Adopting temporal tables in DB2, Part 1: Basic migration scenarios for system-period tables

Jan-Eike Michels
Matthias Nicola

October 18, 2012

The temporal features in IBM® DB2® 10 provide rich functionality for time-based data management. Temporal tables in DB2 can record the complete history of their data changes so you can go back in time and query any past data state. Temporal tables also enable you to track and manage the business validity of your data, indicating when the information is valid in the real world. This "Adopting temporal tables in DB2" series explains how to migrate existing tables and temporal solutions to temporal tables in DB2. Part 1 describes basic scenarios for adopting system-period temporal tables in DB2.

View more content in this series

Introduction to temporal data management with DB2

DB2 supports time-based data management that allows you to insert, update, delete, and query data in the past, the present, and future while keeping a complete history of what you knew and when you knew it.

Temporal tables in DB2

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, which is also known as transaction time.

- **Application-period temporal tables**— Where applications supply dates or timestamps to describe the business validity of their 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 their data while DB2 keeps a full history of any updates and deletes.

This article series assumes you are familiar with temporal tables in DB2. The article "A Matter of Time: Temporal Data Management in DB2" provides an introduction to these topics. "DB2 best practices: Temporal data management with DB2" provides additional usage guidelines. For example, range partitioning for temporal tables, privileges for history data, history conscious schema design, and other recommendations are included in the best practices piece.

**System-period temporal tables — the basics**

When you create a table with a system time period, you're instructing DB2 to automatically capture changes made to the table and to save "old" rows in a history table—a separate table with the same structure as your base table (also called current table).

Defining a new system-period temporal table from scratch involves the following steps:

1. **Create the base table for the current data**

   **Listing 1. Definition of the base table**

   ```sql
   CREATE TABLE employees (  
   empid        BIGINT NOT NULL PRIMARY KEY,  
   name         VARCHAR(20),  
   deptid       INTEGER,  
   salary       DECIMAL(7,2),  
   system_begin TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW BEGIN,  
   system_end   TIMESTAMP(12) NOT NULL GENERATED ALWAYS AS ROW END,  
   trans_start  TIMESTAMP(12) GENERATED ALWAYS AS TRANSACTION START ID,  
   PERIOD SYSTEM_TIME (system_begin, system_end) );
   ```

2. **Create the history table with the same structure as the base table**

   **Listing 2. Definition of the history table**

   ```sql
   CREATE TABLE employees_hist (  
   empid        BIGINT NOT NULL,  
   name         VARCHAR(20),  
   deptid       INTEGER,  
   salary       DECIMAL(7,2),  
   system_begin TIMESTAMP(12) NOT NULL,  
   system_end   TIMESTAMP(12) NOT NULL,  
   trans_start  TIMESTAMP(12) );
   ```

   Alternatively, the history can also be created with the LIKE clause in the CREATE TABLE statement, which ensures that the columns are the same as in the base table:

   ```sql
   CREATE TABLE employees_hist LIKE employees;
   ```

3. **Enable versioning and specify which history table to use**

   **Listing 3. Enabling versioning**

   ```sql
   ALTER TABLE employees  
   ADD VERSIONING USE HISTORY TABLE employees_hist;
   ```
Once this link is established, history rows are automatically created and time-travel queries are supported.

**Querying a system-period temporal table**

You can run regular SQL queries on a system-period temporal table as on any other table. For example, the following query returns the current data for employee 1000:

```sql
SELECT empid, name, deptid, salary, system_begin, system_end
FROM employees
WHERE empid = 1000 ;
```

Additionally, a system-period temporal table can be queried with the new `FOR SYSTEM_TIME` clause to retrieve past states of your data. For example, the next query returns the record of employee 500 as it was in the database on 1 Feb 2011 (at midnight):

```sql
SELECT empid, name, deptid, salary, system_begin, system_end
FROM employees FOR SYSTEM_TIME AS OF '2011-02-01'
WHERE empid = 500 ;
```

The row returned by this query can be a current row or a historical row. DB2 transparently examines the current table and the history table, and returns the correct result. The value in the `FOR SYSTEM_TIME AS OF` clause can be a date, timestamp, expression, parameter marker, or a host variable.

