Managing time in DB2 with temporal consistency

Enforce time-based referential integrity

Matthias Nicola
Martin Sommerlandt

July 12, 2012

The new temporal features in IBM® DB2® 10 provide rich capabilities for time-based data management. For example, a date range can be assigned to each row of data to indicate when a row is deemed valid by your applications or business. Tables enabled to track such business validity are called application-period temporal tables and their business time periods can be in the past, present, or future. For a parent table and a child table that have a foreign key relationship, the notion of referential integrity can be extended to consider not only the traditional key value but also the business time period of any given row. This article explains such temporal referential integrity and how time-based consistency can be enforced in DB2. The content of this article is applicable to DB2 for z/OS® and DB2 for Linux®, UNIX®, and Windows®; all included SQL samples were tested in DB2 10.1 for LUW.

Introduction to temporal data management with DB2

DB2 10 for z/OS and DB2 10 for Linux, UNIX, and Windows introduce time-based data management that allows you to query and manipulate data in the past, the present, and the future while keeping a complete history of all data changes. The collection of temporal features in DB2 is also known as Time Travel Query.

DB2 supports three types of temporal tables:

- **System-period temporal tables**— Where DB2 transparently keeps a history of old rows that have been updated or deleted over time. With new constructs in the SQL language standard, users can "go back in time" and query the database at any chosen point in the past. This is based on internally assigned system timestamps that DB2 uses to manage system time, also known as transaction time.

- **Application-period temporal tables**— Where applications supply dates or timestamps to describe the business validity of data. New SQL constructs enable applications to insert, query, update, and delete data in the past, present, or future. DB2 automatically applies constraints and row-splits to correctly maintain the application-supplied business time, also known as valid time.
• **Bitemporal tables**— Manage system time and business time. Bitemporal tables combine all the capabilities of system-period and application-period temporal tables. This combination enables applications to manage the business validity of data while DB2 keeps a full history of any updates and deletes.

For the remainder of this article, we assume you are familiar with the basics of application-period temporal tables in DB2. The discussion of temporal referential integrity (RI) also applies to business time in bitemporal tables. You should know how to create and use such tables, and understand how DB2 may perform row-splits when you update or delete data for a specified portion of business time. The article "A Matter of Time: Temporal Data Management in DB2" provides an introduction to these topics.

**What is temporal referential integrity?**

**An example**

In this section, we revisit traditional referential integrity (RI) in a relational database then expand the discussion to temporal RI. Our scenario involves a fictitious business that receives products from suppliers, sells these products to customers, and occasionally runs promotions that offer selected products at discount prices for limited periods of time. To help you understand how you can implement and enforce temporal RI, we use this application scenario and sample data.

**Traditional RI**

**Listing 1** shows two tables involved in managing this business. The parent table, product_avail, contains one row for each available product and identifies the supplier that provides this product. The child table, promotion, has one row for each special offer, defined by a promotion ID, the respective product ID, and the reduced price. Both tables might have additional columns, but for simplicity, these few columns are sufficient for now.

**Listing 1. Simple example of referential integrity between parent and child table**

```sql
CREATE TABLE product_avail(
    prodID     INTEGER NOT NULL,
    supplier   VARCHAR(32),
    PRIMARY KEY(prodID) );

CREATE TABLE promotion(
    promoID    INTEGER NOT NULL,
    prodID     INTEGER NOT NULL,
    price      DECIMAL(10,2),
    PRIMARY KEY(promoID),
    FOREIGN KEY (prodID) REFERENCES product_avail(prodID) );
```

Since a promotion can only be offered for an available product, every row in the promotion table must contain a `prodID` that exists in the `product_avail` table. This condition is called *referential integrity* and is declared and enforced by the foreign key (FK) constraint on the promotion table. Due to this constraint, any attempt to insert or update a promotion with a `prodID` that doesn't exist in the `product_avail` table is rejected by DB2.
Introducing time

In our fictitious business, different suppliers can provide the same product at different times of the year. Similarly, we might choose to run different promotions for the same product at different points in time. Since time is such an important dimension in business, we create the tables from Listing 1 as application-period temporal tables, which means we include a BUSINESS_TIME period in both tables (see Listing 2).

