IBM® DB2® for Linux®, UNIX®, and Windows®

Best Practices

Temporal Data Management with DB2

Matthias Nicola
IBM Silicon Valley Lab
mnicola@us.ibm.com

Issued: August 2012

Issued: August 2012

Executive Summary

The temporal features in the IBM® DB2® for Linux®, UNIX®, and Windows® Version 10 product (hereafter referred to as "DB2") provide rich functionality for time-based data management. For example, you can choose to record the complete history of data changes for a database table so that you can "go back in time" and query any past state of your data. You can also indicate the business validity of data by assigning a pair of date or timestamp values to a row to indicate when the information is deemed valid in the real world. Using new and standardized SQL syntax, you can easily insert, update, delete, and query data in the past, present, or future.

The temporal features in the DB2 product enable you to accurately track information and data changes over time and provide an efficient and cost-effective way to address auditing and compliance requirements. This article describes a set of best practices that help ensure smooth operation and high performance for DB2 temporal data management.

Introduction to the temporal capabilities in DB2

The DB2 for Linux, UNIX, and Windows Version 10 software ("DB2") supports time-based data management that allows you to insert, update, delete, and query data in the past, the present, and the future while keeping a complete history of "what you knew" and "when you knew it".

DB2 supports three types of temporal tables:

- **System-period temporal tables (STTs).**
  For STTs, DB2 transparently keeps a history of updated and deleted rows over time. With new constructs in the SQL:2011 standard, you can "go back in time" and query the database as of any chosen point in the past. This is based on system timestamps that DB2 assigns internally to manage *system time*, also known as *transaction time*.
- **Application-period temporal tables (ATTs).**
  Applications supply dates or timestamps to describe the business validity of their data in ATTs. New SQL constructs allow users to insert, query, update, and delete data in the past, present, or future. DB2 automatically applies temporal constraints and "row-splits" to correctly maintain the application-supplied *business time*, also known as *valid time*.
- **Bitemporal tables (BTTs).**
  BTTs manage both *system time* and *business time*, and 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. Every BTT is also an STT and an ATT.

For the remainder of this article we assume that you are familiar with the basics of system-period temporal tables, application-period temporal tables, and bitemporal tables in DB2. You should know how to create, query, and update such tables, and understand how DB2 might 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 a complete introduction to these topics.

Summary of best practices

The following list summarizes the best practices for temporal data management. The subsequent sections explain these guidelines in more detail.

- Modeling data and applications
  - Use system-time to track when data was changed inside the DB2 system.
  - Use business-time to track when data was, is, or will be valid in the real-world.
  - Use bitemporal tables to combine system and business time as needed.
  - Use inclusive-exclusive periods rather than inclusive-inclusive periods in your application logic.
  - Don't use real-world dates or timestamps to define business time periods.

- Administering database objects
  - Query the catalog view SYSCAT.PERIODS to list temporal tables together with their periods and history tables.
  - Prevent unintended loss of history tables with the RESTRICT ON DROP option.
  - Know how to drop columns from tables with system time.
  - Define access privileges on historical tables to prevent unintended data changes.
  - Consider using the APPEND ON option for history tables, which can improve performance.
  - Include system time columns in any unique indexes on history tables.
  - Choose between automatic system time adjustments and transaction rollbacks.
- Designing database schemas
  - Determine which tables require a trail of historical data.
  - Consider using vertical table splits to reduce history space consumption.
  - In partitioned database, use the same distribution key for the history table and its corresponding base table.
  - Understand the behavior of identity columns with business time and row splits.
- Querying temporal data
  - Include the "system_end" column in result sets to distinguish current data from historical data.
  - Know how to easily find the previous version of a row.
  - Know how to efficiently compare different versions of a row.
  - Detect and avoid gaps between business time periods if needed.
  - Use the compilation environment descriptors to accurately monitor the use of temporal special registers.
  - Understand how to formulate joins between tables with business time.
- Developing applications
  - Don't use "24" in the hour component of a time or timestamp value.
  - Create global variables for "mindate" and "maxdate", to enable consistent use of application-defined constants.
  - Define triggers or stored procedures if needed for temporal referential integrity.
  - Undo unintended data changes and revert tables to previous points in time with the sample stored procedure REVERT_TABLE_SYSTEM_TIME.
  - To update and insert ("upsert") data for a portion of business time, use a DELETE followed by an INSERT, rather than an UPDATE followed by a SELECT and an INSERT.
  - Choose the temporal sensitivity of application packages and stored procedures by using bind options and routine options.
- Managing historical data
  - Consider using a separate table space for history data.
  - Prune and archive historical data as needed.
  - Understand range partitioning for current and historical data.
  - Understand the recovery options for current and historical data.

Best practices for data and application modeling

The successful development of temporal database applications requires careful data modeling choices. This section helps you to make some of those decisions.

When to use system time, business time, or bitemporal

You might wonder when to use system time, when to use business time, and when to use both for bitemporal data management. The choice depends on the dimension of time that you want to track and the type of temporal queries that you need to support. The following table compares the key characteristics of system time and business time and helps you make the right choice.

Characteristics of system time	**Characteristics of business time**
Captures the time when changes happen to data inside a DB2 database	Captures the time when changes happen to business objects in the real world
Maintains a history of updated and deleted rows, generated by DB2	Maintains application-driven changes to the time dimension of business objects
History based on DB2 system timestamps	Dates or timestamps are provided by the application
DB2's physical view of time	Your application's logical view of time
Spans from the past to the present time	Spans past, present, and future time
System validity (transaction time)	Business validity (valid time)
Supports queries such as: *"Which policies were stored in the database on June 30?"*	Supports queries such as: *"Which policies were valid on June 30?"*

Be aware that the two queries in the last row of the table are very different from each other. A policy might have existed in a DB2 database on June 30 (*system time*), but that is not a reliable indicator of whether that policy was in effect or valid on June 30 (*business time*). For example, the policy might have expired in May, or might not come into effect until July. System time tracks when a record was added, changed, or deleted in a database, which is independent from when the data is valid in the real world (*business time*).

A policy might have been in effect on June 30 (*business time*), but might not have been added to a DB2 database until some later date (*system time*). For example, the policy might have been created and inserted into a DB2 database in July, but backdated with an effective date of June 30.

Consider the following guidelines when you choose temporal tables:

- Use system time and system-period temporal tables to track and maintain a history of when information was physically inserted, updated, or deleted inside your DB2 database.
- Use business time and application-period temporal tables, if you need to describe when information is valid in the real world, outside of DB2.
- Use bitemporal tables, if you need to track both dimensions of time. With a bitemporal table you can manage business time with full traceability of data changes and corrections.

Inclusive-inclusive or inclusive-exclusive time periods

A period is an interval of time that is defined by two date or timestamp columns in a temporal table. A period contains a begin column and an end column. The begin column indicates the beginning of the period and the end column indicates the end of the period.

If the specified end point is considered to be part of the period, then it is an inclusive-inclusive period, also known as a *closed-closed* period. For example, if a car insurance policy is valid from January 1 to June 30, and the last day of validity is June 30, then the period specified as "January 1 to June 30" is considered an inclusive-inclusive period.

In contrast, if the specified end point is the first point in time when the given information is no longer valid, then it is an *inclusive-exclusive* period, or *closed-open* period. For example, if a car insurance policy starts on January 1 and *expires* on July 1, meaning the last day of validity is June 30, then the period specified as "January 1 to July 1" is considered an inclusive-exclusive period.

There are many examples of inclusive-inclusive and inclusive-exclusive periods. For example, if John Doe works on project XYZ from April 1 to Sept 30, or if Mary is enrolled in a 5-day class from October 24 to 28, then these periods are understood to be *inclusive-inclusive*. However, if a hotel reservation specifies check-in on May 5 and check-out on May 9, or if a passport was issued on Feb 3, 2010 and expires on Feb 4, 2020, then these are *inclusive-exclusive* periods.

DB2 manages all system time and business time periods as inclusive-exclusive periods. Therefore it is recommended to also use inclusive-exclusive periods at the application level and to avoid mapping between inclusive-inclusive and inclusive-exclusive periods.