**NOTE:** In DB2 for z/OS®, the literal value 2011-02-01 must be written as `TIMESTAMP '2011-02-01'` so that the value is correctly cast to the target data type.

To retrieve all data (current and history rows) for employee 4711, use this SQL:

```sql
SELECT empid, name, deptid, salary, system_begin, system_end
FROM employees FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30'
WHERE empid = 4711 ;
```

Sometimes you might wish to retrieve the previous version of a row, which is the latest version before the current version. The following query returns the previous version of the row for employee 1212:

```sql
SELECT prev.*
FROM employees curr,
     employees FOR SYSTEM_TIME BETWEEN '0001-01-01' AND CURRENT_TIMESTAMP prev
WHERE curr.empid = 1212
AND prev.empid = 1212
AND curr.system_begin = prev.system_end;
```

Alternatively, the same result can be obtained by querying the history table directly:
Typical migration scenarios

Adopting system-period temporal tables in DB2 is easy. The exacts steps to migrate an existing solution to system-period temporal tables depend on the characteristics of your existing tables and data, such as:

- Whether you are already recording history rows, for example with triggers
- Whether you use a single table to hold current and historical data or two separate tables
- Whether you store one or two timestamp values for each version of a row
- The period model (inclusive-exclusive vs. inclusive-inclusive) you have chosen for your existing temporal solution

Based on these properties, Part 1 and Part 2 of this series examine five flavors of existing solutions and describe their migration to system-period temporal tables in DB2.

Table 1. Overview of migration scenarios discussed in part 1 and 2 of this article series

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Existing solution records history?</th>
<th>Existing history is in a separate table?</th>
<th>No. of timestamp columns used for versioning?</th>
<th>Your period model?</th>
<th>Article</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario 0</td>
<td>No</td>
<td>N/A</td>
<td>None</td>
<td>None</td>
<td>Part 1</td>
</tr>
<tr>
<td>Scenario 1</td>
<td>Yes</td>
<td>Yes</td>
<td>Two</td>
<td>Inclusive-exclusive</td>
<td>Part 1</td>
</tr>
<tr>
<td>Scenario 2</td>
<td>Yes</td>
<td>No</td>
<td>Two</td>
<td>Inclusive-exclusive</td>
<td>Part 2</td>
</tr>
<tr>
<td>Scenario 3</td>
<td>Yes</td>
<td>No</td>
<td>One</td>
<td>Inclusive-exclusive</td>
<td>Part 2</td>
</tr>
<tr>
<td>Scenario 4</td>
<td>Yes</td>
<td>No</td>
<td>Two</td>
<td>Inclusive-inclusive</td>
<td>Part 2</td>
</tr>
</tbody>
</table>

Scenario 0 is the simplest and covered first.

For scenarios 1-4, here are the general steps for migration to a system-period temporal table in DB2:

1. Ensure that no table access (read or write) takes place during the migration.
2. Disable any custom triggers or application code responsible for creating history rows on update/delete.
3. Align the table schema: Create a history table and any additional timestamp columns, if necessary. Ensure that the current table and the history table have the same columns with the same names, order, nullability, and data types. Tip: Use the CREATE TABLE statement with a LIKE clause, as shown earlier.
4. Change the data type of existing timestamp columns to TIMESTAMP(12).
5. Move existing history rows into a history table, if your existing solution uses a single table for current and history data.

6. Adjust applications: INSERT/UPDATE/DELETE statements typically require minimal or no changes. Depending on the existing period and the number of timestamp columns, minor changes to existing queries might be necessary. Additional changes are recommended for ease of use and better performance.

**Scenario 0 — Enable versioning for a non-temporal table**

This scenario describes how to start recording history and enable time-travel queries for a regular table that does not yet have any existing history associated with it. Let's take the table employees_s0 in **Listing 4** as an example.

**Listing 4. Existing table definition**

```sql
CREATE TABLE employees_s0 (  
  empid        BIGINT NOT NULL PRIMARY KEY,  
  name         VARCHAR(20),  
  deptid       INTEGER,  
  salary       DECIMAL(7,2) );
```

You can easily turn this table into a system-period temporal table using the three statements in **Listing 5**. The `ALTER TABLE` statement adds the three mandatory timestamp columns and declares the `SYSTEM_TIME` period. The subsequent statements create the corresponding history table and enable versioning, respectively. After these steps, reorganization of the table is not necessary.