The added BUSINESS_TIME period consists of a pair of date or timestamp columns plus a PERIOD declaration that defines this pair of columns as a period. We also choose to extend the primary key definition with the optional keywords BUSINESS_TIME WITHOUT OVERLAPS, which indicate that multiple rows can have the same prodID value as long as the BUSINESS_TIME periods of these rows do not overlap. The same primary key extension is applied to the promotion table.

Since the prodID column in the product_avail table now can have duplicate values, it can no longer be referenced as a foreign key in the promotion table. We will need to enforce RI differently.

Listing 2. Tables from Listing 1 now as application-period temporal tables

```sql
CREATE TABLE product_avail(
    prodID       INTEGER NOT NULL,
    supplier     VARCHAR(32),
    avail_start  DATE NOT NULL,
    avail_end    DATE NOT NULL,
    PERIOD BUSINESS_TIME (avail_start, avail_end),
    PRIMARY KEY(prodID, BUSINESS_TIME WITHOUT OVERLAPS)  );

CREATE TABLE promotion(
    promoID    INTEGER NOT NULL,
    prodID     INTEGER NOT NULL,
    price      DECIMAL(10,2),
    promo_start  DATE NOT NULL,
    promo_end    DATE NOT NULL,
    PERIOD BUSINESS_TIME (promo_start, promo_end),
    PRIMARY KEY(promoID, BUSINESS_TIME WITHOUT OVERLAPS)  );
```

Figure 1 shows the tables populated with information about three products and two promotions. For example, product 9105 is delivered by supplier company A beginning 1 Jan 2012 and ending on 1 Jun 2012, which means the last day of delivery is 31 May 2012. (Note that periods are always specified in an inclusive-exclusive manner; the specified end dates are no longer part of the validity period.) From 1 Jun onward, product 9015 is provided by company B, ending on 1 Sep. During September and October, we receive no supply of product 9015, but during November and December, this product is available again from company A. When we have no supplier for a product, then that product is considered out of stock.
The primary key (prodID, BUSINESS_TIME WITHOUT OVERLAPS) ensures that no two suppliers deliver the same product at the same time. If we want to allow multiple suppliers to deliver the same product at the same time, we could define the primary key as (prodID, supplier, BUSINESS_TIME WITHOUT OVERLAPS).

The promotion table in Figure 1 defines two promotions for product 9105. The first promotion offers the product for $19.95 from 15 Jan 2012, ending on 15 Mar 2012. That is, the last day of the discounted price is 14 Mar 2012. The second promotion starts on 1 May and ends on 1 Jul 2012, with a price of $16.95. At all other times, the regular price for product 9105 is in effect. That price is stored elsewhere.

**Temporal RI**

The data in Figure 1 fulfills a traditional foreign key relationship between the two tables because each promotion references a product ID that exists in the product_avail table. However, our business must also ensure that a promotion is offered only during a period of time when the respective product is actually available. Imagine the confusion and dissatisfaction among customers if we advertised a discount for a product that isn't available in any of our stores.

More generally speaking, you may wish to enforce a temporal relationship between the business time periods of the parent rows and the child rows in two related tables. Examples of such relationships include the following:

- **Period equality**— The business time period of the child row must be identical to the period of a single parent row. In our example, this condition would enforce that the duration of a promotion always matches the period during which the product is available from a particular supplier. A traditional FK on the product ID and the period start and end columns can enforce this relationship.
- **Period containment**— The business time period of the child row must be fully contained within the periods of one or multiple parent rows.
- **Period containment, single parent**— The business time period of the child row must be fully contained within the period of a single parent row.
- **Overlap**— The business time period of the child row must overlap with the period of a parent row.
- **Start-within**— The business time period of the child row must start within the period of a parent row.
• **Start-after**— The business time period of the child row must start after the period of a parent row.

**Figure 2** visualizes the business time periods of the sample data in **Figure 1** and allows us to examine some of the temporal relationships listed above. Each row of a table is represented by a rectangle whose length represents the duration of the business time period of that row. Rows from the product_avail table are shown in blue, rows from the promotion table are in green.

