DB2 Information Management Software

A Matter of Time: Temporal Data Management in DB2 for z/OS

Cynthia M. Saracco, saracco@us.ibm.com
Matthias Nicola, mnicola@us.ibm.com
Lenisha Gandhi, lenisha@us.ibm.com

IBM Silicon Valley Laboratory

Executive summary

You could say it was just a matter of time. Industry regulations and competitive pressures are prompting IT managers to maintain more data for longer periods of time and to provide better ways for business users to analyze past, current, and future events. To help organizations achieve these goals, IBM has built greater awareness of time into DB2 10 for z/OS.

New DB2 temporal data management technology enables firms to track and query historical, current, and future conditions in a straightforward and efficient manner. The result is a simpler way to implement auditing and compliance initiatives, to pinpoint (and correct) human errors, to ensure the integrity of data over time, and to assess changing business conditions. Instead of hard-coding greater awareness of time into database applications, triggers, and stored procedures, firms can use simple SQL statements to instruct DB2 to automatically manage multiple versions of their data as well as track effective dates for changing business conditions, such as insurance coverage, product prices, medical prescriptions, and interest rates on credit cards.

This paper introduces you to the key concepts and capabilities of DB2's temporal data management support on z/OS. It explains when this technology can be useful and provides several examples to help you understand how easy it can be to deploy it. In many cases, deployment can be transparent to existing applications, including pre-built applications purchased from software vendors.

Introduction

If "time is money," then DB2's temporal data management technology is designed to save you some of both. Built-in support for managing multiple versions of data and tracking effective business dates can save database administrators and application developers considerable time and effort.

In prior releases, database professionals were forced to create triggers or complex application logic to manage time-dependent conditions. Now, DB2 10 for z/OS minimizes or eliminates such efforts through the introduction of new table design options, new query syntax and semantics (derived from emerging ANSI/ISO SQL standardization efforts), and other new features.

When is temporal data management needed?

Before delving into the technical aspects of DB2's temporal data support, let's consider a few application scenarios:

1. An internal audit requires a financial institution to report on changes made to a client's records during the past 5 years.
2. A pending lawsuit prompts a hospital to reassess its knowledge of a patient's medical condition just before a new treatment was ordered.
3. A client challenges an insurance firm's resolution of a claim involving a car accident. The insurance firm needs to determine the policy's terms in effect when the accident occurred.
4. An online travel agency wants to detect inconsistencies in itineraries. For example, if someone books a hotel in Rome for eight days and reserves a car in New York for three of those days, the agency would like to flag the situation for review.
5. A retailer needs to ensure that no more than one discount is offered for a given product during any period of time.
6. A client inquiry reveals a data entry error involving the three-month introductory interest rate on a credit card. The bank needs to retroactively correct the error (and compute a new balance, if necessary).

For each of these situations, time is critical. DB2's new temporal data management support helps firms implement time-aware applications and queries with minimal effort, as we'll discuss shortly. However, before exploring DB2's new capabilities, we need to introduce a few time-related concepts, including the differences between *system time* and *business time.*

Basic concepts

System time involves tracking when changes are made to the state of a table, such as when an insurance policy is modified or a loan is created. *Business time* involves tracking the effective dates of certain business conditions, such as the terms of an insurance policy or the interest rate of a loan. (Business time is sometimes referred to as "valid time" or "application time.") Indeed, some organizations need to track both types of temporal data in a single table; such tables would be considered *bitemporal.*

How do these concepts apply to the application scenarios we discussed in the previous section? Scenarios 1 and 2 require knowledge of system time because understanding the historical state(s) of one or more tables is important. Scenarios 3 through 6 require knowledge of business time because understanding and managing the effective dates of various business conditions is important. Furthermore, Scenario 6 may also require system time (i.e., it may require bitemporal support) if the bank wants to retroactively correct the data entry error *and* maintain a record of when the error was corrected.