The new timestamp columns are defined as **IMPLICITLY HIDDEN**, which is optional and ensures that they do not appear in the result set of `SELECT *` queries. Hence, existing applications will see exactly the same query results as before the migration. No changes to existing queries or insert, update, and delete statements are required.

**Listing 5. Converting the table employees_s0 into an STT**

```sql
ALTER TABLE employees_s0  
ADD COLUMN sys_start TIMESTAMP(12) NOT NULL GENERATED AS ROW BEGIN IMPLICITLY HIDDEN  
ADD COLUMN sys_end TIMESTAMP(12) NOT NULL GENERATED AS ROW END IMPLICITLY HIDDEN  
ADD COLUMN trans_id TIMESTAMP(12) GENERATED AS TRANSACTION START ID IMPLICITLY HIDDEN  
ADD PERIOD SYSTEM_TIME (sys_start, sys_end);
CREATE TABLE employees_s0_hist LIKE employees_s0;  
ALTER TABLE employees_s0 ADD VERSIONING USE HISTORY TABLE employees_s0_hist;
```

When you add timestamp columns to an existing table, as in **Listing 5**, all existing rows get the value 9999-12-30 in the `sys_end` column, indicating that all rows are current rows. However, DB2 does not know when these rows were originally inserted and what their `sys_start` values should be. Hence, all existing rows initially get the `sys_start` value 0001-01-01, which is January 1 in the year 0001.

If you prefer to use the current time as the system start time for all existing rows, you have two options to achieve this:
Add the sys_start column with the desired default value first, and then issue a second ALTER TABLE statement to make the column GENERATED AS ROW BEGIN. See Listing 6.

You use the steps in Listing 5 but export and reload all rows before you enable versioning. This is shown in Listing 7.

**Listing 6. Converting the table employees_s0 into an STT, with custom value for sys_start**

```sql
ALTER TABLE employees_s0
ADD COLUMN sys_start TIMESTAMP(12) NOT NULL DEFAULT CURRENT_TIMESTAMP IMPLICITLY HIDDEN
ADD COLUMN sys_end   TIMESTAMP(12) NOT NULL GENERATED AS ROW END IMPLICITLY HIDDEN
ADD COLUMN trans_id  TIMESTAMP(12) GENERATED AS TRANSACTION START ID IMPLICITLY HIDDEN;

ALTER TABLE employees_s0
ALTER COLUMN sys_start DROP DEFAULT SET GENERATED AS ROW BEGIN
ADD PERIOD SYSTEM_TIME (sys_start, sys_end);

CREATE TABLE employees_s0_hist LIKE employees_s0;

ALTER TABLE employees_s0 ADD VERSIONING USE HISTORY TABLE employees_s0_hist;
```

**Listing 7. Loading data with the PERIODIGNORE modifier**

```sql
EXPORT TO emp.del OF DEL MODIFIED BY IMPLICITLYHIDDENINCLUDE
   SELECT * FROM employees_s0;

LOAD FROM emp.del OF DEL MODIFIED BY PERIODIGNORE IMPLICITLYHIDDENINCLUDE
   REPLACE INTO employees_s0;

ALTER TABLE employees_s0 ADD VERSIONING USE HISTORY TABLE employees_s0_hist;
```

The modifier PERIODIGNORE in the LOAD command instructs DB2 to ignore the timestamps in the exported data and instead generate new timestamps during load. In other situations, you might find it helpful to use the modifier PERIODOVERRIDE, which loads existing timestamps into the system time columns instead of generating new timestamps during the load operation.

**Scenario 1 — Migrate two tables for current and historical data**

In this scenario, we look at an existing temporal solution that has similar properties as system-period temporal tables in DB2, including the following:

- There are two separate tables, one for the current data and one for history data.
- The period model is inclusive-exclusive.
- Two timestamp columns are used for the period.