We can make the following observations in **Figure 2**:

- The promotions with **promoID** 16 and 17 do not fulfill the condition of period equality with their parent rows because the promotion periods are shorter than the periods of their parent rows in the product_avail table.
- Promotion 16 fulfills the period containment condition — and even period containment with a single parent — because its offering period is contained in one period during which product 9105 is provided by supplier company A.
- Promotion 17 satisfies period containment (with multiple parent rows) because its offering period consist of the months May and June where the product is supplied by companies A and B, with no gap in the supply. Promotion 17 does not satisfy period containment with a single parent row.
- Promotions 16 and 17 both fulfill the overlap and the start-within relationships with the first product row

**Figure 2. Timeline of product_avail periods (top, blue) and promotion periods (bottom, green)**

Based on your own use case and application requirements, you need to determine which temporal relationship to enforce between parent and child rows.

For our product and promotion example, we choose **period containment** (with one or multiple parent rows) as the desired relationship to define temporal RI. Hence, the remainder of this article assumes period containment as the notion for temporal RI.

**Temporal RI violations**

Our marketing department expects a decrease in sales during August and September, and decides to offer product 9105 for only $15.95 during that period. The following **INSERT** is issued for promotion 18:

```sql
INSERT INTO promotion (promoID, prodID, price, promo_start, promo_end)
VALUES (18, 9105, '2012-08-01', '2012-10-01');
```
This new promotion does not violate traditional RI because the foreign key 9105 exists in the product_avail table. However, you can see in Figure 3 that promotion 18 violates our chosen notion of temporal RI, that is, period containment. This new promotion runs all through September, but we have no supplier for product 9105 during September (and we keep no stock).

If we had chosen the overlap or the start-within condition to define temporal RI, promotion 18 would not be in violation.

**Figure 3. Promotion 18 violates temporal RI**

We might be able to detect and avoid this temporal RI violation by discovering that the end date of promotion 18, 2012-10-01, is outside any existing business time period for product 9105 in the product_avail table. To rectify the situation, promotion 18 needs to be modified or deleted from the promotion table.

Now let's consider another promotion entered with the following `INSERT` statement for promotion 19:

```sql
INSERT INTO promotion (promoID, prodID, price, promo_start, promo_end)
VALUES (19, 9105, '2012-08-01', '2012-12-01');
```

This promotion starts on 1 Aug 2012 and ends on 1 Dec 2012. Although the start and end date of this promotion each lie within existing product availability periods, temporal RI (period containment) is violated because there is a supplier gap between these two dates. As illustrated in Figure 4, promotion 19 is valid during August, September, October, and November, but we have no supply during September and October.
Managing time in DB2 with temporal consistency

Figure 4. Promotion 18 violates temporal RI

The promotion 19 example illustrates an important fact: If the parent periods for a particular key value can have gaps, containment violations cannot be detected simply by checking that the start date and the end date of a child period are contained in existing parent periods. However, if you know for sure that there are no period gaps between the parent rows with the same key value, period containment is easier to enforce.

Promotion 17 illustrates another important observation: to enforce period containment with one or multiple parent rows it is generally not sufficient to check whether the start and end date of a child period are both contained in the same parent period. Such a check would enforce period containment in a single parent and would reject promotion 17.

Options for enforcing temporal RI

There are typically four options for handling temporal RI:

1. No enforcement. Do not define any constraints when you know temporal RI is not relevant for your application or never violated.
2. Enforce in the application. A good choice if your applications already enforce temporal RI. Otherwise, it can lead to more application complexity and places a burden on application developers.
3. Enforce using triggers. Create triggers for insert, update, and delete operations to enforce temporal RI.
4. Enforce using stored procedures. Create stored procedures that enforce temporal RI for many rows in bulk operations.

In the remainder of this article, we provide some examples for options 3 and 4, triggers and stored procedures, to enforce period containment between a child row and one or multiple parent rows. The code samples have been tested on DB2 10 for Linux, UNIX, and Windows®, but the same general concepts also apply to DB2 for z/OS®.