Time *periods* apply to both business time and system time. As you might imagine, a period indicates the starting and ending points of a time interval. With DB2, administrators identify two columns in a table to indicate the start and end times of a period. Simple extensions to the syntax of the CREATE TABLE and ALTER TABLE statements accomplish this, enabling administrators to employ temporal data support for new or existing tables.

DB2 uses an *inclusive, exclusive* approach for modeling time periods. Simply put, the period's start time is included in the period but its end time is not. So, if the end time of an insurance policy was recorded as midnight on Dec. 31, 2007, the policy would have been active until midnight (but not at midnight).

DB2 10 extends SQL syntax to support temporal functionality, which this paper will discuss shortly. IBM is working with the ANSI and ISO SQL standard committees to incorporate these extensions into the next edition of SQL standard, expected in 2011. As of this writing, the ANSI and ISO SQL committees have accepted an early IBM proposal for tables with system time periods. IBM is working on a proposal involving business time (application periods) and bitemporal tables, which it expects to bring forward to the SQL standard committees shortly.

Benefits of temporal data management

DB2's built-in support for managing temporal data reduces application logic and ensures consistent handling of time-related events across all applications that run against your database, including purchased applications. Through simple declarative SQL statements, administrators can instruct DB2 to automatically maintain a history of database changes or track effective business dates, eliminating the need for such logic to be hand-coded into triggers, stored procedures, or in-house applications. This, in turn, helps firms adhere more quickly to new compliance initiatives. Furthermore, a consistent approach to managing temporal data reduces query complexity and promotes enhanced analysis of time-dependent events.

Let's consider the effort required to manually implement a subset of what DB2 provides for temporal data support. For example, imagine that you wanted to manage effective dates in a small number of tables. You could write triggers and database stored procedures to capture the necessary temporal logic, or you could use a combination of

triggers and application code (perhaps written in Java or COBOL). In either case, it could easily require several weeks to design, code, and test your work.

By contrast, if you used DB2's built-in support for business time periods, you would merely need to write and execute a few simple SQL statements – a task that's likely to require less than an hour. Of course, if you need a broader range of temporal capabilities, such as support for system time, bitemporal data, and temporal uniqueness, the contrast in effort would become even sharper. It could take months to manually implement a more comprehensive temporal solution using triggers and stored procedures or application logic. That's much longer than it would take to simply write the appropriate DB2 SQL statements, which we'll discuss shortly.

Sample scenario

To help you understand DB2's temporal data management support, we use a common application scenario and sample data throughout the remainder of this paper. The scenario involves car insurance policies, which we've represented in a single table for simplicity. The table tracks a subset of typical information associated with such policies: a policy identifier (ID), vehicle identification number (VIN), the estimated mileage the car is driven annually, whether or not a rental car will be provided if the car needs repair as a result of a claim, and the total coverage amount (including accidental property damage, medical expenses, etc).

Fig. 1 illustrates the basic structure of the POLICY table without any temporal support.

Fig. 1: Sample POLICY table (without temporal support)

ID	VIN	annual_mileage	rental_car	coverage_amt
1111	A1111	10000	Y	500000

Let's explore how DB2's temporal support can help you manage such insurance policies. We'll turn to system time first.

Managing data versions with system time

DB2's support for system time enables you to automatically track and manage multiple versions of your data. Specifically, by defining a table with a system time period, you're instructing DB2 to automatically capture changes made to the state of your table and to save "old" rows in a *history table* – a separate table with the same structure as your current table. Temporal queries referencing your current table will cause DB2 to transparently access this history table when needed, as you'll see shortly. This feature enables you to work with historical data easily, avoiding the need for complex WHERE clauses with various timestamp and join conditions.