Benefit of inclusive-exclusive periods

Using inclusive-exclusive periods makes it very easy to detect or avoid gaps between time periods. Detecting or avoiding gaps can be important, for example to ensure that an interest rate is assigned to an account at all times or that an employee always has a single salary defined. Consider the following two rows in a table with business time:

empID	dept	salary	bus_start	bus_end
67890	M15	7000	2011-01-01	2011-06-01
67890	M15	7500	2011-06-01	9999-12-31

Figure 1: Inclusive-exclusive periods using DATE columns

The first row is valid up to and including 2011-05-31, but is no longer valid on 2011-06-01 which is the first day of validity for the second row. **If the business end value of the first row equals the business start value of the "next" row, then this property guarantees that there is no gap (and no overlap) between the two periods.** This important property holds regardless of the data type or "granularity" that you choose for period start and end points.

The table in Figure 2 uses TIMESTAMP(0) for the business start and business end values. Again, the simple equality of the bus_end value of the first row and the bus_start value of the second row ensures that there is no gap. The section "*How to detect gaps between periods*" on page 21 provides an SQL query to check for gaps.

product	price	bus_start	bus_end
90015	$99	2011-01-01-09.00.00	2011-06-01-17:30:00
90015	$129	2011-06-01-17:30:00	2011-06-01-22:00:00

Figure 2: Inclusive-exclusive periods using TIMESTAMP(0) columns

Complexity of inclusive-inclusive periods

DB2 does not use the inclusive-inclusive model because the logic for managing inclusive-inclusive periods is complex and dependent on the granularity (precision) of the data type.

Let's reexamine the preceding examples, this time using the inclusive-inclusive model. Is there a gap between the periods in the table in Figure 3? If you know that the granularity is and will always be the DATE data type, then there is no gap. The reason is that if the data type is DATE then there are no intermediate values between 2011-05-31 and 2011-06-01. However, if an application reads these values into a TIMESTAMP(0) variable, then the values are converted to 2011-05-31-00:00:00 and 2011-06-01-00:00:00, which results in a 1-day gap between them.

empID	dept	salary	bus_start	bus_end
67890	M15	7000	2011-01-01	2011-05-31
67890	M15	7500	2011-06-01	9999-12-31

Figure 3: Inclusive-inclusive periods with DATE columns

Similarly, there is no gap between the values 2011-06-01-17:29:59 and 2011-06-01-17:30:00 if they interpreted as TIMESTAMP(0), but there is a gap of one second if the values are interpreted as or converted to TIMESTAMP(6).

In general, the use of inclusive-inclusive periods is prone to errors if applications or SQL statements assume a granularity that doesn't match the data, or if date/time values are (possibly unintentionally) converted from one data type and precision to another.

How to map between inclusive-inclusive periods and inclusive-exclusive periods

If you have existing applications that use inclusive-inclusive periods, you can convert such closed-closed periods into the closed-open format used by DB2. For example, if the data type is DATE, then you can add + 1 DAY and -1 DAY in SQL operations where appropriate.

In the following INSERT statement the application-supplied inclusive end-date 2011-05-31 is incremented by one day to convert it into the exclusive end date that DB2 expects. Similarly, the subsequent query subtracts one day to return the original inclusive end date.

```
INSERT INTO employees(empID, dept, salary, bus_start, bus_end)
VALUES (67890, 'M15', 7000, '2011-01-01', DATE( '2011-05-31') + 1 DAY );

SELECT empID, dept, salary, bus_start, bus_end – 1 DAY as bus_end
FROM employees WHERE… ;
```

Keep in mind that such value conversions increase the complexity of application and SQL coding. For timestamps values, such conversions are cumbersome and can be error prone.

How to select columns for business time periods

When business records have real-world begin and end dates, such as the start and end of the coverage period of an insurance policy, don't use these real-world dates to define business time periods. It is better to introduce an additional pair of DATE or TIMESTAMP columns.

To understand this recommendation, let's consider two examples. In the first example, we manage simple customer addresses that consist of an ID, name, street, city, state, and country, as defined in the following table:

```
CREATE TABLE customer_address (
    customerID       INTEGER NOT NULL,
    name             VARCHAR(32),
    street           VARCHAR(64),
    city             VARCHAR(32),
    state            VARCHAR(32),
    country          VARCHAR(32),
  PRIMARY KEY(customerID) );
```

Dates or timestamps are usually not part of a mailing address. However, if you want to manage information about which address is valid for a certain customer at different points in time, you could add two DATE columns and declare them as a business time period:

```
CREATE TABLE customer_address (
    customerID       INTEGER NOT NULL,
    name             VARCHAR(32),
    street           VARCHAR(64),
    city             VARCHAR(32),
    state            VARCHAR(32),
    country          VARCHAR(32),
    valid_start      DATE NOT NULL,
    valid_end        DATE NOT NULL,
  PERIOD BUSINESS_TIME(valid_start, valid_end),
  PRIMARY KEY(customerID, BUSINESS_TIME WITHOUT OVERLAPS) );
```

Business time is used instead of system time to represent the time when address changes happen in the real world, not the time when those changes are made in the DB2 database. For example, if a customer notifies you that his address will change at the beginning of next month, you might issue an update in DB2 today. However, the valid_end value of the current address and the valid_start value of the new address are actually in the future.

Using real business dates for business time periods

With the previous examples in mind, let's look at an example for managing vehicle insurance policies. As illustrated in the following code sample, a simple policy record contains the policy ID, estimated annual mileage, rental car coverage indicator, maximum damage coverage, insurance premium, and the coverage period begin and end dates of the insurance coverage.

```
CREATE TABLE policy (
    id               INTEGER NOT NULL,
    annual_mileage   INTEGER,
    rental_car       CHAR(1),
    coverage_amt     INTEGER,
    premium          DECIMAL(8,2),
    coverage_start   DATE NOT NULL,
    coverage_end     DATE NOT NULL,
  PRIMARY KEY(id) );
```

Since the details of a policy can change during the coverage period, business time helps to accurately reflect which policy conditions are valid at which time. You might decide to use the columns coverage_start and coverage_end as a business time period, as shown next:

```
CREATE TABLE policy (
    id               INTEGER NOT NULL,
    annual_mileage   INTEGER,
    rental_car       CHAR(1),
    coverage_amt     INTEGER,
    premium          DECIMAL(8,2),
    coverage_start   DATE NOT NULL,
    coverage_end     DATE NOT NULL,
  PERIOD BUSINESS_TIME(coverage_start, coverage_end),
  PRIMARY KEY(id, BUSINESS_TIME WITHOUT OVERLAPS) );
```

The following operations show that this business time period definition is typically not optimal. For example, consider the following two events:

1. A new policy is purchased with a coverage period from January 1 to July 1, 2012.
2. Later the policy is updated to include coverage for rental car usage at no additional charge from June 1 onwards.

The following SQL statements perform the insert and the update of the policy, respectively:

```
INSERT INTO policy
VALUES (1, 25000, 'N', 1000000, 474.56, '2012-01-01', '2012-07-01');

UPDATE policy
  FOR PORTION OF BUSINESS_TIME FROM '2012-06-01' TO '2012-07-01'
SET rental_car = 'Y'
WHERE id = 1;
```

As a result of the update operation, the policy is now represented by two rows:

id	annual_ mileage	rc	coverage_ amt	premium	coverage_ start	coverage_ end
1	25000	**N**	1000000	474.56	2012-01-01	**2012-06-01**
1	25000	**Y**	1000000	474.56	**2012-06-01**	2012-07-01

However, neither row by itself reflects the fact that the policy has a coverage period from January 1 to July 1, 2012. If additional updates occur, then the policy might be represented by even more rows. As a result, applications would have to obtain the MIN(coverage_start) and the MAX(coverage_end) in order to determine the full coverage period. Clearly, this is undesirable.

This insurance policy example differs from the preceding customer address example because coverage_start and coverage_end are regular attributes of a policy even if no business time period is used. Therefore, these columns should remain regular attributes and should not be used to define a period. Instead, new DATE columns should be introduced, as in the customer address example.

