Products table.
- **pp_u1** – This is a UNIQUE CONSTRAINT that ensures that each ProductID/EffectiveEndDT combination exists only once.

Let's add two additional facts about the way these CONSTRAINTs will work together:

- When an effective end date of NULL is inserted into the table, the self-referencing FK CONSTRAINT (**pp_fk1**) allows it, meaning that is the only case where there need not exist a corresponding effective start date.
- Because of the UNIQUE CONSTRAINT (**pp_u1**), only one such NULL effective end date can be in the table for a given product.

Populating our Product/Prices Table

Now, we will put some data into our **ProductPrices** table with an INSERT, where we have ensured that the contiguous nature of the end-to-start dates by setting them appropriately.

```
INSERT INTO dbo.ProductPrices
(
    ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice
)
```

```
VALUES (1, '2013-06-04', '2013-06-17', 15.25)
    ,(2, '2013-05-01', '2013-08-09', 42.13)
    ,(1, '2013-06-17', '2013-06-25', 16.33)
    ,(1, '2013-06-25', '2014-02-23', 16.45)
    ,(2, '2013-08-09', '2013-08-10', 45.88)
    ,(2, '2013-08-10', '2014-03-01', 34.98)
    ,(2, '2014-03-01', NULL, 45.18)
    ,(1, '2014-02-23', NULL, 16.65)
    ,(3, '2013-01-01', NULL, 17.77);


SELECT *
FROM dbo.ProductPrices;
```

Our final SELECT returns these rows from the **ProductPrices** table, showing that all is well:

```
ProductID  EffectiveStartDT         EffectiveEndDT           ProductPrice
1          2013-06-04 00:00:00.000  2013-06-17 00:00:00.000  15.25
1          2013-06-17 00:00:00.000  2013-06-25 00:00:00.000  16.33
1          2013-06-25 00:00:00.000  2014-02-23 00:00:00.000  16.45
1          2014-02-23 00:00:00.000  NULL                     16.65
2          2013-05-01 00:00:00.000  2013-08-09 00:00:00.000  42.13
2          2013-08-09 00:00:00.000  2013-08-10 00:00:00.000  45.88
2          2013-08-10 00:00:00.000  2014-03-01 00:00:00.000  34.98
2          2014-03-01 00:00:00.000  NULL                     45.18
3          2013-01-01 00:00:00.000  NULL                     17.77
```

The NULL **EffectiveEndDT** on each of the products indicates an open-ended time period, meaning that final price extends out into the future without bound.

Suppose we wish to find the price of a particular product on a specific date. Let's examine a couple of cases:

```
DECLARE @DateOfInterest DATETIME = '2013-12-31';

-- Find the price effective on 2013-12-31
SELECT a.ProductID, a.ProductName, b.ProductPrice
FROM dbo.Products a
JOIN dbo.ProductPrices b
ON a.ProductID = b.ProductID
WHERE a.ProductID = 1 AND
    b.EffectiveStartDT <= @DateOfInterest AND
    ISNULL(b.EffectiveEndDT, a.DiscontinuedDT) > @DateOfInterest;

-- Discontinue ProductID=1 at the end of 2014
UPDATE dbo.Products
SET DiscontinuedDT = '2015-01-01'
WHERE ProductID = 1;

SELECT @DateOfInterest = '2015-01-01';

-- Find the price effective on 2015-01-01
SELECT a.ProductID, a.ProductName, b.ProductPrice
FROM dbo.Products a
JOIN dbo.ProductPrices b
ON a.ProductID = b.ProductID
WHERE a.ProductID = 1 AND
    b.EffectiveStartDT <= @DateOfInterest AND
    ISNULL(b.EffectiveEndDT, a.DiscontinuedDT) > @DateOfInterest;
```

We locate the proper time record by finding the one for that product that is greater than or equal to the start date and less than the end date. Our first query reports the product price for **ProductID=1** on 2013-12-31 as:

```
ProductID  ProductName  ProductPrice
1          Mouse Trap   16.45
```

Suppose we no longer want to sell mouse traps in 2015 because after that time we only want to sell better mouse traps. We can set the discontinued date of **`Product=1`** to be 2015-01-01, and then run with the corresponding date of interest accordingly. The second SELECT query shows this and returns no rows, indicating no price for the product on that date because it has been discontinued.

You do need to make sure that the discontinued date is always later than the latest effective pricing record's start date. But that isn't really the point of today's exercise.

Our CONSTRAINTs Maintain Data Integrity for our ProductPrices

These constraints in our simple product/price example are enforcing the essential business rules:

- All products with a current price must have a single entry that has a NULL end date
- An effective end date is always greater than the start date
- A valid price is positive and greater than zero
- All products must exist in the referenced products table
- No product should have more than one row with a NULL end-date
- No product should ever have more than one relevant row for any point in time.
- For every non-NULL end-date on a row, the next row for that product must have that end-date as the new start-date.
- No product can have duplicate start-dates or duplicate end-dates.

If we didn't enforce these rules, we would allow bad data, such as having an end date overlapping a start date on the following row. This would allow an item to have more than one price at any point in time. Although this should be caught in the application's front end, bad data can get through for one reason or another. When that happens, it must be identified and constraints set in this way will block it.

We have not yet tested the CONSTRAINTs we've created to make sure that they enforce data integrity according to the business rules, so let's do that now.

All products must exist in the referenced products table

```
-- Fails because ProductID=4 doesn't exist (pp_fk2)
INSERT INTO dbo.ProductPrices
(
    ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice
)
VALUES (4, '2014-01-01', NULL, 21.33);

-- Now we create ProductID=4
INSERT INTO dbo.Products (ProductName)
VALUES ('Another trap');

BEGIN TRANSACTION T1;

-- This now works because ProductID = 4 exists
INSERT INTO dbo.ProductPrices
(
    ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice
)
VALUES (4, '2014-01-01', NULL, 21.33);

SELECT *
FROM dbo.ProductPrices
WHERE ProductID = 4;

-- But we will roll it back to keep this row out of our test data
ROLLBACK TRANSACTION T1;
```

In the above example, the first INSERT fails because **ProductID=4** does not exist in our Products table (CONSTRAINT **pp_fk2**). Once we've inserted **ProductID=4**, the exact same insert works just fine. Note how at the end we rolled back the transaction we created before our second insert into **ProductPrices**. We've done this to maintain the previously noted entries in that table, and we will continue to do this for any T-SQL query that would change the data without violating one of our constraints.