Creating a table with system time

Defining a table with system time involves three simple steps:

1. **Create the base table for current data.** Include three TIMESTAMP(12) columns – two for the start/end points of the system time and one for the transaction start time. (DB2 uses the transaction start time column to track when the transaction first executed a statement that changes the table's data.) You can define all three TIMESTAMP columns as GENERATED ALWAYS so that DB2 will automatically generate these values on INSERT, UPDATE, and DELETE. This relieves you from specifying values for these columns when writing to the database and also ensures that the timestamps are accurate. Optionally, you may define these column as IMPLICITLY HIDDEN so that they won't show up in SELECT * statements.
2. **Create the history table.** Define the structure of this table to be identical to the table containing the current data. You can achieve this easily by using a CREATE TABLE . . . LIKE statement.
3. **Alter the current table to enable versioning and identify the history table.**

Let's step through an example to see how easy it is to instruct DB2 to automatically maintain multiple versions of your data using system time.

```
-- Step 1: Create a table with a SYSTEM_TIME period.
-- (Our definition specifies that the TRANS_START column will be hidden.)
CREATE TABLE policy (
   id             INT primary key not null,
   vin            VARCHAR(10),
   annual_mileage INT,
   rental_car     CHAR(1),
   coverage_amt   INT,
   sys_start      TIMESTAMP(12) GENERATED ALWAYS AS ROW BEGIN NOT NULL,
   sys_end        TIMESTAMP(12) GENERATED ALWAYS AS ROW END NOT NULL,
   trans_start    TIMESTAMP(12) GENERATED ALWAYS
                             AS TRANSACTION START ID IMPLICITLY HIDDEN,
  PERIOD SYSTEM_TIME (sys_start, sys_end)
 );

-- Step 2: Create an associated history table.
CREATE TABLE policy_history like policy;

-- Step 3: Enable versioning.
ALTER TABLE policy ADD VERSIONING USE HISTORY TABLE policy_history;
```

Fig. 2 illustrates the two empty tables created as a result. Throughout this section, we depict the history table in gray to help you easily distinguish it from the current table.

Fig. 2: Sample tables for our system time scenario

POLICY table (contains current data)

ID	VIN	annual_mileage	rental_car	coverage_amt	sys_start	sys_end	trans_start

POLICY_HISTORY table (contains historical data)

ID	VIN	annual_mileage	rental_car	coverage_amt	sys_start	sys_end	trans_start

You can also use the ALTER TABLE statement to modify existing tables to track system time. To do so, you would need to add appropriate TIMESTAMP(12) columns and define the PERIOD SYSTEM_TIME.

Inserting data into a table with system time

Inserting data into a table with system time isn't any different from inserting data into an ordinary table. For example, imagine that on Nov. 15, 2010, you needed to enter two new car insurance policy records into your POLICY table. The following statements accomplish this:

```
INSERT INTO policy(id, vin, annual_mileage, rental_car, coverage_amt)
     VALUES(1111,  'A1111',  10000, 'Y', 500000);

INSERT INTO policy(id, vin, annual_mileage, rental_car, coverage_amt)
     VALUES(1414,  'B7777',  14000, 'N', 750000);
```

When inserting each row into the current table, DB2 generates appropriate TIMESTAMP(12) values for its system time columns and the transaction start time. Note that none of these were referenced in the INSERT statements – DB2 automatically records the necessary information. Fig. 3 depicts the contents of the POLICY and POLICY_HISTORY tables as a result of this query. (To make our example easier to follow, Fig. 3 shows only the date portion of the TIMESTAMP(12) columns. Dates appear in YYYY-MM-DD format. Since the column containing the transaction's start time was defined as hidden, we've omitted it from Fig. 3 as well.)

Fig. 3: Current and history table contents after INSERTs on Nov. 15, 2010

POLICY

ID	VIN	annual_mileage	rental_car	coverage_amt	sys_start	sys_end
1111	A1111	10000	Y	500000	2010-11-15	9999-12-31
1414	B7777	14000	N	750000	2010-11-15	9999-12-31

POLICY_HISTORY (empty)

ID	VIN	annual_mileage	rental_car	coverage_amt	sys_start	sys_end

The SYSTEM_START values in the POLICY table reflect when the rows were inserted (on Nov. 15, 2010, in our example). The SYSTEM_END values are set to Dec. 31, 9999, the maximum allowable time supported by DB2. Effectively, this indicates that these rows have not expired – that is, the rows contain current data.