Benefit of additional DATE columns for a business time period

Let's define the policy table with new DATE columns that serve as the business time period:

```
CREATE TABLE policy (
    id                INTEGER NOT NULL,
    annual_mileage    INTEGER,
    rental_car        CHAR(1),
    coverage_amt      INTEGER,
    premium           DECIMAL(8,2),
    coverage_start    DATE NOT NULL,
    coverage_end      DATE NOT NULL,
    valid_start       DATE NOT NULL,
    valid_end         DATE NOT NULL,
  PERIOD BUSINESS_TIME(valid_start, valid_end),
  PRIMARY KEY(id, BUSINESS_TIME WITHOUT OVERLAPS) );
```

If you insert the same policy information as before and perform the same update for the rental car coverage, you now have the following two rows:

id	annual_ mileage	rc	coverage_ amt	premium	coverage_ start	coverage_ end	valid_ start	valid_ end
1	25000	N	1000000	474.56	2012-01-01	2012-07-01	2012-01-01	**2012-06-01**
1	25000	Y	1000000	474.56	2012-01-01	2012-07-01	**2012-06-01**	2012-07-01

Each row shows that the policy coverage period runs from Jan 1 to July 1, 2012. This is correct because the update did not change the actual coverage period. The additional valid_start and valid_end columns accurately reflect the business validity of the two rows without obscuring the coverage period values.

In most, if not all, application scenarios, introducing an extra pair of columns for the business time period is more suitable than using the actual business dates as the period columns.

Best practices for administering database objects

Additional considerations apply when you are managing temporal tables rather than regular tables.

Identify periods and history tables in the DB2 catalog

Every system-period temporal table and every bitemporal table consists of a base table and an associated history table. When you update or delete a row in the base table, DB2 transparently inserts a copy of the old row into the associated history table. The base table and its history table are linked by an ALTER TABLE command that enables versioning and the maintenance of the history table. For example:

```
ALTER TABLE policy ADD VERSIONING USE HISTORY TABLE policy_history;
```

With this link between a base table and its history table, your queries have access to current data and data from any past point in time. Applications typically only access the base table and don't even need to know that there is a history table. DB2 transparently reads from the history table when a query on the base table requests data for a past point in time.

You can query the catalog view **syscat.periods** to determine whether a given table is a temporal table, which periods are defined, and whether there is a history table. For example, the following query reveals that the table POLICY has a system time and a business time period. Hence, it is a bitemporal table. The name of the associated history table is POLICY_HISTORY.

```
SELECT tabname, periodname, begincolname, endcolname, historytabname
FROM syscat.periods
WHERE tabname = 'POLICY';

TABNAME  PERIODNAME     BEGINCOLNAME  ENDCOLNAME  HISTORYTABNAME
-------- -------------- ------------- ----------- --------------
POLICY   SYSTEM_TIME    SYS_START     SYS_END     POLICY_HISTORY
POLICY   BUSINESS_TIME  BUS_START     BUS_END     -

  2 record(s) selected.
```

Use RESTRICT ON DROP to avoid unintended loss of history

While versioning is enabled, a base table and its history table act as a single unit. Therefore, dropping a base table with a DROP TABLE statement automatically drops its history table, without warning. To protect against accidentally dropping a history table, define the base table or the history table with the RESTRICT ON DROP option:

```
CREATE TABLE policy_history LIKE policy WITH RESTRICT ON DROP;
```

You can also enable the RESTRICT ON DROP option with an ALTER statement:

```
ALTER TABLE policy_history ADD RESTRICT ON DROP;
```

If you have enabled the RESTRICT ON DROP option for the base table or the history table, neither table can be dropped while versioning is enabled. For example, if you want to drop the history table but not the base table, perform the following three steps:

1. Stop versioning to break the link between the base table and the history table:
```
ALTER TABLE policy DROP VERSIONING;
```
2. If the history table was defined with RESTRICT ON DROP, remove that attribute:
```
ALTER TABLE policy_history DROP RESTRICT ON DROP;
```
3. Drop the base table.
```
DROP TABLE policy_history;
```

How to drop a column from a system-period temporal table

As with dropping tables, dropping columns can cause loss of history data. However, there is no RESTRICT ON DROP option at the column level to prevent accidental dropping of a column. Therefore, DB2 does not automatically propagate an ALTER TABLE … DROP COLUMN statement from the base table to the history table. Instead, DB2 rejects any attempt to drop a column while versioning is enabled.

If you are certain that it is safe to drop a column from both the base table and the history table, you must first disable versioning, drop the column from *both* tables, and then enable versioning again. (You don't need to stop versioning to *add* a column, which not incur data loss.)

If you drop a column from a base table, you must remove the same column from the history table so that the schemas of the two tables stay in sync. Otherwise versioning cannot be enabled.

While versioning is disabled, any updates and deletes on the base table are not propagated to the history table. Therefore, you might want to lock the base table in either shared or exclusive mode for the brief period when versioning is disabled. You can explicitly lock the base table for the current unit of work (transaction) with the LOCK TABLE statement. You must first disable autocommit, so that the lock is held until the COMMIT statement.

1. Temporarily prevent write operations on the base table:
```
LOCK TABLE policy IN SHARE MODE;
```
2. Stop versioning to break the link between the base table and the history table:
```
ALTER TABLE policy DROP VERSIONING;
```
3. Drop the columns:
```
ALTER TABLE policy         DROP COLUMN rental_car;
ALTER TABLE policy_history DROP COLUMN rental_car;
```
4. Re-enable versioning:
```
ALTER TABLE policy ADD VERSIONING
                   USE HISTORY TABLE policy_history;
```
5. Commit to unlock the base table:
```
COMMIT;
```

Define privileges on history tables

In many usage scenarios, a history table represents a critical audit log that must not be altered. However, history tables can be accessed directly and so it is possible to insert, update, or delete rows in a history table. Therefore you should use GRANT and REVOKE statements to define access privileges and prevent unauthorized users from manipulating the history table directly.

Consider our example with the policy and policy_history tables. Users who perform read or write operations on the policy table do not need explicit privileges on the policy_history table. For example, the following query with the FOR SYSTEM_TIME AS OF clause causes DB2 to transparently access the policy_history table:

```
SELECT *
FROM policy FOR SYSTEM_TIME AS OF '2010-06-16';
```

The user executing this query requires the SELECT privilege only on the policy table, not on the policy_history table. Similarly, any DELETE or UPDATE on the policy table causes DB2 to perform inserts into the policy_history table, but the user does not require the INSERT privilege on the history table. Consequently, you can revoke privileges for accessing the history table without preventing users from using all the temporal features of the base table.

Consider using APPEND ON for history tables

During normal operation, a history table is subject to insert and read activity only, not delete or update activity. Due to the absence of deletes and updates, a history table typically does not have a lot of free space that can be re-used by new rows. When a row is inserted into the history table the search for free space is typically unsuccessful and the row is added at the end of the table.

If inserts into the history table are limiting the performance of your workload, you can sometimes improve performance by enabling the APPEND ON option for the history table:

```
ALTER TABLE policy_history APPEND ON;
```

This option avoids the cost of the free space search and directly appends new rows at the end of the table. However, the APPEND ON option also prevents the reuse of space that is left behind by deleted rows. If you use DELETE statements to prune rows from the history table, observe at least one of the following two guidelines:

- Reorganize the history table periodically to release the free space that is left behind by DELETE operations.
- Don't use APPEND ON so that the free space can be reused by inserts.

Include system time columns in unique indexes on a history table

A unique key on a system-period temporal table or bitemporal table is typically not unique on the associated history table. The reason is that the history table can contain multiple versions of the same row, each with the same original key.

If you want to define the same indexes on a history table as on its base table, choose one of the following two options:

- Define the indexes on the history table as non-unique indexes by omitting the UNIQUE keyword from the CREATE INDEX statement. Otherwise update or delete operations on the base table will fail due to unique key violations on the history table.
- Include one or both of the columns of the SYSTEM_TIME period, such as sys_start or sys_end, in the definition of a unique index on the history table. The original primary key of the base table plus one or both of the system timestamps forms a unique key for the history table.

Tip: History indexes that combine the original primary key with system time columns are often useful to ensure good performance of temporal queries that use the FOR SYSTEM_TIME clause.

Choose between system-time adjustments and rollbacks

When you run a multi-user workload with concurrent read and write operations on a system-period temporal table, two concurrent transactions might modify the same row. Concurrent modifications of the same row can sometimes lead to inconsistent history rows. For illustration, let's consider the following example of transactions A and B and their write and commit operations that happen at different points in time:

Time	Transaction A	Transaction B
T1	`INSERT INTO mytable(c1, c2)` `  VALUES(1, 15);`	
T2		`INSERT INTO mytable(c1, c2)` `  VALUES(2, 30);`
T3		`COMMIT;`
T4	`UPDATE mytable SET c2 = 33` `WHERE c1 = 2;`	
T5	`COMMIT;`	

The timestamp of the first write operations within a transaction determines the system time timestamp that is assigned to *all* rows that are modified in that transaction. This timestamp identifies all rows that are written by the same transaction. In the example above, Transaction A inserts one row and updates another, and both rows have the system timestamp T1.