Enforcing temporal RI with triggers

To enforce temporal RI, you can create triggers for some or all of the following events:
• **Insertion of rows in the child table.**
  - Temporal RI might be violated if the period of a new product promotion is not contained in a business period for that product. A trigger can detect this condition and reject the insert of a promotion.

• **Update of rows in the child table**
  - Temporal RI might be violated if the primary key or the business time period of a child row is changed. The trigger can reject the update if needed.

• **Deletion of rows from the parent table.**
  - Temporal RI might be violated if a product is deleted while a related promotion still exists. In this case, the trigger can implement a delete rule of your choice, such as:
    - Rejecting and roll back the delete operation.
    - Cascading the delete and remove associated rows in the child table.
    - Modifying the child rows in some other application-defined manner.

• **Update of rows in the parent table.**
  - Temporal RI might be violated if the primary key or the business time period of a parent row is changed. In this case, the trigger can either reject the update or it can delete or modify the related rows in the child table.

The following sections provide sample triggers for inserts and updates of child table rows.

**Basic temporal RI triggers (no gaps)**

In this section we assume that the periods for any given product in the parent table (product_avail) never have gaps. This assumption implies that the temporal RI violation of promotion 19 in Figure 4 cannot occur and so the trigger does not need to check for this special case. This allows for a simple trigger implementation. Later we discuss how this trigger can be extended to handle gaps.

Listing 3 shows a trigger that is executed whenever a new row is inserted into the promotion table. The trigger checks temporal RI for the new row, which is to say that it verifies that the period of the new promotion row is contained in the period(s) of one or multiple parent rows for the same product ID.

The trigger performs two checks. First, it checks whether the start date of the inserted promotion is contained in the business period of a product_avail row with a matching prodID value. Then it checks a similar condition for the end date of the inserted promotion. If either of these tests fails (for example, if the count of matching parent rows is zero), then temporal RI is violated and an error is raised. If both the start and end date are contained in periods of corresponding parent rows, temporal RI is maintained and the insert completes. Otherwise, the trigger rejects the insert, signals a custom SQL state, and issues a message that temporal RI is violated.

**Listing 3. Trigger to enforce temporal RI upon insert of a child row**

```sql
CREATE TRIGGER promotion_insert_RI
BEFORE INSERT ON promotion
REFERENCING NEW AS new FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  IF (  -- Is promotion.promo_start part of an existing product_avail period?
```
A similar trigger is necessary if we expect that the columns `prodID`, `promo_start`, or `promo_end` of a promotion might be updated. For example, an update of the `promo_start` or `promo_end` values can move or increase the promotion period and potentially violate temporal RI. The logic of the trigger is the same as in Listing 4, except that the trigger fires only if any of these three columns is affected by an update statement.

**Listing 4. Trigger to enforce temporal RI upon update of a child row**

```sql
CREATE TRIGGER promotion_update_RI
BEFORE UPDATE OF prodID, promo_start, promo_end ON promotion
REFERENCING NEW AS new FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
  -- same trigger body as in Listing 3
END IF;
END@```

Next, let's discuss how these triggers can be enhanced to correctly deal with gaps in parent periods.

**Temporal RI triggers that check for gaps**

If the parent rows for a given key value can have gaps between their business periods, as in our product availability example, then the temporal RI verification in the trigger needs to be more sophisticated. The trigger needs to be enhanced to also detect "gap violations" such as promotion 19 in Figure 4. Listing 5 and Listing 6 show two alternative ways of coding a trigger that verifies period containment with the necessary gap check.

The trigger in Listing 5 selects parent rows from the product_avail table whose periods overlap with the period of the new row being inserted into the promotion table. For each of these parent rows, a left-outer self-join on the product_avail table retrieves any "next" parent rows whose period starts where the previous ends (`current.avail_end = next.avail_start`) (i.e., no gap). Result rows of this left outer join that have a NULL value in the `next.avail_start` column represent a gap in the periods of the parent rows. The other conditions in the trigger ensure that the start and end dates of the new promotion are contained in existing parent periods.
If there is no parent row in the product_avail table whose prodID value matches the prodID of the newly inserted promotion, then the outer join produces an empty result set. In this case, the SUM(start_check)+SUM(end_and_gap_check) in the top-level SELECT clause produces NULL. This NULL value causes the COALESCE function to return -1 and lets the integrity test fail as necessary.

**Listing 5. Enforcing temporal RI upon insert of a child row, considering parent period gaps**