Updating data in a table with system time

When you update current data, DB2 automatically maintains an old version of the data in the appropriate history table. This happens transparently without any programming or user effort.

Imagine that the following statement is executed on Jan. 31, 2011 to change the coverage amount for Policy 1111 to 750000:

```
UPDATE policy
SET coverage_amt = 750000
WHERE id = 1111;
```

Let's explore how DB2 processes this statement. As shown in Fig. 4, DB2 updates the value of the row in the current table. In addition, it moves a copy of the old row to the history table. For both tables, DB2 correctly records the system time start and end values for these rows. In particular, DB2 sets the SYS_END column value for this row in the history table to the time of the transaction that issued the UPDATE statement. All this occurs automatically in a manner that is transparent to the user. (Although not shown in Fig. 4, DB2 also records the transaction start time in both tables.)

In effect, DB2's processing of this UPDATE statement records that Policy 1111 had a coverage amount of 500000 set from Nov. 15, 2010 to Jan. 31, 2011; thereafter, the coverage amount was set to 750000.

Fig. 4: Current and history table contents after UPDATE on Jan. 31, 2011

POLICY

ID	VIN	annual_mileage	rental_car	coverage_amt	sys_start	sys_end
1111	A1111	10000	Y	750000	2011-01-31	9999-12-31
1414	B7777	14000	N	750000	2010-11-15	9999-12-31

POLICY_HISTORY

ID	VIN	annual_mileage	rental_car	coverage_amt	sys_start	sys_end
1111	A1111	10000	Y	500000	2010-11-15	2011-01-31

As you might expect, any subsequent updates to policies are handled in a similar manner. For example, assume that Policy 1111 is updated on Jan. 31, 2012 to change several details of the insurance policy, such as the annual mileage estimate, rental car coverage, and overall coverage amount. Here is the corresponding UPDATE statement:

```
UPDATE policy
SET annual_mileage = 5000, rental_car='N', coverage_amt = 250000
WHERE id = 1111;
```

Executing this statement causes DB2 to automatically modify the POLICY and POLICY_HISTORY tables as shown in Fig. 5.

Fig. 5: Current and history table contents after UPDATE on Jan. 31, 2012

POLICY

ID	VIN	annual_mileage	rental_car	coverage_amt	sys_start	sys_end
1111	A1111	5000	N	250000	2012-01-31	9999-12-31
1414	B7777	14000	N	750000	2010-11-15	9999-12-31

POLICY_HISTORY

ID	VIN	annual_mileage	rental_car	coverage_amt	sys_start	sys_end
1111	A1111	10000	Y	500000	2010-11-15	2011-01-31
1111	A1111	10000	Y	750000	2011-01-31	2012-01-31

Deleting data from a table with system time

When you delete current data, DB2 automatically removes the data from the current table and maintains an old version of the data in the appropriate history table. DB2 sets the end time of the (deleted) data in the history table to the transaction start time of the DELETE statement. This happens transparently without any programming or user effort. As you'll see shortly, users can access this deleted data (i.e., the old data versions) through queries that contain an appropriate time period specification.

Imagine that the row for Policy 1414 is deleted on March. 31, 2012 with the following statement:

```
DELETE FROM policy WHERE id = 1414;
```

As shown in Fig. 6, DB2 removes the row from the current table and records the old version in the history table, setting the SYS_END column value for that row to the date of its deletion (March 31, 2012).

Fig. 6: Current and history table contents after DELETE on March 31, 2012

POLICY

ID	VIN	annual_mileage	rental_car	coverage_amt	sys_start	sys_end
1111	A1111	5000	N	250000	2012-01-31	9999-12-31

POLICY_HISTORY

ID	VIN	annual_mileage	rental_car	coverage_amt	sys_start	sys_end
1111	A1111	10000	Y	500000	2010-11-15	2011-01-31
1111	A1111	10000	Y	750000	2011-01-31	2012-01-31
1414	B7777	14000	N	750000	2010-11-15	2012-03-31

Querying a table with system time

Querying a table with system time is simple. The syntax and semantics of basic SELECT statements remain unchanged. In particular, SELECT statements without any period specifications apply to data in the current table, as you might expect. Thus, existing applications, stored procedures, and database reports won't be impacted by adding system time support to your existing tables. Instead, you'll simply be able to include three new period specifications in your SELECT statements to transparently access historical data (or a combination of current and historical data).