When Transaction B commits at time T3, the system_start timestamp of the inserted row (2, 30) is T2, the time of the first write operation in Transaction B. Hence, at time T3 the tables contain the following rows, and the star (*) indicates the uncommitted row from Transaction A:

At T3:

mytable

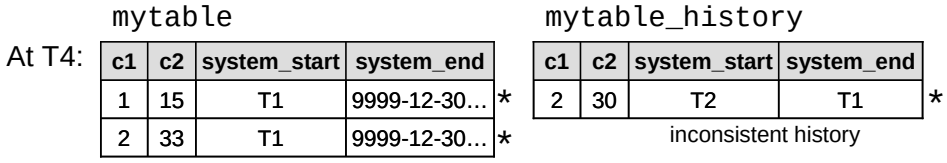
c1	c2	system_start	system_end	
1	15	T1	9999-12-30...	*
2	30	T2	9999-12-30...	

mytable_history

c1	c2	system_start	system_end

Subsequently, at time T4, Transaction A updates the row that Transaction B has inserted and generates a corresponding history row. By definition, the system_start value of this history row is the timestamp of the original row (T2), and the system_end value of this history row is the

timestamp of the update, which is T1 (the time of the first write in Transaction A). In this situation, the system_start value T2 is greater than the system_end value T1, which violates the constraint that system_start must be less than system_end. This inconsistent state is shown in the following picture:



This violation can be resolved in one of two ways:

(a) **Rollback (SQL20528N)**. The default behavior is to roll-back Transaction A and return error code 20528.

(b) **System time adjustment (SQL5191W)**. Alternatively, you can choose to let DB2 increase the conflicting timestamp T1 to T2 + *delta*, the next possible timestamp after T2. This adjustment happens only if you set the database configuration parameter SYSTIME_PERIOD_ADJ to YES. Whenever an adjustment happens, warning SQL5191W is issued.

Best practices for database schema design

Designing database schemas with temporal tables is subject to several trade-offs, which are discussed in this section.

Carefully determine which tables require history

Considering the great benefits of traveling back in time and examining a past state of your data, you might be tempted to add a history table and versioning to *all* tables in a database. After all, this enables you to track any data change anywhere in the database. However, adding history tables also increases resource consumption, such as the following:

- History tables consume storage space. Although this can be greatly mitigated by compression, historical data requires some storage space.
- Inserts into history tables are subject to logging and increase the need for log I/O bandwidth.
- Inserts into history tables use pages in the bufferpool and increase the need for page cleaning activity.
- History tables typically have at least one index, which adds to the space and I/O cost.
- History tables increase the elapsed time and space requirement of database backups.

These types of resource consumption depend on several factors, most notably the following ones:

- The frequency of updates and deletes in your workload because they cause inserts into history tables.
- The average number of rows affected by updates and deletes.
- The number of indexes defined on history tables and the number of columns in those history indexes.

Before you enable versioning for tables in a database, determine which tables have a business need for history tracking. Based on the expected number of updates and deletes per day, you can estimate the size of a history table and the number of history inserts per day. This gives you a feel for the additional cost that a given history table would introduce.

Also, you should determine how much history to retain, and when and how to prune (and possibly archive) old history rows. The topic of history retention is further discussed in the section "*Best practices for managing history*" on page 29.

Consider vertical table splits to reduce history space consumption

Updating just a single field of a row in a system-period temporal table or bitemporal table causes DB2 to write a before-image of the *entire* row into a history table. In other words, versioning always happens at the granularity of rows even if only a small portion of a row changes. With this in mind, consider the following situations:

- You have a table that contains a LOB or XML column with potentially very large values. You know that these values rarely get updated. Copying the entire LOB or XML document to the history table when some *other* column in the same row is updated can increase the space consumption of the history drastically. You can save space if you store the LOB or XML in a separate table. This separate table can also be temporal table, if you want to keep a history of deleted LOB or XML values.

- You have a very wide table that has hundreds of columns, for example, 500. If you need to retain history for only 50 of the 500 columns, you can save space by splitting the table vertically into two tables: one system-period temporal table or bitemporal table that contains the 50 columns that require history tracking, and one regular (non-temporal) table for the remaining 450 columns.

A vertical table split can save significant amounts of storage space, but also affects the performance of INSERT, UPDATE, DELETE, and SELECT statements in the following manner:

- Inserting a new (logical) record now requires two INSERT statements instead of just one, affecting two tables (and at least two primary key indexes) instead of just one. This can reduce insert performance.
- Update operations now produce less history, which reduces logging and can improve update performance.
- Deleting a logical record now requires two DELETE statements, one for each of the two tables. If one of the tables is a regular table without history, deletions now produce less history, which reduces logging and can improve delete performance.
- Queries that read full (logical) records require a join across the two tables, which can be slower than non-join queries against a single wider table. Queries that access columns from only one of the tables can be faster.

You should evaluate these trade-offs for your specific application scenario and your goals regarding performance and space consumption.

In a partitioned database, collocate current and history data

When you create a table in a partitioned database (DPF) you can declare one or multiple columns as the distribution key that is used to hash the rows of the table across the database partitions. For a system-period temporal table, make sure that you use the same distribution key for the base table and the associated history table. This ensures that the two tables are *collocated*, which means that rows with the same key values reside in the same database partition.

For a system-period temporal table, collocation of base and history table implies that a current row and its historical versions are stored in the same database partition. Consequently, when a row is updated or deleted in the base table, the required insert into the history table happens within the same database partition. This is good for performance because it avoids shipping of new history rows across database partitions.

Understand the behavior of identity columns with business time

Identity columns are a convenient mechanism to automatically assign a unique value to each row in a table. If a table has a business time period then UPDATE statements may apply to a specified portion of business time, which can lead to row-splits. As a result, multiple rows may exist for the same logical object but describe this object for different portions of time. If the table has an identity column, be aware that each row will have a distinct identity value even if multiple rows describe the same logical object.

Let's consider the following example for illustration. The insert statement adds information for object 55 to the table. This information is valid from January 1, 2013 to January 1, 2014. The new row receives the generated identity value 1. Next, object 55 is updated for the portion of time from May 1 to August 1, 2013. This update operation results in a three-way row split.

```
CREATE TABLE bustest(
       id         INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY,
       object_id  INTEGER NOT NULL,
       data       VARCHAR(16),
       bus_start  DATE NOT NULL,
       bus_end    DATE NOT NULL,
    PERIOD BUSINESS_TIME(bus_start, bus_end),
    PRIMARY KEY(object_id, BUSINESS_TIME WITHOUT OVERLAPS));

INSERT INTO bustest (object_id, data, bus_start, bus_end)
  VALUES(55, 'abcde', '2013-01-01', '2014-01-01');

SELECT * FROM bustest;

ID          OBJECT_ID   DATA            BUS_START  BUS_END
----------- ----------- --------------- ---------- ----------
          1          55 abcde           01/01/2013 01/01/2014

  1 record(s) selected.

UPDATE bustest
FOR PORTION OF BUSINESS_TIME FROM '2013-05-01' TO '2013-08-01'
SET data = 'xyz'
WHERE OBJECT_ID = 55;
                                       -- example continues on the next page
```

```
SELECT * FROM bustest;

ID          OBJECT_ID   DATA            BUS_START  BUS_END
----------- ----------- --------------- ---------- ----------
          1          55 xyz             05/01/2013 08/01/2013
          2          55 abcde           01/01/2013 05/01/2013
          3          55 abcde           08/01/2013 01/01/2014

  3 record(s) selected.
```

The SELECT statement reveals the result of the row-split. The original row with identity value 1 has been updated for the specified portion of time. Two new rows have been added to the table to represent the portions of time that were not affected by the UPDATE. These new rows have identity values 2 and 3.

As a result, an identity column is not a suitable identifier for the logical objects that are represented in a table with business time. Instead, each identity value uniquely identifies a different period-object combination. If you need generated object identifiers that do not change in the presence of row-splits, consider the use of a sequence. A sequence enables you to increment the generated value only upon insert of a new object and does not change during row-splits.