Since the INSERT of **ProductID=4** into the **ProductPrices** table above does not fail, it is possible to seed each product with an open-ended price without any problems.

An effective end date is always greater than the start date

```
-- An effective end date that is less than the start date
INSERT INTO dbo.ProductPrices
(
    ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice
)
VALUES (4, '2014-01-01', NULL, 12.33)
    ,(4, '2014-06-04', '2014-01-01', 18.25);
```

This INSERT would violate **pp_ck1**.

A valid price is positive and greater than zero

```
-- A product price that is zero
INSERT INTO dbo.ProductPrices
(
    ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice
)
VALUES (4, '2014-01-01', NULL, 0);
```

This insert violates **pp_ck2**. **ProductID=4** remains in our Products table because we left it there in the previous set of queries. Remember at this time there are still no price records for **ProductID=4** (both insert attempts failed).

No product can have duplicate start-dates or end-dates

```
-- A duplicate effective start date
INSERT INTO dbo.ProductPrices
(
    ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice
)
VALUES (4, '2013-06-04', '2013-12-31', 18.25)
     ,(4, '2013-06-04', NULL, 19.22);
```

This insert fails on the PRIMARY KEY CONSTRAINT (**pp_pk**) because we're trying to insert the same effective start date more than once.

```
  -- A duplicate effective end date
INSERT INTO dbo.ProductPrices
(
    ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice
)
VALUES (4, '2013-06-04', '2013-12-31', 18.25)
    ,(4, '2013-06-30', '2013-12-31', 19.22);
```

This fails because we're trying to insert the same effective end date twice, violating our UNIQUE CONSTRAINT (**pp_u1**). Note that were both end dates NULL, the UNIQUE constraint would still fail (only one NULL end date is allowed for each product).

For every non-NULL end-date on a row, the next row for that product must have that end-date as the new start-date

```
-- A gap between the first and second records' effective dates
INSERT INTO dbo.ProductPrices
(
    ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice
)
VALUES (4, '2013-06-04', '2013-12-31', 18.25)
    ,(4, '2014-01-01', '2014-08-09', 22.13)
    ,(4, '2014-08-09', NULL, 25.13);

-- An overlap between the first and second record's effective dates
INSERT INTO dbo.ProductPrices
(
    ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice
)
VALUES (4, '2013-06-04', '2014-01-01', 18.25)
    ,(4, '2013-12-30', '2014-08-09', 22.13)
    ,(4, '2014-08-09', NULL, 25.13);
```

These two fail the self-referencing FOREIGN KEY CONSTRAINT (**pp_fk1**) because of either a gap or an overlap in the end-to-start date relationship, meaning those rows we're trying to insert don't have contiguous dates.

Maintaining our ProductPrices – The Dilemma

We've already shown that it is pretty simple to insert a new product's prices into our table. But what of the case where we need to do any of the following:

- Insert a new row where a product already has a price record.
- Change a product price for an existing row.
- Change the effective start date for an existing product/price.
- Delete the row for an effective date.

Some of these things are easy to do, while some are less trivial. For example:

```
BEGIN TRANSACTION T1;

-- Case 1: Insert a new price row for an existing product
INSERT INTO dbo.ProductPrices
(
    ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice
)
VALUES (1, '2013-06-01', '2013-06-04', 18.25);
```

The insert case above works only because we are inserting the first row (prior to the earliest effective date) and because we know what the next effective start date is (2013-06-04). In order to insert a row anywhere else, it is necessary to modify at least one other row's effective end date, and that is not something that can be done with an INSERT.

```
-- Case 2: Change a product price
UPDATE dbo.ProductPrices
SET ProductPrice = 14.25
WHERE ProductID = 1 AND EffectiveStartDT = '2013-06-17'

SELECT *
FROM dbo.ProductPrices
WHERE ProductID = 1;
```

The update in Case 2 above works because all we are changing is the product's price. Had we tried instead to modify **ProductID**, **EffectiveStartDT** or **EffectiveEndDT** for that row, it would have violated one of the CONSTRAINTS.

```
-- Case 3: Change the effective start date for the first product/price
UPDATE dbo.ProductPrices
SET EffectiveStartDT = '2013-05-01'
WHERE ProductID = 1 AND EffectiveStartDT = '2013-06-01'

SELECT *
FROM dbo.ProductPrices
WHERE ProductID = 1;
```

While it is possible to change the effective start date of the first row (earliest effective start date), it is not possible to modify the effective start date of any other row. In order to do that, once again it becomes necessary to update the effective end date of at least one other row, and while that could be done with a carefully crafted UPDATE, it is a non-trivial exercise.

```
-- Case 4: Delete the row for the first product/price
DELETE FROM dbo.ProductPrices
WHERE ProductID = 1 AND EffectiveStartDT = '2013-05-01'

SELECT *
FROM dbo.ProductPrices
WHERE ProductID = 1;

ROLLBACK TRANSACTION T1;
```

Once again, it is possible to delete the first effective start date row for any product, but the same cannot be done if the row to be deleted is in the middle or at the end of the set without violating one or more CONSTRAINTs. Unless of course it was possible to at the same time modify the effective end dates of one or more rows using DELETE (which it is not).

The astute SQL-ers will at this time raise their collective hands and say, “hey wait a minute. All those things that you said INSERT, UPDATE and DELETE cannot do could be done by using MERGE instead.” And you would be right! In fact, that is precisely how Alex proposed to manage maintenance of his history tables in the referenced articles. He simply developed a stored procedure to maintain the tables using a very clever MERGE.

So clearly we could do the same thing here, albeit a little differently due to the fact that we’ve constructed our CONSTRAINTs somewhat differently.