Let's explore a few examples so you can see how easy it is to write temporal queries involving system time. But first, we'll start with the most basic scenario – a situation where you need to access only the most current information.

Imagine that your current and history tables contain the car insurance policy data shown in Fig. 6 (in the previous section). Then consider the following query:

```
SELECT coverage_amt FROM policy WHERE id = 1111;
```

As you might expect, DB2 returns one row with a coverage amount of 250000. (This is amount stored in the row that contains current information about Policy 1111.)

What happens if you want to work with older versions of data? You simply include one of three supported period specifications in the FROM clause of your query:

- `FOR SYSTEM_TIME AS OF ...` This enables you to query your data as of a certain point in time.
- `FOR SYSTEM_TIME FROM ... TO ...` This enables you to query your data from a certain time to a certain time. DB2 uses an *inclusive, exclusive* approach for this period specification. In other words, the specified start time is included in the period but the specified end time is not.
- `FOR SYSTEM_TIME BETWEEN ... AND ...` This enables you to query your data between a range of start/end times. DB2 uses an *inclusive, inclusive* approach for this period specification. In other words, the specified start and end times are both included in the period.

Let's step through an example. To obtain information about the coverage amount recorded in the database for Policy 1111 for Dec. 1, 2010, you could write the following query:

```
SELECT coverage_amt
FROM policy FOR SYSTEM_TIME AS OF TIMESTAMP('2010-12-01')
WHERE id = 1111;
```

To resolve this query, DB2 transparently accesses data in the history table to retrieve the correct information – specifically, to return a value of 500000 for this query. Note that you didn't need to reference the history table in your query. The FOR SYSTEM_TIME period specification causes DB2 to automatically access the history table as appropriate.

Let's consider another example. To determine the total number of policy records for vehicle A1111 since Nov. 30, 2011, you could write the following query:

```
SELECT count(*)
FROM policy FOR SYSTEM_TIME FROM TIMESTAMP('2011-11-30')
                              TO TIMESTAMP('9999-12-31')
WHERE vin = 'A1111';
```

Given our sample data, this query returns the value of 2. As shown in Fig. 6, one row in the current table and one row in the history table (the second row) meet the query's criteria. Note that this query references only the current table in the FROM clause, but DB2 automatically accesses the history table due to the system time period specification.

Tracking effective dates with business time

As we mentioned earlier, business time involves tracking when certain business conditions are, were, or will be valid. For example, a given product might have been priced at $45 during one month and $50 during another month. Or a credit card may have an interest rate of 16% one year and 18% the next year. Business time is useful in such situations, because it enables applications to track and manage effective dates easily.

Like system time, business time requires the use of a time period – the start and end points of the business condition. However, unlike system time, there is no separate history table. Past, present, and future effective dates and their associated business data are all maintained in a single table. In addition, users supply the start/end values for their business time period columns when they write data to the database. Finally, there is no need for a transaction start time column.

Let's explore how to use this new DB2 technology in our sample application scenario.

Creating a table with business time

Creating a table with business time merely involves including appropriate columns for the start/end points of the time period and a PERIOD BUSINESS_TIME clause. The business time start/end columns can be date or timestamp data types.