Best practices for temporal queries

This section describes tips and tricks for writing queries against temporal tables.

Distinguish current from history rows in a result set

A common question is how to distinguish current rows from history rows in a result set. Consider the following query that retrieves all rows as of on June 16, 2010:

```
SELECT *
FROM policy FOR SYSTEM_TIME AS OF '2010-06-16';
```

The query retrieves zero or more rows from both the policy and the policy_history table:

- The date 2010-06-16 is converted to the timestamp 2010-06-16-00.00.00.000000000000 and the query retrieves all rows from the base table (policy) that were inserted at or before this timestamp and have *not* been deleted or updated. Those rows were current then and are still current now, and they match the temporal predicate.
- The query also retrieves the prior versions of any rows that were deleted or updated since 2010-06-16-00.00.00.000000000000. These are prior versions that were current on 2010-06-16-00.00.00.000000000000 and they are retrieved from the history table.

The result is a union of some rows from the base table and some rows from the history table. For each row, you can determine which table it came from based on the value in the sys_end column:

- Rows from the current table always have the value 9999-12-30-00.00.00.000000000000 in the sys_end column.
- Rows from the history table always have a sys_end value that is older (lower) than the current timestamp.

Find the previous version of a row

A common requirement is to find the previous version of a row, which is the latest version before the current version. This previous version is the most recent version of the row in the history table. The following query retrieves the previous version of the policy with ID = 1414.

```
SELECT *
FROM policy_history
WHERE id = 1414
  AND sys_end = (SELECT MAX(sys_end)
                 FROM policy_history
                 WHERE id = 1414);
```

This query references the history table explicitly, but you can obtain the same result also without referencing the history table. The next query exploits the fact that the system *start* timestamp of the current row equals the system *end* timestamp of its previous version. This relationship exists because the current row started its existence at the same point in time when the previous version ended and was inserted into the history table. In other words, if you update a row one or multiple times, there are no gaps between the system periods of the versions of that row.

```
SELECT prev.*
FROM policy cur,
     policy FOR SYSTEM_TIME BETWEEN '0001-01-01' AND CURRENT_TIMESTAMP prev
WHERE cur.id  = 1414
  AND prev.id = 1414
  AND cur.sys_start = prev.sys_end;
```

Another option is to retrieve both the current row and the previous version:

```
SELECT *
FROM policy FOR SYSTEM_TIME FROM '0001-01-01' TO '9999-12-31'
WHERE id = 1414
ORDER BY sys_start DESC
FETCH FIRST 2 ROWS ONLY;
```

Compare two versions of a row

Sometimes you might want to compare two versions of a row, for example to compute the delta between the two rows. The following query examines the change in the coverage amount of a specific policy between two different points in system time.

```
SELECT p1.coverage_amt AS Jan01,
       p2.coverage_amt AS May31,
       p2.coverage_amt - p1.coverage_amt AS Delta
FROM policy FOR SYSTEM_TIME AS OF '2011-01-01' p1,
     policy FOR SYSTEM_TIME AS OF '2011-05-31' p2
WHERE p1.id = 1414
  AND p2.id = 1414;
```

Note that the join query above returns no result if there is data for only one of the two specified points in time. For such cases you can consider a full outer join:

```
SELECT p1.coverage_amt AS Jan01,
       p2.coverage_amt AS May31,
       p2.coverage_amt - p1.coverage_amt AS Delta
FROM policy FOR SYSTEM_TIME AS OF '2011-01-01' p1
     FULL OUTER JOIN
     policy FOR SYSTEM_TIME AS OF '2011-05-31' p2
     ON p1.id = p2.id
WHERE p1.id = 1414
   OR p2.id = 1414;
```

The following query performs a similar comparison between the current data and the data from three months ago:

```
SELECT p1.coverage_amt AS Current,
       p2.coverage_amt AS ThreeMonthsAgo,
       p2.coverage_amt - p1.coverage_amt AS Delta
FROM policy FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP  p1
     FULL OUTER JOIN
     policy FOR SYSTEM_TIME AS OF CURRENT_TIMESTAMP - 3 MONTHS p2
     ON p1.id = p2.id
WHERE p1.id = 1414 OR p2.id = 1414;
```

If you want to compare information between two different points in business time, you can use the same queries, but replace the keywords FOR SYSTEM_TIME with FOR BUSINESS_TIME.

Detect gaps between periods

Some applications that use business time might have to prevent gaps between the business time periods of particular entity. For example, assume you use business time to reflect that a product has different prices for different periods of time. A gap between two periods for the same product would imply that the price is undefined for a certain range of time, which can be undesirable.

Figure 4 shows sample data that describes the different prices for a product during the year 2013. There is a gap from March 1 to April 1, 2013 because no price information exists for this month.

product_id	price	bus_start	bus_end
1	14.95	01/01/2013	02/01/2013
1	16.95	02/01/2013	03/01/2013
1	19.95	04/01/2013	05/01/2013
1	20.95	05/01/2013	12/31/2013

Figure 4: Product price information with a temporal gap

The following query detects such gaps. The query orders the periods chronologically and compares each start date with the end date of the preceding period. If the start date of one period is greater than the end date of the previous period, then a gap exists. The clause "PARTITION BY product_ID" ensures that this check happens separately for each product that might appear in the table.

```
SELECT product_ID,
       previous_end AS gap_start,
       bus_start    AS gap_end
FROM
  ( SELECT product_ID, bus_start, bus_end,
           MIN(bus_end) OVER (PARTITION BY product_ID ORDER BY bus_start
                              ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)
           AS previous_end
    FROM product)
WHERE bus_start > previous_end;


PRODUCTID  GAP_START   GAP_END
---------  ----------  ----------
        1  03/01/2013  04/01/2013

  1 record(s) selected.
```

You can include the test for gaps in database triggers to detect and prevent gaps automatically when rows are inserted, updated, or deleted.

Monitor temporal queries when special registers are active

If you have existing applications, reporting tools, or SQL scripts that you want to run against a specific point in system time or business time, it can be tedious to add FOR SYSTEM_TIME and FOR BUSINESS_TIME clauses to all of your queries. Luckily, you don't have to. You can use the special registers CURRENT TEMPORAL SYSTEM_TIME and CURRENT TEMPORAL BUSINESS_TIME to set a database session to a specific point in system time or business time, respectively. Subsequently, you can run existing queries against the specified time without having to modify the SQL. To disable these special registers, set them to NULL.

If one or both of these special registers is set to a non-null value, the execution plan of a query (that is, the output of the db2exfmt tool) shows the original statement and the rewritten statement where DB2 has added additional predicates to enforce the temporal special registers.

If you monitor SQL activity with the MON_GET_PKG_CACHE_STMT monitoring function or with the activity event monitor, only the original SQL is captured and the influence of the temporal special registers is not immediately visible. However, the captured information for each SQL statement includes a compilation environment descriptor (comp_env_desc). This descriptor can be passed into the table function COMPILATION_ENV to extract the settings that were applicable when the statement was compiled. This information in compilation environment reveals which temporal special register has or hasn't affected a particular query.

The following example shows how to retrieve the compilation environment for the statement 'SELECT * FROM policy'. The information shows the values of the temporal special registers that have affected the compilation and execution of the query.

```
SET CURRENT TEMPORAL SYSTEM_TIME '12/01/1997';
SET CURRENT TEMPORAL BUSINESS_TIME '06/15/1998';

SELECT * FROM policy;

SELECT VARCHAR(t.name, 30) AS name, VARCHAR(t.value, 40) AS value
FROM TABLE(MON_GET_PKG_CACHE_STMT(NULL,NULL, NULL,-1)) as p,
     TABLE(COMPILATION_ENV(p.comp_env_desc)) as t
WHERE p.stmt_text = 'SELECT * FROM policy';

NAME                          VALUE
----------------------------- ---------------------------------
ISOLATION                     CS
QUERY_OPTIMIZATION            5
MIN_DEC_DIV_3                 NO
DEGREE                        0
SQLRULES                      DB2
REFRESH_AGE                   +00000000000000.000000
RESOLUTION_TIMESTAMP          2012-03-19-17.13.11.000000
FEDERATED_ASYNCHRONY          0
CURRENT TEMPORAL BUSINESS_TIME 1998-06-15-00.00.00.000000000000
CURRENT TEMPORAL SYSTEM_TIME   1997-12-01-00.00.00.000000000000
SCHEMA                        MNICOLA
MAINTAINED_TABLE_TYPE         SYSTEM

  12 record(s) selected.
```

Understand how to join application-period temporal tables

When you join two tables that contain business time periods, you must determine the appropriate join condition for your application. In addition to the equality of the join keys, you might also have to compare the business time periods of the parent table with the business time periods in the child table. (Note that one row in a child table can have several corresponding parent rows with identical key values but different business time periods.)