**Existing table definitions**

Let's assume the existing temporal solution uses the following tables to capture information about employees and the history of changes.
Listing 8. Existing table definitions

```sql
CREATE TABLE employees_s1 (  
    empid        BIGINT NOT NULL PRIMARY KEY,  
    name         VARCHAR(20),  
    deptid       INTEGER,  
    salary       DECIMAL(7,2),  
    system_begin TIMESTAMP(6) NOT NULL DEFAULT CURRENT TIMESTAMP,  
    system_end   TIMESTAMP(6) NOT NULL DEFAULT TIMESTAMP '3000-01-01 00:00:00.000000' );

CREATE TABLE employees_s1_hist (  
    empid        BIGINT NOT NULL,  
    name         VARCHAR(20),  
    deptid       INTEGER,  
    salary       DECIMAL(7,2),  
    system_begin TIMESTAMP(6) NOT NULL,  
    system_end   TIMESTAMP(6) NOT NULL );
```

The table `employees_s1` holds current information about employees. Each row in this table has the `system_end` value `3000-01-01 00:00:00.000000` to indicate that the information is current until changed.

We assume that appropriate `AFTER DELETE` and `AFTER UPDATE` triggers are defined on the table `employees_s1` to insert the before images of updated and deleted rows into the table `employees_s1_hist`.

Table 2 and Table 3 show sample content of these two database tables.

Table 2. Data in employees_s1

<table>
<thead>
<tr>
<th>empid</th>
<th>name</th>
<th>deptid</th>
<th>salary</th>
<th>system_begin</th>
<th>system_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>John</td>
<td>1</td>
<td>5000.00</td>
<td>2010-05-11 12:00:00.000000</td>
<td>3000-01-01 00:00:00.000000</td>
</tr>
<tr>
<td>1212</td>
<td>James</td>
<td>2</td>
<td>4500.00</td>
<td>2011-05-11 09:30:00.100000</td>
<td>3000-01-01 00:00:00.000000</td>
</tr>
<tr>
<td>4711</td>
<td>Maddy</td>
<td>1</td>
<td>5250.00</td>
<td>2011-07-30 09:25:47.123456</td>
<td>3000-01-01 00:00:00.000000</td>
</tr>
</tbody>
</table>

Table 3. Data in employees_s1_hist

<table>
<thead>
<tr>
<th>empid</th>
<th>name</th>
<th>deptid</th>
<th>salary</th>
<th>system_begin</th>
<th>system_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>Peter</td>
<td>1</td>
<td>4000.00</td>
<td>2010-05-11 12:00:00.000000</td>
<td>2011-06-30 09:15:45.123456</td>
</tr>
<tr>
<td>1212</td>
<td>James</td>
<td>1</td>
<td>4000.00</td>
<td>2010-05-11 12:00:00.000000</td>
<td>2011-05-11 09:30:00.100000</td>
</tr>
<tr>
<td>4711</td>
<td>Maddy</td>
<td>1</td>
<td>4000.00</td>
<td>2010-05-11 12:00:00.000000</td>
<td>2011-07-30 09:25:47.123456</td>
</tr>
</tbody>
</table>

Drop any triggers that create history rows

Since DB2 generates history rows for updated or deleted records automatically, you should remove any triggers that generate history rows in your existing solution. Drop the triggers at the beginning of the migration process to avoid the unnecessary generation of (possibly) incorrect history rows, in case any rows need to be updated as part of the migration process itself.
Migrate the table definition and data

The existing table definitions that we assume in this scenario (Listing 8) differ from a system-period temporal table (Listing 1) in the following properties:

1. The existing system_end value for current rows is 3000-01-01, but should be 9999-12-30 in a system-period temporal table.
2. The data type of the system_begin and system_end columns is \texttt{TIMESTAMP(6)} , but it must be \texttt{TIMESTAMP(12)} for a system-period temporal table.
3. System-period temporal tables must have a transaction ID column in the base table and the history table.
4. The generation definitions of the system_begin and system_end columns are different:
   \texttt{DEFAULT CURRENT TIMESTAMP/TIMESTAMP \ldots} \texttt{VS. GENERATED ALWAYS AS ROW BEGIN/END.}
5. A system-period temporal table requires a \texttt{PERIOD SYSTEM TIME} specification.
6. For a system-period temporal table, versioning must be enabled.

The statements in Listing 9 address these differences and convert the table employees_s1 into a system-period temporal table.