To demonstrate this, we’ll starting with an empty ProductPrices table and insert some sample data.

```
INSERT INTO dbo.ProductPrices (ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice)
VALUES (1, '2013-01-01', '2013-04-01', 11.11)
    ,(1, '2013-04-01', '2013-06-01', 12.12)
    ,(1, '2013-06-01', NULL, 13.13);
```

And we can now simply insert a new current price, modifying the existing price by changing the NULL to the end date and inserting a new row with the current price. We have to supply the row to be updated and the new row

```
MERGE dbo.ProductPrices t
USING (
VALUES (1,'2013-06-01','2013-06-05',14.14),
          (1,'2013-06-05', NULL, 14.14)
) s (ProductID, EffectiveStartDT,EffectiveEndDT, ProductPrice)
  ON s.ProductID=t.ProductID and s.EffectiveStartDT=t.EffectiveStartDT
    WHEN MATCHED THEN UPDATE SET EffectiveEndDT = s.EffectiveEndDT
    WHEN NOT MATCHED THEN INSERT (ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice)
VALUES(s.ProductID, s.EffectiveStartDT, s.EffectiveEndDT, s.ProductPrice);
```

This is fine where the SQL is hand-crafted, or you are using an isolation layer of stored procedures. What if the application has direct access to this table and the ORM (Object Relational Mapper), or UI Widget, being used is able to use only simple SELECT, INSERT, UPDATE and UPDATE statements?

In cases like this it would be impossible to impose any requirement on the application to do a MERGE statement.

We also may not want our application to have to decide what the correct effective end date is when we insert, update or delete any row in our product prices table. Since the title of this article includes the phrase “self-maintaining,” we want our

SQL to do just that. Is it possible to isolate all the complexity from the application, yet allow them to access the table directly to maintain it?

While that sounds like a pretty tall order, it probably won't surprise you that I've found a way to do all of this thanks to the wonders of a SQL 2012 LEAD analytical function.

Review of the SQL 2012 LEAD Analytical Function

Let's show a quick and simple example of LEAD when applied to the **ProductPrices** table:

```
SELECT *
    ,NextEffectiveStartDT = LEAD(EffectiveStartDT, 1, 0) OVER
        (PARTITION BY ProductID ORDER BY EffectiveStartDT)
FROM dbo.ProductPrices;
```

The results produced by this SELECT are:

```
ProductID  EffectiveStartDT         EffectiveEndDT           ProductPrice  NextEffectiveStartDT
1          2013-06-04 00:00:00.000  2013-06-17 00:00:00.000  15.25         2013-06-17 00:00:00.000
1          2013-06-17 00:00:00.000  2013-06-25 00:00:00.000  16.33         2013-06-25 00:00:00.000
1          2013-06-25 00:00:00.000  2014-02-23 00:00:00.000  16.45         2014-02-23 00:00:00.000
1          2014-02-23 00:00:00.000  NULL                     16.65         1900-01-01 00:00:00.000
2          2013-05-01 00:00:00.000  2013-08-09 00:00:00.000  42.13         2013-08-09 00:00:00.000
2          2013-08-09 00:00:00.000  2013-08-10 00:00:00.000  45.88         2013-08-10 00:00:00.000
2          2013-08-10 00:00:00.000  2014-03-01 00:00:00.000  34.98         2014-03-01 00:00:00.000
2          2014-03-01 00:00:00.000  NULL                     45.18         1900-01-01 00:00:00.000
3          2013-01-01 00:00:00.000  NULL                     17.77         1900-01-01 00:00:00.000
```

Here we have used the optional second and third arguments of LEAD:

- The second argument: the offset (number of rows) to look ahead to retrieve the value for **EffectiveStartDT**.
- The third argument: by setting the third argument to 0, when there is no following row the **NextEffectiveStartDT** is set to 1900-01-01, which works quite well for the DATETIME data type, but not for other date or time data types. For those you'd need to use an explicit date, like '1900-01-01'. Ultimately, you should use any date that isn't expected to be within the time period of your effective dates.

For each case except the latest effective start date row, the **NextEffectiveStartDT**matches the **EffectiveStartDT** of the following row. This will be quite useful in what we're about to propose.

Building Blocks of a TRIGGER to Support Self-maintenance of our Effective Dates

The title of this subsection gives away the show, however as always the devil is in the details.

Before we propose a TRIGGER (an INSTEAD OF TRIGGER), let's review a couple of features that are available in all T-SQL TRIGGERS, specifically the virtual tables.

- **INSERTED** – The INSERTED virtual table contains one row for each row to be inserted into the table at the time the TRIGGER is fired. In the case of an INSTEAD OF TRIGGER, these rows have not yet been inserted. On a DELETE event, the INSERTED virtual table contains no rows, but on INSERT and UPDATE events it will always have at least one row.
- **DELETED** – The DELETED virtual table contains one row for each row to be deleted from the table at the time the TRIGGER is fired. In the case of an INSTEAD OF TRIGGER, these rows have not yet been deleted. On an INSERT event, the DELETED virtual table contains no rows, but on DELETE and UPDATE events it will always have at least one row.

It is also important to note that on an UPDATE event, both INSERTED and DELETED tables contain precisely the same number of rows. Both tables have all of the columns that correspond to the underlying table on which the TRIGGER is defined.

Suppose that we're inside of a TRIGGER, and we run the following query:

```
WITH TargetPrices (ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice) AS
(
    SELECT ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice
    FROM dbo.ProductPrices a
    WHERE EXISTS
    (
        SELECT 1
        FROM INSERTED x
        WHERE a.ProductID = x.ProductID
        UNION ALL
        SELECT 1
        FROM DELETED x
        WHERE a.ProductID = x.ProductID
    )
)
SELECT *
FROM TargetPrices;
```

It should be reasonably obvious, given what we've said about the INSERTED and DELETED virtual tables above, that this query will produce a results set that includes all rows from **ProductPrices** for any **ProductID** that is contained in either of the INSERTED or DELETED virtual tables. We won't show that set, and if you're unclear on this you may want to construct your own TRIGGER and put this SELECT statement in it to see the results.