You can consider join conditions such as the following between parent and child rows with business time:

- Equality of keys, regardless of the business time periods in the parent and child rows.
- Equality of keys, and the business time period of the child row must be *identical* to the period of a *single* parent row.
- Equality of keys, and the business time period of the child row must be *fully contained* within the periods of *one or multiple* parent rows.
- Equality of keys, and the business time period of the child row must be *fully contained* within the period of a *single* parent row.
- Equality of keys, and the business time period of the child row must *overlap* with the period of a parent rows.
- Equality of keys, and the business time period of the child row must *start within* the period of a parent row.
- Equality of keys, and the business time period of the child row must *start after* the period of a parent row.
- *etc.*

The required business semantics of the join dictates the temporal condition that you need to add to the join condition. You can code the temporal part of the join condition as SQL predicates in a WHERE clause, but you can also create user-defined functions as convenient abbreviations for "contains", "overlaps", "starts-after", and so on. (Temporal join conditions are related to the topic of temporal referential integrity on page 25.)

Best Practices for Application Development

When coding time-based applications, observe the guidelines and recommendations described in the following sections.

Be careful at midnight! (Don't use 24!)

In DB2, the smallest time value is 00.00.00 and the largest is 24.00.00. Both these values represent midnight. Because DB2 allows "24" as the hour component of a time or timestamp value, the largest possible timestamp value in DB2 is 9999-12-31-**24**.00.00.000000000000.

However, the time and timestamp data types in some programming languages do not allow "24" to represent an hour of the day. In Java the largest time value is 23.59.59. This is important if you retrieve time or timestamp values from DB2 into Java variables of type java.sql.Timestamp or java.sql.Time. For example, the DB2 timestamp value 2010-11-03-24.00.00 denotes midnight on November 3 but a Java application automatically converts it to 2010-11-04-00.00.00, which is the beginning of November. 4 Not only the hour but also the date portion of the timestamp changed.

The following table provides additional examples of how certain DB2 timestamp values change when they are retrieved into a Java application:

Value in DB2	**Same value when retrieved from DB2 into a Java time or timestamp variable**
2010-11-03-24.00.00	2010-11-04-00.00.00 (next day)
2011-12-31-24.00.00	2012-01-01-00.00.00 (next year)
9999-12-31-24.00.00	10000-01-01-00.00.00 (next millennium)

To avoid such issues, do not use 24 as the hour component of a DB2 time or timestamp value. Instead, use 23.59.59 to denote the largest time of a day.

Create global variables for "mindate" and "maxdate"

When you manage business time, such as effective dates, you might have data that is effective since an unknown point in time in the past. Similarly, you might have records that are valid "until further notice" or "until the end of time".

In such scenarios it is common practice to use a minimal date value to denote the "beginning of time" and a maximum date value to indicate the "end of time" or "infinity". To ensure that all applications use the same minimum and maximum dates, you can define constants that are global SQL variables with a fixed value:

```
CREATE VARIABLE MYCONSTANTS.MAX_DATE  DATE ('9999-12-31');
CREATE VARIABLE MYCONSTANTS.MIN_DATE  DATE ('0000-01-01');
```

These statements create two variables MAX_DATE and MIN_DATE of type DATE in the schema MYCONSTANTS. If needed, you can define similar variables of type TIMESTAMP. Wherever these variable names are used in queries, inserts, updates, or deletes, DB2 replaces the variables with their assigned value. Defining these constants in their own schema avoids conflicts with column names or other SQL identifiers.

Determine your need for temporal referential integrity checks

Traditional referential integrity requires every row in a child table to have a corresponding row in a parent table. For example, consider two tables: product (the parent) and promotion (the child). Each row in the promotion table describes a special offer for one of the products in the product table. Referential integrity is violated if there is a promotion for a product that is not listed in the product table.

Temporal referential integrity is an even stronger condition that can be enforced between two tables with business time periods. If the product and promotion tables have business time periods that describe the validity periods of products and promotions, you might want to ensure that every row in the promotion table has a business time period that is *contained* in the business time period of the corresponding parent row in the product table. This constraint is an example of temporal referential integrity. You can enforce such temporal referential integrity with triggers or stored procedures. For details, see the article "*Managing Time in DB2 - with Temporal Integrity!*"

As another example, consider a logical business record such as a purchase order that contains multiple items. You might want to use a business time period for the *entire* purchase order even if the order is stored across multiple normalized tables, such as an order table and an item table. In this case, you might want to ensure that every row in the item table has a business period that is *identical* to the business period of its parent row in the order table. This equality can be enforced in DB2 by a traditional foreign key constraint on the columns orderID, bus_start, and bus_end.

Revert a table to prior state and reverse unintended changes

Even in the best of worlds, "bad" data can enter a database. A user might issue INSERT, UPDATE, or DELETE statements that are later discovered to be incorrect. A traditional solution is to perform a database restore from the most recent backup plus a roll-forward recovery. This approach is very intrusive and incurs database downtime.

You can take a simpler and less intrusive approach when the affected table is a system-period temporal table with an associated history table. In this case, you can delete the offending rows from the base table, read their correct previous versions from the history table, and insert them into the base table. This process is called *revert*.

The delete operations that are issued as part of the revert process are tracked in the history table, so you can undo the revert process itself if necessary.

You can code your own custom SQL to perform revert operations, or use the sample stored procedure REVERT_TABLE_SYSTEM_TIME that is available in the **sqllib/samples/clp** directory. You can use the revert stored procedure as-is or customize it for your particular needs. The procedure takes the following parameters as input:

- **`tableschema`**: Schema name of the table to revert
- **`tablename`**: Unqualified name of the table to revert to a past point in time
- **`timestamp`**: A past point in time to which you want to revert the table. This timestamp can indicate one of two things:
  (a) The exact point in time to which you want to revert the table. For example, the time of the last known good state of the table. In this case, set the **`before`** parameter to "NO".
  (b) The timestamp of the bad transaction that must be undone, and you want to revert the table to the last state *before* this transaction. In this case, set **`before`** to "YES".
- **`before`**: YES / NO, as explained above.
- **`row_identifying_predicate`**: An optional predicate to run the revert operation on the subset of rows that match this predicate. If not specified, all current rows are potentially subject to the revert operation.

How to update and insert ("upsert") data for a portion of business time

Consider the following sample scenario. You run a business and are planning promotions to sell some products at a discount for certain periods of time. You represent the temporal validity of a promotion by a business time period. For the last few weeks before Christmas, you decide that product 9105 will be on sale for $19.95 and you issue the following INSERT statement:

```
INSERT INTO promotion(promoID, product, price, bus_start, bus_end)
  VALUES (1, 9105, 19.95, '2011-12-01', '2011-12-25');
```

The new promotion row and its period are shown in Figure 5:

Table: promotion

promoID	product	price	bus_start	bus_end
1	9105	$19.95	2011-12-01	2011-12-25

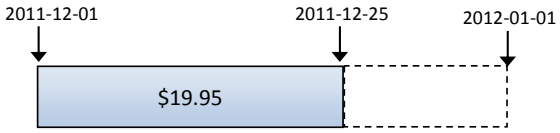


Figure 5: Initial state of the promotion for product 9105

Subsequently, you decide to modify and extend the promotion: you want to set the price to $14.95 from December 15 through December 31. That is, you want to achieve the state shown in Figure 6. What is the best way to make this change?