**Listing 9. Converting the table employees_s1 into an STT**

```sql
-- 1. Change the system_end values to the same value that DB2 generates
UPDATE employees_s1 SET system_end = '9999-12-30';

-- 2.+3. Change data types to \texttt{TIMESTAMP(12)} and add the transID column
ALTER TABLE employees_s1
    ALTER COLUMN system_begin SET DATA TYPE \texttt{TIMESTAMP(12)}
    ALTER COLUMN system_end   SET DATA TYPE \texttt{TIMESTAMP(12)}
    ADD   COLUMN trans_start \texttt{TIMESTAMP(12)} GENERATED ALWAYS
           AS TRANSACTION START ID IMPLICITLY HIDDEN;

ALTER TABLE employees_s1_hist
    ALTER COLUMN system_begin SET DATA TYPE \texttt{TIMESTAMP(12)}
    ALTER COLUMN system_end   SET DATA TYPE \texttt{TIMESTAMP(12)}
    ADD COLUMN trans_start \texttt{TIMESTAMP(12)} IMPLICITLY HIDDEN;

-- 4.+5. Set the auto generation of the columns system_begin and system_end, and declare these columns as a system time period
ALTER TABLE employees_s1
    ALTER COLUMN system_begin DROP DEFAULT SET GENERATED ALWAYS AS ROW BEGIN
    ALTER COLUMN system_end   DROP DEFAULT SET GENERATED ALWAYS AS ROW END
    ADD PERIOD \texttt{SYSTEM TIME (system_begin, system_end)};

-- 6. Reorg the tables and enable versioning
REORG TABLE employees_s1;
REORG TABLE employees_s1_hist;

ALTER TABLE employees_s1
    ADD VERSIONING USE HISTORY TABLE employees_s1_hist;
```

Let’s discuss each of these steps in more detail:

1. Updating the existing system_end values from 3000-01-01 to the same value that DB2 generates (9999-12-30) is highly recommended but not strictly necessary for the migration process. To identify current rows by a single common system_end value after the migration, that value must be 9999-12-30.
Updating many rows in a single statement might require a lot of log space. To avoid a log-full condition, make sure your log is large enough or update the rows in a series of smaller batches with intermediate commits.

Note that DB2 does not use the value 9999-12-31 as the system_end value for current rows. The reason is that the value 9999-12-31 might change to a value in the year 10000 if an application converts it to a different time zone. This is undesirable because a date with five-digit year cannot be inserted or loaded in DB2 again.

2. When you increase the precision of the system_begin and system_end columns from `TIMESTAMP(6)` to `TIMESTAMP(12)`, existing values in these columns are automatically cast and padded with six additional zeros. For example, the value 2010-05-11 12:00:00.000000 becomes 2010-05-11 12:00:00.000000000000.

3. Because the new column trans_start is defined as nullable, all existing rows have the NULL value in this column. For new rows, DB2 generates a value for this column automatically, if needed. The trans_start column is defined as `IMPLICITLY HIDDEN` so it does not show up in the result set of "SELECT *" queries. But it can still be retrieved or compared if you use its column name explicitly in the `SELECT` or `WHERE` clause.

4. Once the system_begin and system_end columns have been changed to `GENERATED ALWAYS AS ROW BEGIN/END`, users cannot provide values for these columns in `INSERT` or `UPDATE` statements. Instead, DB2 always generates a timestamp value for the transaction in which the insert or update took place.

5. Adding the `PERIOD SYSTEM_TIME` declaration will fail if the columns involved do not have the required properties.

6. Explicit activation of versioning is required so the table becomes an STT, history is automatically recorded, and temporal queries are supported. Enabling versioning will fail if the columns of the history table don’t match the base table. A REORG of both tables is required before any `INSERT`, `UPDATE`, or `DELETE` statements can be executed. You might also want to update statistics at this time by issuing the `RUNSTATS` command on the base and on the history table separately.

**Application changes**

Existing applications that read or write to your tables may or may not require minor changes, depending on how exactly they access the tables. For example, a Java™ application is not affected by the data type change from `TIMESTAMP(6)` to `TIMESTAMP(12)`.