```sql
CREATE OR REPLACE TRIGGER promotion_insert_RI_withGaps
BEFORE INSERT ON promotion
REFERENCING NEW AS new FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
    IF ( SELECT COALESCE(SUM(start_check)+SUM(end_and_gap_check), -1) check
        FROM ( SELECT CASE WHEN new.promo_start < current.avail_start
            AND current.avail_start =
            (SELECT MIN(avail_start)
            FROM product_avail
            WHERE prodID = new.prodID
            AND avail_start < new.promo_end
            AND avail_end > new.promo_start)
            THEN -1 -- PROMOTION.PROMO_START is out of product_avail range!
        ELSE 0 -- PROMOTION.PROMO_START is in range of current or previous
            -- product_avail period
        END start_check,
        CASE WHEN current.avail_end < new.promo_end
            THEN -1 -- PROMOTION.PROMO_END is out of product_avail range,
            -- or product_avail contains gaps!
        ELSE 0 -- PROMOTION.PROMO_END is out of current
            -- product_avail period exists
        END ELSE 0 -- PROMOTION.PROMO_END is in a product_avail period
        END end_and_gap_check
        FROM product_avail AS current
        LEFT OUTER JOIN product_avail AS next
        ON current.avail_end = next.avail_start
        AND current.prodID = next.prodID
        WHERE current.prodID = new.prodID
        AND current.avail_start < new.promo_end
        AND current.avail_end > new.promo_start ) )<0 THEN SIGNAL SQLSTATE 'RI999'
        SET MESSAGE_TEXT='PROMOTION PERIOD VIOLATES TEMPORAL RI!';
    END IF;
END @
```

The best approach to understanding the logic in this trigger is to draw pictures such as Figure 4 and walk through the conditions and CASE expressions.

**Listing 5** is just one possible solution, and you might find other ways in SQL to code the same conditions. One alternative is shown in **Listing 6**. The trigger body in **Listing 6** checks three conditions. The first SELECT statement verifies that the start date of the inserted promotion lies within the business period of a product row with the same key value. The second SELECT statement verifies that the end date of the new promotion lies within the period of a matching product row. And the third SELECT checks for gaps between periods of the relevant parent rows in the product_avail table.
Listing 6. Enforcing temporal RI upon insert of a child row, considering parent period gaps

CREATE OR REPLACE TRIGGER promotion_insert_RI_withGaps
BEFORE INSERT ON promotion
REFERENCING NEW AS new
FOR EACH ROW MODE DB2SQL
BEGIN ATOMIC
   IF (-- verify that a product row exists with prodID = promotion.prodID
      -- and whose period contains promo_start
      (SELECT COUNT(*)
       FROM product_avail prod
       WHERE new.prodID = prod.prodID
       AND prod.avail_start <= new.promo_start
       AND new.promo_start < prod.avail_end)   < 0 )
   OR (-- verify that a product row exists with prodID = promotion.prodID
      -- and whose period contains promo_end
      (SELECT COUNT(*)
       FROM product_avail prod
       WHERE new.prodID = prod.prodID
       AND prod.avail_start < new.promo_end
       AND new.promo_end <= prod.avail_end)   < 0 )
   OR (-- check for any gaps between the relevant product rows that
      -- have prodID = promotion.prodID
      (SELECT COUNT(*)
       FROM product_avail prod
       WHERE new.prodID = prod.prodID
       AND new.promo_start < prod.avail_end
       AND prod.avail_end < new.promo_end
       AND NOT EXISTS (SELECT *
                        FROM product_avail prod2
                        WHERE prod2.prodID = prod.prodID
                        AND prod2.avail_start <= prod.avail_end
                        AND prod.avail_end < prod2.avail_end) )   > 0 )
   THEN SIGNAL SQLSTATE 'RI999'
      SET MESSAGE_TEXT='PROMOTION PERIOD VIOLATES TEMPORAL RI!';
   END IF;
END @

Enforcing temporal RI with stored procedures

If you are adding, changing, or deleting many rows in a table, it can be more efficient to check the temporal RI with set-based operations in a stored procedure rather than with a trigger that is activated once for every affected row. For example, if you perform load or import operations, or bulk updates or deletes, you might want to use a stored procedure to verify temporal RI.