Table: promotion

promoID	product	price	bus_start	bus_end
1	9105	$19.95	2011-12-01	2011-12-15
1	9105	$14.95	2011-12-15	2012-01-01

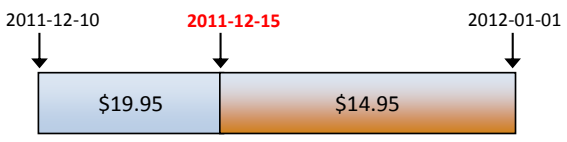


Figure 6: Required state of the promotion for product 9105

An initial but suboptimal approach

A first step in changing the information in Figure 5 to the state in Figure 6, might be to issue an UPDATE statement for the portion of time from December 15 to January 01:

```
UPDATE promotion
     FOR PORTION OF BUSINESS_TIME FROM '2011-12-15' TO '2012-01-01'
SET price = 14.95
WHERE promoID = 1 AND product = 9105;
```

However, the update changes only the existing period, which ends on 2011-12-25. The update does not extend the existing period to the end of December, nor does it insert a new period. The result of this update is not entirely what we wanted (see Figure 7).

promoID	product	price	bus_start	bus_end
1	9105	$19.95	2011-12-01	2011-12-15
1	9105	$14.95	2011-12-15	2011-12-25

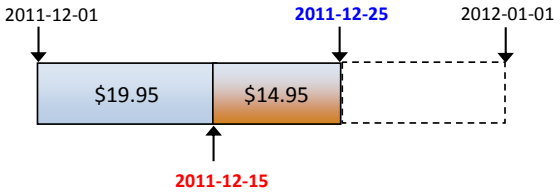


Figure 7: Unwanted state of the promotion for product 9105

To reach the state in Figure 6, you could first issue the previous UPDATE statement and then an INSERT statement to add another row for the period from 2011-12-25 to 2012-01-01. But, how would you know that the missing period needs to start at 2011-12-25? This date stems from the original promotion, so you need a query to obtain this date before using it in an INSERT:

```
SELECT bus_end INTO :BE
FROM promotion
WHERE promoID = 1 AND product = 9105 AND ... ;

INSERT INTO promotion VALUES (1, 9105, 14.95, :BE, '2012-01-01');
```

The insert statement then produces the state shown in Figure 8.

promoID	product	price	bus_start	bus_end
1	9105	$19.95	2011-12-01	2011-12-15
1	9105	$14.95	2011-12-15	2011-12-25
1	9105	$14.95	2011-12-25	2012-01-01

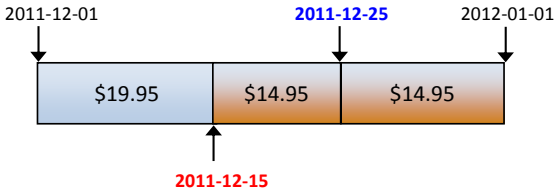


Figure 8: New state of the promotion for product 9105

Logically, this is the required state of the data. However, this approach has two drawbacks. First, you need a query to examine the existing period before you can issue the INSERT statement to "fill the blank". This query can potentially be complex, especially if there are already multiple existing promotion rows for the given key. Second, this approach creates three rows (Figure 8) to represent information that could be represented by only two rows (Figure 6). The second and

third row in Figure 8 represent consecutive periods with identical price information, and they do not automatically collapse into one.

Note that a MERGE statement does not provide a better solution. Although update and delete operations within a MERGE statement can use a FOR PORTION OF clause, they do not automatically adjust to existing periods in the table, as would be required in this case.

A better approach

A better solution is to issue a DELETE followed by an INSERT statement, as shown in Figure 9. The delete for portion of business time from December 15 to January 1 removes any existing information that might conflict with the new promotion that you want to insert. The insert can then add a row for the new promotion from December 15 to January 1. An additional query to examine the existing state of the data is not necessarily needed. Another advantage is that the required information is represented by only two rows.

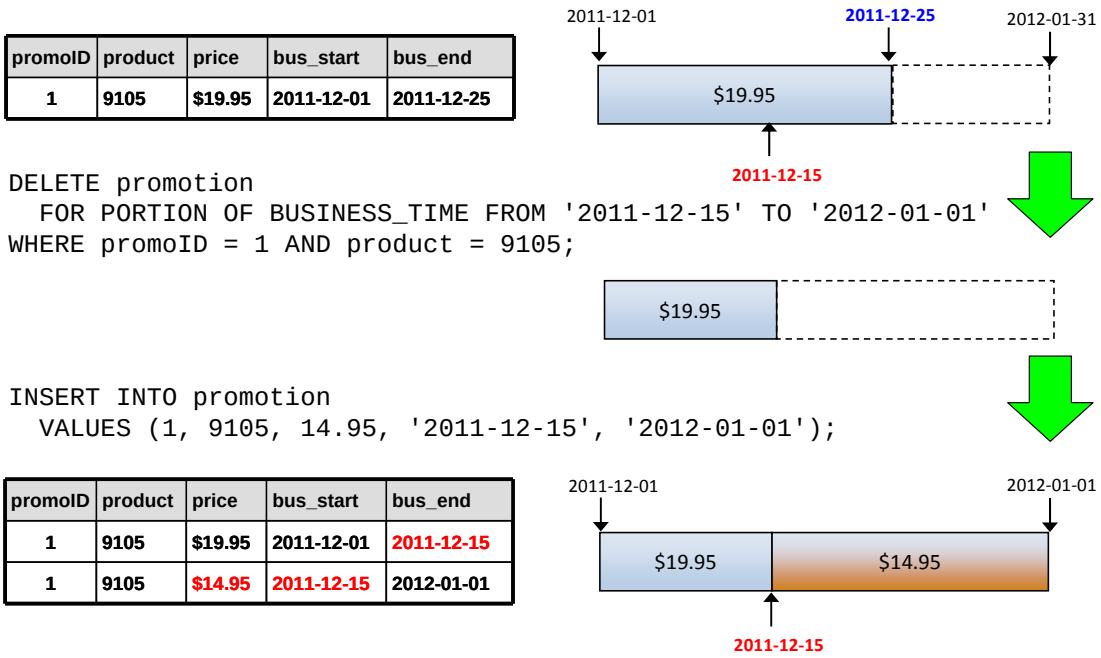


Figure 9: Preferred approach to change the promotion for product 9105

Choose the temporal sensitivity of packages and stored procedures

DB2 10 introduces two special registers called CURRENT TEMPORAL SYSTEM_TIME and CURRENT TEMPORAL BUSINESS_TIME. If you set these registers to a date or timestamp value, then queries against temporal tables will behave as if they had a FOR SYSTEM_TIME AS OF or FOR BUSINESS_TIME AS OF clause. These special registers allow you to run queries against different points in time without changing the query text.

When you bind application packages, you can use the options SYSTIMESENSITIVE and BUSTIMESENSITIVE in the BIND command to choose whether the packages should obey the temporal special registers or not. The bind option SYSTIMESENSITIVE determines whether references to system-period temporal tables in static and dynamic SQL statements in a package will be affected by the value of the CURRENT TEMPORAL SYSTEM_TIME special register. Similarly, the option BUSTIMESENSITIVE indicates whether access to application-period temporal tables will be affected by the CURRENT TEMPORAL BUSINESS_TIME special register.

The default value for both bind options is YES for new packages and NO for existing packages that you migrate from a previous DB2 release. Set these bind options to YES only if you want the temporal registers to affect the packages with. Otherwise, set these bind options to NO.

By default, new SQL procedures that you create are also sensitive to the CURRENT TEMPORAL SYSTEM_TIME and CURRENT TEMPORAL BUSINESS_TIME special registers. If a SQL procedure doesn't need to be sensitive to the temporal registers, set the following routine options before creating the procedure:

```
CALL SET_ROUTINE_OPTS('SYSTIMESENSITIVE NO BUSTIMESENSITIVE NO')
```

Best practices for managing history

Capturing the before images of updated and deleted rows in history tables can significantly add to the growth of your database. Therefore, managing the accumulating history data deserves special attention.

Consider using a separate table space for history data

When you create a system-period temporal table, the base table and the associated history table can be in the same or different table spaces. Placing the history table in a separate table space provides more flexibility and offers the following (non-exhaustive list of) options:

- You can place current and history data in different storage locations and storage groups
- You can choose a different page size or buffer pool for current and history data
- You can backup current and history data independently, if desired
- You can restore and recover current and history data independently, if desired

Pruning and archiving history data

The size of a history table and the rate at which it grows depends on the frequency of updates and deletes in your workload and the average number of rows affected by UPDATE and DELETE statements. To curb the resource consumption, it is common to prune and possibly archive history data regularly.

Generally, archiving data from history tables is not significantly different than archiving data in other tables. Consider using IBM Optim Data Growth software to implement your archiving strategy.