**INSERT, UPDATE, OR DELETE statements**

In this migration scenario, most if not all `INSERT`, `UPDATE`, or `DELETE` statements continue to work unmodified. The reason is that in the existing solution (Listing 8) the values for the system_begin and system_end columns were automatically supplied by default values, so applications did not have to provide values for these columns explicitly. After the migration to a system-period temporal table, DB2 continues to generate values for these columns automatically. Also, in the existing solution history rows were created by a database trigger, which got replaced by DB2’s automatic generation of history rows. Again, no changes on the application side are required.
If an application contains `INSERT` or `UPDATE` statements that write to the history table or provide values for the `system_begin` and `system_end` columns in the current table, those statements need to be changed. The reason is that DB2 performs these writes automatically for you.

### Queries

Any application that explicitly tests for the previous `system_end` value, either in application code or in an SQL `WHERE` clause, such as `WHERE system_end = '3000-01-01 00:00:00.000000'`, should be changed to test for the DB2 generated `system_end` value 9999-12-30 instead.

Many common temporal queries will still work unchanged, but can be greatly simplified. For example, assume you want to retrieve the information recorded for employee 500 as of midnight on 1 Feb 2011. **Listing 10** shows what such a query would have looked like before the migration as well as a simplified version of the same query you can use after the migration.

**Listing 10. Retrieving employee 500 as of 1 Feb 2011**

```sql
-- Before the migration:
SELECT empid, name, deptid, salary, system_begin, system_end
FROM employees_s1
WHERE system_begin <= '2011-02-01'
  AND system_end > '2011-02-01'
  AND empid = 500
UNION ALL
SELECT empid, name, deptid, salary, system_begin, system_end
FROM employees_s1_hist
WHERE system_begin <= '2011-02-01'
  AND system_end > '2011-02-01'
  AND empid = 500;
-- Simplified query after the migration:
SELECT empid, name, deptid, salary, system_begin, system_end
FROM employees_s1 FOR SYSTEM_TIME AS OF '2011-02-01'
WHERE empid = 500;
```

Although the original query still works on the migrated tables (if you have the read privilege for the history table), the simplified query is a lot shorter, easier to understand, and thus less error-prone.

Similarly, imagine you want to see all current and history rows for employee 4711. **Listing 11** illustrates how you can code such queries before and after the migration.

**Listing 11. Retrieving all current and history rows for employee 4711**

```sql
-- Before the migration:
SELECT empid, name, deptid, salary, system_begin, system_end
FROM employees_s1
WHERE empid = 4711
UNION ALL
SELECT empid, name, deptid, salary, system_begin, system_end
FROM employees_s1_hist
WHERE empid = 4711;
-- Simplified query after the migration:
SELECT empid, name, deptid, salary, system_begin, system_end
FROM employees_s1 FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-30'
WHERE empid = 4711;
```
Variations and additional considerations

In this section, we discuss several variations of migration scenario 1.

Null value in the ROW_END column to indicate "until the end of time"

In this migration scenario, we have assumed that the existing data uses 3000-01-01 as the system_end value for current rows to indicate "until the end of time." What if your application has used the NULL value instead? In that case, you must replace all system_end values for current rows with the value 9999-12-30. Additionally, any queries that previously tested for the NULL value in the system_end column should be changed.

For example, the predicate `system_end IS NULL` should be changed to `system_end = '9999-12-30'`. Similarly, a search condition in your existing solution such as `system_end > '2012-02-01' OR system_end IS NULL` can be simplified to `system_end > '2012-02-01'`.

Null values in the ROW BEGIN column

In a system-period temporal table, the system_begin column cannot be NULL. If your existing system_begin column contains NULL values, maybe to indicate that rows existed since some unknown point in time, you must replace these NULL values with a non-null value such as 0001-01-01 or 1900-01-01.

ROW BEGIN value is not less than ROW END value

In each row of a system-period temporal table, the system_begin value must be less than the system_end value. If this is not true for your existing data, queries might return unexpected results. If you are not sure, consider defining the constraint `CHECK(system_begin < system_end)` on your existing tables employees_s1 and employees_s1_hist before the migration. If there are any rows that violate this constraint, you must correct or delete them.

You can drop the constraint once all rows satisfy the constraint and the migration is completed. When versioning is enabled for a system-period temporal table, DB2 automatically ensures that system_begin is less than system_end, so the constraint would only be unnecessary overhead.