For example, the procedure in Listing 7 verifies temporal RI between the product_avail table and the promotion table. It assumes that the periods for a given product in the product_avail table contain no gaps. The core RI logic in this procedure is analogous to the logic in the trigger in Listing 3.

When creating such a procedure, you can choose how you want to handle temporal RI. The violation handling in Listing 7 follows a simple approach: If there is at least one promotion row that violates temporal RI, the procedure signals a custom SQL state and issues a message that temporal RI is violated.

Listing 7. Stored procedure to check temporal RI in bulk

CREATE PROCEDURE check_temporal_integrity()
BEGIN
    IF (SELECT COUNT(*) FROM (  
        SELECT prodID, promo_start, promo_end,  
        -- check whether a product row contains promotion.promo_start  
        (SELECT COUNT(*) FROM product_avail p  
            WHERE p.ProdID = promotion.ProdID  
            AND p.avail_start <= promotion.promo_start  
            AND p.avail_end > promotion.promo_start) c1,  
        -- check whether a product_avail row contains promotion.promo_end  
        (SELECT COUNT(*) FROM product_avail p  
            WHERE p.ProdID = promotion.ProdID  
            AND p.avail_start < promotion.promo_end  
            AND p.avail_end >= promotion.promo_end) c2  
        FROM promotion)
    ) > 0
    THEN SIGNAL SQLSTATE 'RI999'
        SET MESSAGE_TEXT='AT LEAST ONE PROMOTION VIOLATES TEMPORAL RI!';
    ELSE SIGNAL SQLSTATE 'RI000'
        SET MESSAGE_TEXT='TEMPORAL RI HAS BEEN SUCCESSFULLY VERIFIED!';
    END IF;
END@;

Instead of just counting the offending rows, you can also return their primary key values, insert them into an exception table, or perform any custom action that meets your application requirements. For example, the procedure in Listing 8 copies the offending promotion rows into a separate table and deletes them from the promotion table.

Listing 8. Stored procedure to copy and delete rows that violate temporal RI

CREATE PROCEDURE check_temporal_integrity()
BEGIN
    INSERT INTO promo_exception
        SELECT promoID, prodID, price, promo_start, promo_end
    FROM (  
        SELECT promoID, prodID, price, promo_start, promo_end,  
        -- check whether a product row contains promotion.promo_start  
        (SELECT COUNT(*) FROM product_avail p  
            WHERE p.ProdID = promotion.ProdID  
            AND p.avail_start <= promotion.promo_start  
            AND p.avail_end > promotion.promo_start) c1,  
        -- check whether a product_avail row contains promotion.promo_end  
        (SELECT COUNT(*) FROM product_avail p  
            WHERE p.ProdID = promotion.ProdID  
            AND p.avail_start < promotion.promo_end  
            AND p.avail_end >= promotion.promo_end) c2  
        FROM promotion)
    ) S  
    -- select all promotion rows, for which either the first  
    -- or the second check fails  
    WHERE c1 = 0 OR c2 = 0;
    DELETE FROM promotion
    WHERE promoID IN (SELECT promoID FROM promo_exception);
END@
Summary

This article has explained the notion of temporal RI and described how triggers and stored procedures can be used to enforce it. Temporal RI between two application-period temporal tables consists of two conditions. First, for every row in the child table, there is one or more matching rows in the parent table with the same key value. Second, the business time period of any child row must satisfy a temporal relationship with the business time period of the related parent rows. For example, you may want to enforce that the child periods are identical to the parent periods, or that they are contained within one or multiple parent periods. Different application scenarios may require different temporal relationships to be enforced.

Enforcing temporal RI at the application level is not trivial. It can be accomplished at the database level with triggers or stored procedures. The samples in this article can serve as a starting point for you when writing your own triggers or procedures for temporal integrity checking.

© Copyright IBM Corporation 2012
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)