We've called our Common Table Expression (CTE) **TargetPrices** for a reason. This will be the target of the actual maintenance work we're going to perform. However we will impose one additional requirement on our TRIGGER, and that is that it should only "touch" (maintain) rows that must be maintained based on the underlying operation. And that's usually going to mean not all of the rows in **TargetPrices** should be touched. We'll see how this can be done in a moment.

Let's now look at another interesting query, which will probably require some examples to understand it. Again, we must assume we are operating inside of our TRIGGER for INSERTED and DELETED to be defined, but we can simulate that case using CTEs. This also happens to be a useful way to simulate code executing in a TRIGGER that uses the INSERTED/DELETED tables, in case you've ever struggled with that before.

```
WITH INSERTED (ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice) AS
(
    SELECT 1, '2013-06-10', NULL, CAST(55.55 AS MONEY)
),
    DELETED (ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice) AS
(
    SELECT 1, 0, NULL, CAST(0 AS MONEY)
    WHERE 1=0                   -- This gives us 0 rows in the DELETED table
),
    TargetPrices AS
(
    SELECT ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice
    FROM dbo.ProductPrices a
    WHERE EXISTS
    (
        SELECT 1
        FROM INSERTED x
        WHERE a.ProductID = x.ProductID
        UNION ALL
        SELECT 1
        FROM DELETED x
        WHERE a.ProductID = x.ProductID
    )
)
SELECT ProductID, EffectiveStartDT, ProductPrice
FROM TargetPrices
UNION ALL
SELECT ProductID, EffectiveStartDT, ProductPrice
FROM INSERTED
EXCEPT
SELECT ProductID, EffectiveStartDT, ProductPrice
```

```
FROM DELETED;
```

The INSERTED CTE contains one row that we wish to insert and the DELETED CTE contains no rows, simulating what might happen on a single row insert within the TRIGGER. The results set produced is this. If there were rows in the DELETED table, those corresponding rows would be removed from the resulting set by the EXCEPT.

```
ProductID  EffectiveStartDT         ProductPrice
1          2013-06-04 00:00:00.000  15.25
1          2013-06-10 00:00:00.000  55.55
1          2013-06-17 00:00:00.000  16.33
1          2013-06-25 00:00:00.000  16.45
1            2014-02-23 00:00:00.000  16.65
```

Notice how the new effective start date we want to insert (2013-06-10) appears in the results set.

Now let's consider some other cases. The first simulates updating a price for an existing row (we'll show only the INSERTED and DELETED CTEs for brevity).

```
WITH INSERTED (ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice) AS
(
    SELECT 1, '2013-06-17', '2013-06-25', CAST(16.88 AS MONEY)
),
    DELETED (ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice) AS
(
    SELECT 1, '2013-06-17', '2013-06-25', CAST(16.33 AS MONEY)
```

Both INSERTED and DELETED include the effective start date of 2013-06-17 for ProductID=1, with DELETED showing the old price and INSERTED showing the new price (the same way the TRIGGER would populate the virtual tables). These results appear as follows (using the full query above).

```
ProductID  EffectiveStartDT         ProductPrice
1          2013-06-04 00:00:00.000  15.25
1          2013-06-17 00:00:00.000  16.88
1          2013-06-25 00:00:00.000  16.45
1          2014-02-23 00:00:00.000  16.65
```

Note how the target row (effective start date = 2013-06-17) shows only the new product price.

The next case shows the same two CTEs if we were attempting to update the 2013-06-17 record's effective start date, without wanting to concern ourselves with the effective end date.

```
WITH INSERTED (ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice) AS
(
    SELECT 1, '2013-06-16', NULL, CAST(16.33 AS MONEY)
),
    DELETED (ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice) AS
(
    SELECT 1, '2013-06-17', '2013-06-25', CAST(16.33 AS MONEY)
```

The full query would produce these results, again eliminating the effective start date row for 2013-06-17.

```
ProductID  EffectiveStartDT         ProductPrice
1          2013-06-04 00:00:00.000  15.25
1          2013-06-16 00:00:00.000  16.33
1          2013-06-25 00:00:00.000  16.45
1          2014-02-23 00:00:00.000  16.65
```

These results are the final rows we'd expect.

One more, to illustrate the case of a DELETE.

```
WITH INSERTED (ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice) AS
(
    SELECT 1, NULL, NULL, CAST(16.33 AS MONEY)
    WHERE 1=0                   -- This gives us 0 rows in the INSERTED table
),
    DELETED (ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice) AS
(
    SELECT 1, '2013-06-17', '2013-06-25', CAST(16.33 AS MONEY)
```

These results are once again the three rows we'd like to be left with, excluding of course the requisite, contiguous effective end dates.

```
ProductID  EffectiveStartDT         ProductPrice
1          2013-06-04 00:00:00.000  15.25
1          2013-06-25 00:00:00.000  16.45
1          2014-02-23 00:00:00.000  16.65
```

Now we will modify our final query slightly to recalculate (adjust) what the effective end date should be for one of these cases (we'll choose the third case where we updated the effective start date for 2013-06-17), using the magic of the SQL 2012 LEAD analytical function.

```
WITH INSERTED (ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice) AS
(
    SELECT 1, '2013-06-16', NULL, CAST(16.33 AS MONEY)
),
    DELETED (ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice) AS
(
    SELECT 1, '2013-06-17', '2013-06-25', CAST(16.33 AS MONEY)
)
,
    TargetPrices AS
(
    SELECT ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice
    FROM dbo.ProductPrices a
    WHERE EXISTS
    (
        SELECT 1
        FROM INSERTED x
        WHERE a.ProductID = x.ProductID
        UNION ALL
        SELECT 1
        FROM DELETED x
        WHERE a.ProductID = x.ProductID
    )
)
SELECT ProductID, EffectiveStartDT
    ,NextEffectiveStartDT=LEAD(a.EffectiveStartDT, 1, 0) OVER
        (PARTITION BY a.ProductID ORDER BY a.EffectiveStartDT)
    -- All time-sensitive attributes
    ,ProductPrice
FROM
(
    SELECT ProductID, EffectiveStartDT
        -- All time-sensitive attributes
        ,ProductPrice
    FROM TargetPrices
    UNION ALL
    SELECT ProductID, EffectiveStartDT
        -- All time-sensitive attributes
        ,ProductPrice
    FROM INSERTED
    EXCEPT
    SELECT ProductID, EffectiveStartDT        -- All time-sensitive attributes
        ,ProductPrice
    FROM DELETED
) a;
```