History table contains more or fewer columns than the base table

In your existing temporal solution, the history table might have a different number of columns than the base table. However, the number of columns as well as their names, position, data type, and nullability must be the same in both tables before you can convert them into a system-period temporal table. So you must add or remove columns so that both tables have the same schema.

If you want to record history only for a subset of the columns in the base table, consider splitting the base table vertically into two tables, as discussed in "DB2 best practices: Temporal data management with DB2."

Need to record the user ID or application ID for each row change

A system-period temporal table does not automatically record which user ID or application ID has cause a particular row change. To record this information, you need to have explicit columns for it.
and logic to provide the proper values. For example, you could use a `BEFORE INSERT` trigger such as in **Listing 12** to record the user ID in the history table:

**Listing 12. Trigger to record user IDs in the history table**

```sql
CREATE TRIGGER pop_user_id_col
    NO CASCADE BEFORE INSERT ON employees_s1_hist
    REFERENCING NEW AS NROW
    FOR EACH ROW MODE DB2SQL
    BEGIN ATOMIC
        SET user_id_col = CURRENT CLIENT_USERID;
    END#
```

**History table also contains current rows**

If your existing solution stores current rows redundantly in the base table and the history table, you must delete the current rows from the history table. Otherwise, temporal queries might return incorrect results.

**History table contains rows with overlapping periods**

For a given key value, such as an `empid` value in our employee example, the history table must not contain two or more rows with the same key and overlapping periods. Overlapping periods for the same key value means that at some point in time, there were two current rows for the same primary key in the current table, which would be inconsistent. As a result, temporal queries might return unexpected results.

If you are not sure, you should verify that your existing history data does not contain overlaps. The query in **Listing 13** returns all overlaps, if any exist.

**Listing 13. A query that detects temporal overlaps**

```sql
SELECT empID,
    previous_end    AS overlap_start,
    system_begin    AS overlap_end
FROM
    (SELECT empID, system_begin, system_end,
        MIN(system_end) OVER (PARTITION BY empID ORDER BY system_begin
        ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)
        AS previous_end
        FROM employees_s1_hist)
WHERE system_begin < previous_end;
```

**Generation of history rows: One per transaction vs. one per statement**

One difference between versioning with a homegrown trigger-based solution and a DB2 system-period temporal table is that the trigger-based solution records history rows per statement whereas a system-period temporal table records history rows per transaction.

If the same row is updated multiple times in a single transaction, a trigger generates a history row for each of these updates. As a result, the history table will contain intermediate versions of the row that were never committed in the database. Consequently, any queries against the history
table can see uncommitted data from the base table. From a transactional point of view this means that any history queries are performing "dirty reads," as if the isolation level was uncommitted read (UR), which is clearly not desirable for most applications.

This problem is solved by the migration to system-period temporal tables in DB2. For each modified row, a system-period temporal table creates at most one history row per transaction, which records the state of the row before the current transaction. This behavior in DB2 saves storage space and ensures that the history data is transactionally correct.

**Summary**

The temporal capabilities in DB2 provide sophisticated support for time-aware data management, compliance, and auditing requirements. Part 1 of this series has described two common migration scenarios to adopt system-period temporal tables in DB2. As it turns out, migrating to temporal tables is really easy.

Scenario 0 is the most basic case where a regular (non-versioned) table is converted to a system-period temporal table with just three simple DDL statements. Scenario 1 assumes that an application has an existing pair of tables to record current and history data using timestamps and triggers. Again, only a handful of DDL statements is required to migrate them to temporal tables. Additional migration scenarios are discussed in subsequent parts of this series.
Related topics

- For a technical introduction to temporal tables in DB2, read "A Matter of Time: Temporal Data Management in DB2."
- Consult usage and performance guidelines: "DB2 best practices: Temporal data management with DB2."
- The benefits of temporal data management are discussed in "Improving data quality for exceptional business accuracy and compliance: Temporal data management with IBM DB2 Time Travel Query."
- Refer to the DB2 product documentation: Time Travel Query using temporal tables.
- Download a DB2 trial version or the free DB2 Express-C to try out the new temporal data management features yourself.
- Now you can use DB2 for free. Download DB2 Express-C, a no-charge version of DB2 Express Edition for the community that offers the same core data features as DB2 Express Edition and provides a solid base to build and deploy applications.