Here's a simple example that creates a table for car insurance policies, including their effective business dates. In this example, the BUS_START and BUS_END columns are defined as DATE data types. The PERIOD BUSINESS_TIME clause instructs DB2 to use these columns to track the start and end points of business time values for each row. To ensure temporal data integrity, DB2 automatically generates an implicit constraint to enforce that BUS_START values are less than BUS_END values.

```
CREATE TABLE policy (
  id              INT NOT NULL,
  vin             VARCHAR(10),
  annual_mileage  INT,
  rental_car      CHAR(1),
  coverage_amt    INT,
  bus_start       DATE NOT NULL,
  bus_end         DATE NOT NULL,
  PERIOD BUSINESS_TIME(bus_start, bus_end),
  PRIMARY KEY(id, BUSINESS_TIME WITHOUT OVERLAPS) );
```

The primary key constraint in this CREATE TABLE statement uses the optional keywords BUSINESS_TIME WITHOUT OVERLAPS. This constraint instructs DB2 to ensure that primary key values are unique for any point in business time. In terms of our insurance policy example, BUSINESS_TIME WITHOUT OVERLAPS means that there cannot be two "versions" or "states" of the same policy that are valid at the same time.

You can also use the ALTER TABLE statement to modify existing tables to track business time. To do so, you would need to add appropriate DATE or TIMESTAMP columns and define the PERIOD BUSINESS_TIME.

Inserting data into a table with business time

Inserting a row into a table with business time is straightforward: you simply need to supply appropriate values for all NOT NULL columns, including the columns representing business time start and end values. For example, to insert a few rows into our sample POLICY table with business time, we could issue these statements:

```
INSERT INTO policy
  VALUES(1111,'A1111',10000,'Y',500000,'2010-01-01','2011-01-01');
INSERT INTO policy
  VALUES(1111,'A1111',10000,'Y',750000,'2011-01-01','9999-12-31');
INSERT INTO policy
  VALUES(1414,'B7777',14000,'N',750000,'2008-05-01','2010-03-01');
INSERT INTO policy
  VALUES(1414,'B7777',12000,'N',600000,'2010-03-01','2011-01-01');
```

Fig. 7 illustrates the resulting contents of this table.

Fig. 7: POLICY table after INSERT statements

ID	VIN	annual_mileage	rental_car	coverage_amt	bus_start	bus_end
1111	A1111	10000	Y	500000	2010-01-01	2011-01-01
1111	A1111	10000	Y	750000	2011-01-01	9999-12-31
1414	B7777	14000	N	750000	2008-05-01	2010-03-01
1414	B7777	12000	N	600000	2010-03-01	2011-01-01

It may help to summarize the contents of this table in business terms. Very briefly, the data shows that Policy 1111 had a coverage amount of 500000 in effect from Jan. 1, 2010 until Jan. 1, 2011. From Jan. 1, 2011 onwards, a coverage amount of 750000 is in effect. Similarly, the table shows that from May 1, 2008 to March 1, 2010, Policy 1414 had coverage of 750000 for a specific vehicle with an estimated annual mileage of 14000. Effective March 1, 2010 to Jan. 1, 2011, the coverage for this policy changed to 600000, and the insured vehicle was expected to be driven 12000 miles annually.

Let's consider the impact of the temporal uniqueness constraint we defined earlier on this table (with the BUSINESS_TIME WITHOUT OVERLAPS clause). Imagine that we issued the following INSERT statement:

```
INSERT INTO policy
  VALUES(1111,'A1111',10000,'Y',900000,'2010-06-01','2011-09-01');
```

DB2 would reject this statement and issue an error message because the statement attempts to add a row for Policy 1111 during the same time that one or more other rows are considered valid for this policy. This would violate the temporal uniqueness constraint. If we intended to adjust the coverage of Policy 1111 from June 1, 2010 until Sept. 1, 2011, we would accomplish this with an appropriate UPDATE statement.

Updating data in a table with business time

As you might imagine, you can still write traditional UPDATE statements for tables with business time periods. In addition, you can also use the new FOR PORTION OF BUSINESS_TIME clause to restrict the update to a specific business time period. If your update impacts data in a row that isn't fully contained within the time period you specified, DB2 will update the row range specified by the period clause and insert additional rows to record the old values for the period not included in the update operation. Let's review an example to see how this works.