This produces a result set that has the contiguous end date saved in the NextEffectiveStartDT column:

```
ProductID  EffectiveStartDT         NextEffectiveStartDT     ProductPrice
1          2013-06-04 00:00:00.000  2013-06-16 00:00:00.000  15.25
1          2013-06-16 00:00:00.000  2013-06-25 00:00:00.000  16.33
1          2013-06-25 00:00:00.000  2014-02-23 00:00:00.000  16.45
1          2014-02-23 00:00:00.000  1900-01-01 00:00:00.000  16.65
```

Note that for the time being, the last row doesn't have a NULL effective end date, but we'll address that later.

By now you may have deduced what we are about to propose, which is a single INSTEAD OF TRIGGER that fires on INSERT, UPDATE and DELETE, which does its work using a MERGE statement.

The TRIGGER to Self-maintain Contiguous, Effective Dates

Without any preamble, we'll list out our TRIGGER now.

```
CREATE TRIGGER dbo.MaintainEffectiveEndDT
ON dbo.ProductPrices
INSTEAD OF INSERT, UPDATE, DELETE
AS
BEGIN

    PRINT 'INSTEAD OF INSERT/UPDATE/DELETE TRIGGER';

    WITH TargetPrices AS
    (
        -- All Product/Price rows where a ProductID appears in either INSERTED or
        -- DELETED virtual tables
        SELECT ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice
        FROM dbo.ProductPrices a
        WHERE EXISTS
        (
            SELECT 1
            FROM INSERTED x
            WHERE a.ProductID = x.ProductID
            UNION ALL
            SELECT 1
            FROM DELETED x
            WHERE a.ProductID = x.ProductID
        )
    ),
        SourcePrices AS
    (
        -- This contains the rows we'll be attempting to MERGE against the Targer Prices
        -- with an additional column for the adjusted effective end date
        SELECT ProductID, EffectiveStartDT
            ,NextEffectiveStartDT=LEAD(a.EffectiveStartDT, 1, 0) OVER
                (PARTITION BY a.ProductID ORDER BY a.EffectiveStartDT)
            -- All time-sensitive attributes
            ,ProductPrice
        FROM
        (
            SELECT ProductID, EffectiveStartDT
                -- All time-sensitive attributes
                ,ProductPrice
            FROM TargetPrices
            UNION ALL
            SELECT ProductID, EffectiveStartDT
                -- All time-sensitive attributes
                ,ProductPrice
            FROM INSERTED
            EXCEPT
            SELECT ProductID, EffectiveStartDT
```

```
                -- All time-sensitive attributes
                ,ProductPrice
            FROM DELETED
        ) a
    )
    -- Perform the merge on TargetPrices from SourcePrices
    MERGE TargetPrices t
    USING SourcePrices s
    -- Matching on the PRIMARY KEY of the ProductPrices table
    ON s.ProductID = t.ProductID AND
        s.EffectiveStartDT = t.EffectiveStartDT
    -- On a match, only update the row if something has changed
    WHEN MATCHED AND
        (s.NextEffectiveStartDT <> ISNULL(t.EffectiveEndDT, 0) OR
        -- Any of the time-sensitive attributes
        s.ProductPrice <> t.ProductPrice)
    THEN UPDATE
        SET EffectiveEndDT = NULLIF(s.NextEffectiveStartDT, 0)
            -- Include here all of the time-sensitive attributes
            ,ProductPrice = s.ProductPrice -- ISNULL(s.ProductPrice, t.ProductPrice)
    WHEN NOT MATCHED -- BY TARGET
    -- Insert the new row
    THEN INSERT
        (ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice)
    -- Using the calculated effective end date
    VALUES
        (s.ProductID, s.EffectiveStartDT, NULLIF(s.NextEffectiveStartDT, 0), s.ProductPrice)
    -- If a row is not in our source and it is in the DELETED virtual table
    -- that means it needs to be deleted from the target
    WHEN NOT MATCHED BY SOURCE AND EXISTS
        (
            SELECT 1
            FROM DELETED x
            WHERE t.ProductID = x.ProductID AND t.EffectiveStartDT = x.EffectiveStartDT
        )
    THEN DELETE;

END
```

Hopefully, the comments and the previous section's examples of LEAD should explain the logic sufficiently to get a basic understanding of what this TRIGGER is doing. However we'll provide some additional explanation because the MERGE itself is not particularly straightforward:

- The results from the **TargetPrices** and **SourcePrices** CTEs were described above.
- NULLIF appears in a couple of places. These are used to convert our invalid dates (1900-01-01) to NULL when it is appropriate to do so.
- Our matching criteria (**ProductID** and **EffectiveStartDT**) is always the PRIMARY KEY for the underlying table.
- The WHEN MATCHED clause identifies cases where we have a matching Source=Target row but something has changed – either **EffectiveEndDT**is different than the calculated, adjusted end date (=**NextEffectiveStartDT**), or the **ProductPrice** (or any of the other time sensitive attributes were they present on this row). This is one way that we're narrowing the actual rows affected by our TRIGGER, to only those we really need to touch.
- The WHEN NOT MATCHED (BY TARGET) clause identifies cases where an insert is required. This is done directly from the source, using **NextEffectiveStartDT** (our adjusted, calculated effective end date) as the effective end date for the inserted row.
- The WHEN NOT MATCHED BY SOURCE clause identifies rows in our target to be deleted, but only when those PRIMARY KEYs exist in the DELETED virtual table.