The criteria for when particular rows can be removed from the history table depend heavily on the nature of the application and business rules that govern the data. Hence, DB2 does not provide a fixed retention policy for history rows. Instead, you should implement a pruning and archiving strategy that meets your business requirements. Consider some of the following examples:

- If you can map your pruning strategy for a history table to a range-partitioning scheme, then detaching partitions is an efficient way of removing history data and making it available for archiving or destruction.
- You can decide to delete history rows that are older than a certain age. The following statement removes history rows older than three years:

  ```
  DELETE FROM policy_history
  WHERE sys_end <  CURRENT_DATE – 3 years;
  ```

- You can choose to remove the history of rows that have been deleted from the base table, that is, history rows for which no current row exists in the base table anymore:

  ```
  DELETE FROM policy_history
  WHERE id NOT IN (SELECT id FROM policy);
  ```

- You can decide to retain the last N versions of a row, and delete history rows if more than N versions exist for a given row.
- You can combine some of the previous criteria or prune rows selected by your own query to reflect the appropriate business rules.

The privilege to perform DELETE, UPDATE, and INSERT operations on a history table should be revoked from most users to prevent unintended or unauthorized manipulation of history data. Typically, only the user ID that periodically prunes history data should have the DELETE privilege on history tables.

How to use range-partitioning for current and history data

Any temporal table can also be a range-partitioned table. A history table that belongs to a system-period temporal table or a bitemporal table can also be range-partitioned.

The partitioning scheme of a history table may differ from the partitioning of its base table. For example, you can partition the current data by month and the history data by year. However, ensure that the ranges that you defined for the history table partitions can absorb any rows that are moved from the base table to the history table, as explained in the following example.

Consider the following table definitions, which serve as an example of insufficient ranges in the partitioned history table.

```
CREATE TABLE policy (
   id             INTEGER PRIMARY KEY NOT NULL,
   annual_mileage INTEGER,
   rental_car     CHAR(1),
   coverage_amt   INTEGER,
   sys_start      TIMESTAMP(12) GENERATED AS ROW BEGIN NOT NULL,
   sys_end        TIMESTAMP(12) GENERATED AS ROW END NOT NULL,
   trans_start    TIMESTAMP(12) GENERATED AS TRANSACTION START ID,
  PERIOD SYSTEM_TIME (sys_start, sys_end) )
  PARTITION BY RANGE(sys_start)
    (STARTING('2012-01-01') ENDING ('2014-12-31') EVERY 1 MONTH );

CREATE TABLE policy_history (
   id             INTEGER NOT NULL,
   annual_mileage INTEGER,
   rental_car     CHAR(1),
   coverage_amt   INTEGER,
   sys_start      TIMESTAMP(12) NOT NULL,
   sys_end        TIMESTAMP(12) NOT NULL,
   trans_start    TIMESTAMP(12))
  PARTITION BY RANGE(sys_start)
    (STARTING('2012-01-01') ENDING ('2013-12-31') EVERY 3 MONTHS );

ALTER TABLE policy ADD VERSIONING USE HISTORY TABLE policy_history;
```

The policy table is partitioned by month; the policy_history table is partitioned by quarter. This difference in granularity is not a problem. However, the policy table has partitions for sys_start values up to 2014-12-31 while the policy_history table has partitions only up to 2013-12-31. If a row in the policy table with a sys_start value of 2014-01-01 or greater is deleted, the insert into the history table fails because it has no partition to accepts this sys_start value. Hence, the entire delete transaction fails. To avoid this problem, define the history table such that the total range of all its partitions is always equal to or greater than the total range of partitions in the base table.

Detaching partitions

After versioning has been enabled with the ALTER TABLE...ADD VERSIONING statement, you can still detach partitions from the history table for pruning and archiving purposes. However, to detach a partition from the base table you must first stop versioning with the ALTER TABLE...DROP VERSIONING statement.

When you stop versioning and detach a partition from the base table, this partition becomes an independent table. It retains all three timestamp columns (row begin, row end, transaction start ID) but not the PERIOD SYSTEM_TIME declaration. The rows in the detached partition are not automatically moved into the history table. If you choose to move these rows into the history table yourself, change the sys_end value of every row from 9999-12-30-00.00.00.000000000000 to the current timestamp. This change is necessary to reflect the point in time when the rows changed from being current to being history. If you do not make this change, temporal queries might return unexpected results.

Attaching partitions

You can attach a table to a partitioned base or history table while versioning is enabled. The table that you attach is not required to have a PERIOD SYSTEM_TIME declaration but it must have all three timestamp columns defined as in the base table.

While versioning is enabled, you cannot use the SET INTEGRITY statement with the FOR EXCEPTION clause. The reason is that moving any exception rows into an exception table cannot be recorded in the history table, which jeopardizes the ability to audit of the base table and its history. However, you can temporarily disable versioning, perform SET INTEGRITY with the FOR EXCEPTION clause, and then enable versioning again.

Recovery considerations for current and history data

When you roll forward the table space for a system-period temporal table to a *point in time*, you must also roll forward the table space for the associated history table to the same point in time in the same ROLLFORWARD statement.

When you roll forward to the end of logs, you can roll forward the base table and the history table together or independently from each other. Independent recovery to end of logs can be useful if you expect the recovery of the history table to take longer than the recovery of the base table. In that case you can choose to recover the base table first and make at least the current data available to your applications (possibly in read-only mode), while the recovery of the history table is still in process. You can also consider using the following 12-step recovery procedure:

1. Recover the base table first (restore and roll forward to the end of logs).
2. Disable versioning for that table.
3. Create a new and empty history table to catch any new history that might get generated.
4. Enable versioning of the base table with the new history table.
5. Bring your application back online.
6. Recover the original history table (restore and roll forward to the end of logs).
7. When recovery and roll forward of the original history is completed, lock the base table to enforce read-only access.
8. Disable versioning (ALTER TABLE...DROP VERSIONING).
9. Move any rows from the new history table into the original recovered history table.
10. Enable versioning with the old history table (ALTER TABLE...ADD VERSIONING).
11. Commit, to release the lock on base table.
12. Drop the "temporary" history table.

If you use a range-partitioned history table, you can replace steps 7 through 12 by simply attaching the recovered history table to the new history table.
This recovery strategy is useful only if you want to bring the current data online sooner than the recovery of the history table space is completed. Keep in mind that queries against past points in time will return no or incomplete history information until the original history table has been brought online. This behavior may or may not be acceptable for a given application.

Best practices for migrating to temporal tables in DB2

Separate articles describe guidelines for migrations from home-grown temporal solutions to the temporal capabilities in DB2. See "Further Reading" below.

Summary

The temporal and bitemporal capabilities in the DB2 for Linux, UNIX, and Windows Version 10 product provide sophisticated support for time-aware data management, compliance, and auditing requirements. This paper provides an initial set of best practices for the design and operation of temporal applications and databases with DB2. These best practices continue to evolve and expand over time, which will be reflected in subsequent versions of this paper.

Further Reading

For an introduction to DB2's temporal features, read the white paper
"**A Matter of Time: Temporal Data Management in DB2**".
http://www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/

For a more detailed discussion of temporal referential integrity, see the article
"**Managing Time in DB2 with Temporal Consistency**"
http://www.ibm.com/developerworks/data/library/techarticle/dm-1207db2temporalintegrity/

Guidelines for the migration of existing temporal solutions to the temporal features in DB2 are available in the article series "**Adopting temporal tables in DB2**"
http://www.ibm.com/developerworks/data/library/techarticle/dm-1210temporaltablesdb2/
http://www.ibm.com/developerworks/data/library/techarticle/dm-1210temporaltablesdb2pt2/

For questions, contact the author or post in the **DB2 Temporal discussion forum**:
http://www.ibm.com/developerworks/forums/forum.jspa?forumID=2316

DB2 Best Practices:
http://www.ibm.com/developerworks/data/bestpractices/db2luw/

Acknowledgements

Thanks to Martin Sommerlandt, Eileen Lin, Stan Musker, and Serge Boivin for their help with this paper.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

Without limiting the above disclaimers, IBM provides no representations or warranties regarding the accuracy, reliability or serviceability of any information or recommendations provided in this publication, or with respect to any results that may be obtained by the use of the information or observance of any recommendations provided herein. The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any recommendations or techniques herein is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Anyone attempting to adapt these techniques to their own environment does so at their own risk.

This document and the information contained herein may be used solely in connection with the IBM products discussed in this document.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-

level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.