You may also be wondering how this is going to work, for those cases above that we said couldn't be done for the respective INSERT, UPDATE and DELETE cases. The reason it will work is that within the MERGE statement, all necessary adjustments are made to the effective end dates prior to the process writing anything to the table, and SQL Server validating the CONSTRAINTs. Maybe it is simply best to see it in action.

Testing our Self-maintaining Effective Dates TRIGGER

Let's start with some cases of INSERTs. In the first, we'll truncate our table and restore the original target row set without having to specify the effective end dates for any of the inserted rows (the TRIGGER does this for us).

```
TRUNCATE TABLE dbo.ProductPrices;

INSERT INTO dbo.ProductPrices
(
    ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice
)
VALUES (1, '2013-06-04', NULL, 15.25)
    ,(2, '2013-05-01', NULL, 42.13)
    ,(1, '2013-06-17', NULL, 16.33)
    ,(1, '2013-06-25', NULL, 16.45)
    ,(2, '2013-08-09', NULL, 45.88)
    ,(2, '2013-08-10', NULL, 34.98)
    ,(2, '2014-03-01', NULL, 45.18)
    ,(1, '2014-02-23', NULL, 16.65)
    ,(3, '2013-01-01', NULL, 17.77);

SELECT *
FROM dbo.ProductPrices;
```

The results you should see, are the same as the original results we told you to keep in mind above, but they're shown again here to keep them fresh in your mind.

```
ProductID  EffectiveStartDT         EffectiveEndDT           ProductPrice
1          2013-06-04 00:00:00.000  2013-06-17 00:00:00.000  15.25
1          2013-06-17 00:00:00.000  2013-06-25 00:00:00.000  16.33
1          2013-06-25 00:00:00.000  2014-02-23 00:00:00.000  16.45
1          2014-02-23 00:00:00.000  NULL                     16.65
2          2013-05-01 00:00:00.000  2013-08-09 00:00:00.000  42.13
2          2013-08-09 00:00:00.000  2013-08-10 00:00:00.000  45.88
2          2013-08-10 00:00:00.000  2014-03-01 00:00:00.000  34.98
2          2014-03-01 00:00:00.000  NULL                     45.18
3          2013-01-01 00:00:00.000  NULL                     17.77
```

Let's look at some additional cases of INSERT.

```
BEGIN TRANSACTION T1;

INSERT INTO dbo.ProductPrices
(
    ProductID, EffectiveStartDT, EffectiveEndDT, ProductPrice
)
-- A row in the middle of ProductID=1
VALUES (1, '2013-06-18', NULL, 21.50)
-- Another row inserted in the middle of ProductID=1
    ,(1, '2013-06-28', NULL, 33.50)
-- A row at the beginning of ProductID=2
    ,(2, '2013-01-01', NULL, 22.50)
-- A row at the end of ProductID=3
    ,(3, '2014-01-01', NULL, 12.22);

SELECT *
FROM ProductPrices;

ROLLBACK TRANSACTION T1;
```

The results below highlight the rows that were inserted (in yellow) and those whose effective end date were adjusted (in green).

```
ProductID  EffectiveStartDT         EffectiveEndDT           ProductPrice
1          2013-06-04 00:00:00.000  2013-06-17 00:00:00.000  15.25
1          2013-06-17 00:00:00.000  2013-06-18 00:00:00.000  16.33
1          2013-06-18 00:00:00.000  2013-06-25 00:00:00.000  21.50
```

```
1          2013-06-25 00:00:00.000  2013-06-28 00:00:00.000  16.45
1          2013-06-28 00:00:00.000  2014-02-23 00:00:00.000  33.50
1          2014-02-23 00:00:00.000  NULL                     16.65
2          2013-01-01 00:00:00.000  2013-05-01 00:00:00.000  22.50
2          2013-05-01 00:00:00.000  2013-08-09 00:00:00.000  42.13
2          2013-08-09 00:00:00.000  2013-08-10 00:00:00.000  45.88
2          2013-08-10 00:00:00.000  2014-03-01 00:00:00.000  34.98
2          2014-03-01 00:00:00.000  NULL                     45.18
3          2013-01-01 00:00:00.000  2014-01-01 00:00:00.000  17.77
3          2014-01-01 00:00:00.000  NULL                     12.22
```

If we count up the yellow and green highlighted rows, we get seven, and that interestingly corresponds to precisely the number of rows reported in the SQL Server Management Studio (SSMS) Messages pane as having been touched by our TRIGGER:

```
INSTEAD OF INSERT/UPDATE/DELETE TRIGGER

(7 row(s) affected)
```

Now let's try some updates, doing so by using a CTE to consolidate the information to update.

```
BEGIN TRANSACTION T1;

WITH OurUpdates (ProductID, EffectiveStartDT, NewEffectiveStartDT, ProductPrice) AS
(
    -- Change the effective start date on a row
    SELECT 1, CAST('2013-06-25' AS DATETIME)
        ,CAST('2013-06-16' AS DATETIME), CAST(NULL AS MONEY)
    UNION ALL
    -- Change the price on a row
    SELECT 1, '2013-06-17', NULL, 16.88
    UNION ALL
    -- Move a row to the beginning
    SELECT 2, '2013-08-10', '2013-01-01', NULL
    UNION ALL
    -- Move a row to the End and change the price
    SELECT 2, '2013-03-01', '2014-05-01', 11.11
)
UPDATE a
SET EffectiveStartDT    = ISNULL(b.NewEffectiveStartDT, a.EffectiveStartDT)
    ,ProductPrice       = ISNULL(b.ProductPrice, a.ProductPrice)
FROM dbo.ProductPrices a
JOIN OurUpdates b
ON a.ProductID = b.ProductID AND a.EffectiveStartDT = b.EffectiveStartDT;

SELECT *
FROM ProductPrices
WHERE ProductID IN (1, 2);

ROLLBACK TRANSACTION T1;
```

An important detail of this UPDATE is the use of ISNULL (or COALESCE for the purist). You should do this to ensure (for example) that moving a row without specifying the**ProductPrice**, retains the original product price.

We'll see this from these results, where changes are once again highlighted (yellow for moved or updated rows, and green for adjusted effective end dates):

```
ProductID  EffectiveStartDT         EffectiveEndDT           ProductPrice
1          2013-06-04 00:00:00.000  2013-06-16 00:00:00.000  15.25
1          2013-06-16 00:00:00.000  2013-06-17 00:00:00.000  16.45        <-- was start date 2013-06-25
1          2013-06-17 00:00:00.000  2014-02-23 00:00:00.000  16.88
1          2014-02-23 00:00:00.000  NULL                     16.65
2          2013-01-01 00:00:00.000  2013-05-01 00:00:00.000  34.98        <-- was start date 2013-08-10
2          2013-05-01 00:00:00.000  2013-08-09 00:00:00.000  42.13
2          2013-08-09 00:00:00.000  2014-05-01 00:00:00.000  45.88
2          2014-05-01 00:00:00.000  NULL                     11.11        <-- was start date 2014-03-01
```

In this case, if we specified the **ProductPrice** as NULL, it is an indicator that we didn't want to change it. In this case, the SSMS Messages pane reports this action for the TRIGGER:

```
INSTEAD OF INSERT/UPDATE/DELETE TRIGGER

(9 row(s) affected)
```

This is three more than the number of highlighted rows, because in the cases where a row is moved (effective start date is changed), it involves a delete and an insert.

Let's now try some deletes, once again consolidated into a CTE.

```
BEGIN TRANSACTION T1;

WITH OurDeletes (ProductID, EffectiveStartDT) AS
(
    -- First row for ProductID=2
    SELECT 2, CAST('2013-05-01' AS DATETIME)
    UNION ALL
    -- Last row for ProductID=2
    SELECT 2, '2014-03-01'
    UNION ALL
    -- A row from the middle of ProductID=1
    SELECT 1, '2013-06-17'
    UNION ALL
    -- Another row from the middle of ProductID=1
    SELECT 1, '2013-06-25'
)
DELETE FROM a
FROM dbo.ProductPrices a
JOIN OurDeletes b
ON a.ProductID = b.ProductID AND a.EffectiveStartDT = b.EffectiveStartDT;

SELECT *
FROM ProductPrices
WHERE ProductID IN (1, 2);

ROLLBACK TRANSACTION T1;
```

The final result set in this case looks like this, where I can't exactly highlight the rows that have been deleted (four of those) but I can highlight in green the ones where the effective end date was adjusted.

```
ProductID  EffectiveStartDT         EffectiveEndDT           ProductPrice
1          2013-06-04 00:00:00.000  2014-02-23 00:00:00.000  15.25
1          2014-02-23 00:00:00.000  NULL                     16.65
2          2013-08-09 00:00:00.000  2013-08-10 00:00:00.000  45.88
2          2013-08-10 00:00:00.000  NULL                     34.98
```

Here the SSMS Messages Pane reports:

```
INSTEAD OF INSERT/UPDATE/DELETE TRIGGER

(6 row(s) affected)
```

Which makes sense considering the deleted rows (4) plus the highlighted rows (2).

Now none of this is worth a hoot if our TRIGGER doesn't work for the case of a MERGE, where we want to insert, update and delete all at the same time. So let's give that a try by building a transaction table in our leading CTE.

```
BEGIN TRANSACTION T1;

WITH TransTable ([Action], ProductID, EffectiveStartDT, NewEffectiveStartDT, ProductPrice) AS
(
```

```
    -- Insert a row for ProductID=1
    SELECT 'I', 1, '2013-06-09', NULL, 14.29
    UNION ALL
    -- Insert another row for ProductID=1 (at end)
    SELECT 'I', 1, '2014-06-01', NULL, 18.99
    UNION ALL
    -- Delete a row for ProductID=2 (in the middle)
    SELECT 'D', 2, '2013-08-09', NULL, NULL
    UNION ALL
    -- Delete the row for ProductID=3
    SELECT 'D', 3, '2013-01-01', NULL, NULL
    UNION ALL
    -- Move a row for ProductID=2
    SELECT 'U', 2, '2013-08-10', '2013-04-01', NULL
)
MERGE dbo.ProductPrices t
USING TransTable s
ON s.ProductID = t.ProductID AND s.EffectiveStartDT = t.EffectiveStartDT
WHEN MATCHED AND s.[action] = 'U'
THEN UPDATE
SET EffectiveStartDT = ISNULL(s.NewEffectiveStartDT, t.EffectiveStartDT)
    ,ProductPrice    = ISNULL(s.ProductPrice, t.ProductPrice)
WHEN MATCHED AND s.[action] = 'D'
THEN DELETE
WHEN NOT MATCHED AND s.[action] = 'I'   -- Should be an INSERT
THEN INSERT
    (ProductID, EffectiveStartDT, ProductPrice)
VALUES
    (s.ProductID, s.EffectiveStartDT, s.ProductPrice);

SELECT *
FROM ProductPrices;

ROLLBACK TRANSACTION T1;
```

Let's check to see if the following results are what we expect./p>

```
ProductID  EffectiveStartDT         EffectiveEndDT           ProductPrice
1          2013-06-04 00:00:00.000  2013-06-09 00:00:00.000  15.25
1          2013-06-09 00:00:00.000  2013-06-17 00:00:00.000  14.29
1          2013-06-17 00:00:00.000  2013-06-25 00:00:00.000  16.33
1          2013-06-25 00:00:00.000  2014-02-23 00:00:00.000  16.45
1          2014-02-23 00:00:00.000  2014-06-01 00:00:00.000  16.65
1          2014-06-01 00:00:00.000  NULL                     18.99
2          2013-04-01 00:00:00.000  2013-05-01 00:00:00.000  34.98
2          2013-05-01 00:00:00.000  2014-03-01 00:00:00.000  42.13
2          2014-03-01 00:00:00.000  NULL                     45.18
```

For **ProductID=1**, we inserted two rows which are highlighted in yellow (second and fifth). This caused updates (self-maintenance) of the two preceding rows' effective end dates (highlighted in green). For **ProductID=2** we deleted a row (2013-08-09), which we obviously can't highlight in yellow, but we can highlight in green the row where the effective end date was adjusted (2013-05-01) as a result. Note that we also deleted the single row for **ProductID=3**. In this case there were no other rows where the end date needed adjustment. Finally, for **ProductID=2** we also moved a row (2013-08-10) to a new effective start date (2013-04-01) and this is highlighted in yellow.

In case you've never worked with TRIGGERs in the case of a MERGE statement, they fire depending on what clauses are fired in the MERGE statement (we are talking here about our latest example, and not the MERGE statement in the TRIGGER). Since our CTE contained transactions that caused each of the three clauses to fire (an UPDATE, an INSERT and a DELETE) we expect that our TRIGGER was fired three times. The results in the SSMS Messages pane confirms this.

```
INSTEAD OF INSERT/UPDATE/DELETE TRIGGER

(4 row(s) affected)
INSTEAD OF INSERT/UPDATE/DELETE TRIGGER
```

```
(3 row(s) affected)
INSTEAD OF INSERT/UPDATE/DELETE TRIGGER

(3 row(s) affected)
```

We'll let the interested reader confirm how the row counts for each execution of the TRIGGER came about (hint: the order is INSERT, UPDATE and finally DELETE). To dig a little deeper into the workings of the TRIGGER, you may want to introduce a couple of SELECTs at the beginning of the TRIGGER and then run the MERGE above again to see what's going on.

```
SELECT * FROM INSERTED;
SELECT * FROM DELETED;
```

Some Caveats for this Approach

In a table such as**ProductPrices**, there may be more than one time-sensitive column, although a relational model purist would probably argue this shouldn't be. In the TRIGGER, you'll note that in several places there is a comment that indicates "time-sensitive attributes." In each case, this is a reminder that should you ALTER the table to include more time-sensitive attributes, those must be handled the same way that **ProductPrice** is handled in the various points within that TRIGGER. In other words, you must change the TRIGGER to include the additional columns.

If you are likely to use a system like this in a team, or if you aren't blessed with a perfect memory, it is possible to set a reminder for anyone that modifies (ALTERs) this table. This can be done through a DATABASE TRIGGER, such as the one that follows.

```
CREATE TRIGGER WarnOnTableALTERs
ON DATABASE
FOR ALTER_TABLE
AS
    SET NOCOUNT ON;

    DECLARE @TablesOnWatch TABLE
    (
        Table_Name      VARCHAR(100) NOT NULL
        ,Table_ID       BIGINT NOT NULL
    );

    INSERT INTO @TablesOnWatch SELECT 'ProductPrices', OBJECT_ID('ProductPrices', 'U');

    SELECT [Table Name]=name, Warning='Was ALTERed and has a TRIGGER on watch'
    FROM sys.all_objects a
    JOIN @TablesOnWatch b ON a.name = b.Table_Name
    WHERE a.[object_id] =
        OBJECT_ID(EVENTDATA().value('(/EVENT_INSTANCE/ObjectName)[1]','NVARCHAR(MAX)')
                , 'U')
GO
```

So now, when anyone comes along and ALTERs this table, a warning message is displayed.

```
BEGIN TRANSACTION T1;

ALTER TABLE dbo.ProductPrices
ADD NewTimeSensitiveAttribute INT NULL;

ROLLBACK TRANSACTION T1;

Table Name      Warning
ProductPrices   Was ALTERed and has a TRIGGER on watch
```

Yes you can ROLLBACK an ALTER of a TABLE!

The other caveat to this approach, which also affects Alex's original (stored procedure) approach, has to do with issues reported about using MERGE. See "Use Caution with SQL Server's MERGE Statement" by 18 times nominated (as of this writing) SQL MVP and avid blogger Aaron Bertrand. I cannot honestly say how many of these issues have been addressed in SQL 2012, but many of them may not be pertinent to your case.

Conclusions, Comparisons and Final Words

It is definitely possible to use constraints to enforce the essential business rules of a temporal table. The necessary updates to the data can be done by using SQL Server's MERGE statement, but in several types of application design, this complexity is best hidden from the application.

This article shows how this complexity can be hidden so that even where the application has full direct access to the table, simple CRUD operations can be used, but at the same time the table is still protected from bad data.

I don't believe I've built a better mousetrap here. I have, rather, offered an alternative mousetrap, which may be applicable and/or effective in some of your use cases. Here is a summary of the advantages and disadvantages of the two approaches, as I see it.

Category	Original Approach	Alternate Approach	Remark
Required SQL Server Version	2008	2012	Original approach is limited by the MERGE, although Alex does offer an alternative that works in SQL 2005.
Additional Storage	One Column for Previous End Date	None	
Implementation	Stored Procedure	TRIGGER	It might be possible to implement the original approach in a TRIGGER.
Specifying Values on DML operations	Must specify Effective End Date and Previous Effective End Date	Need not specify Effective End Date	The alternate approach allows the user to issue normal Data Manipulation Language (DML) INSERTs, UPDATEs, DELETEs and MERGEs to modify data in the table without specifying the effective end date. It may be possible to implement this same effect in the original approach if done with a MERGE in a TRIGGER, as done in the alternate approach.
CONSTRAINTs	All data integrity checks are managed through CONSTRAINTs	All data integrity checks are managed through CONSTRAINTs	

With the alternative approach, you don't need to specify any adjusted effective end dates when you perform normal DML actions. No additional storage is taken up by the additional column for the previous end date that Alex proposed.

It would also be possible to replace the LEAD analytical function in the TRIGGER with an OUTER APPLY/SELECT TOP 1/ORDER BY `EffectiveStartDT` construct (which effectively simulates a LEAD), making for a fully SQL 2008 compatible solution. However we will refrain from showing that, leaving it to our interested readers, because it is likely its performance would not be as good as using LEAD.

One point not described in the comparison above is how relatively easy it would be to implement this on top of your existing temporal tables. Simply create the constraints and the trigger on top of your table having effective start/end dates, and assuming there are no constraint failures (data anomalies) when you create them, your basic CRUD operations should pretty much work without code changes.

Thanks for listening folks! I hope you find this approach useful and tell me about your success stories.