Imagine that you want to update information for Policy 1111 for a portion of time from June 1, 2010 to Sept. 1, 2011– specifically, you want to alter the coverage amount for that period of time. Here's how you could write the UPDATE statement:

```
UPDATE policy
  FOR PORTION OF BUSINESS_TIME FROM '2010-06-01' TO '2011-09-01'
SET coverage_amt = 900000
WHERE id = 1111;
```

Note that the temporal restriction in the query (FOR PORTION OF BUSINESS_TIME FROM . . . TO . . .) appears after the table name, not as part of the WHERE clause.

As shown in Fig. 7, there were originally two rows for Policy 1111. Both rows are affected by our UPDATE statement, because the portion of business time that is being updated overlaps partially with the business period of each row. This overlap is illustrated in the upper part of Fig 8. When DB2 applies the UPDATE, each of the two original rows is split into two rows, as illustrated in the lower part of Fig 8. DB2 adjusts the effective dates of the rows automatically.

Fig. 8. Row splits caused by the UPDATE statement

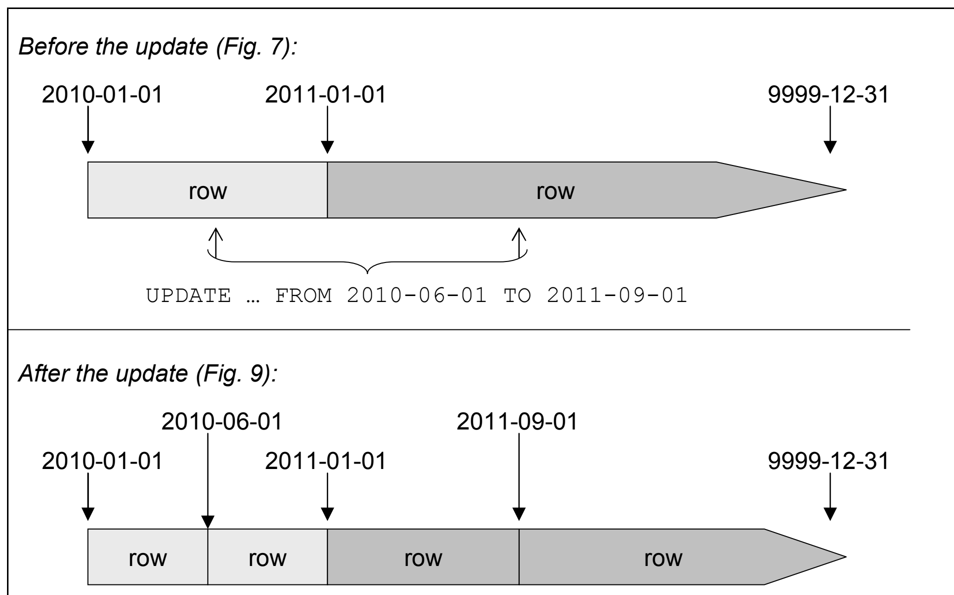


Fig. 9 shows the resulting state of the POLICY table. The first row from Fig. 7 gets split into two new rows, shown in light gray in Fig. 9. The second row from Fig. 7 also gets split into two new rows, shown in dark gray in Fig. 9.

Fig. 9. POLICY table after UPDATE of Policy 1111

ID	VIN	annual_mileage	rental_car	coverage_amt	bus_start	bus_end
1111	A1111	10000	Y	500000	2010-01-01	2010-06-01
1111	A1111	10000	Y	900000	2010-06-01	2011-01-01
1111	A1111	10000	Y	900000	2011-01-01	2011-09-01
1111	A1111	10000	Y	750000	2011-09-01	9999-12-31
1414	B7777	14000	N	750000	2008-05-01	2010-03-01
1414	B7777	12000	N	600000	2010-03-01	2011-01-01

Deleting data from a table with business time

If you want to delete data from a table with business time periods, you can restrict the delete operation to a specific range of time by specifying the FOR PORTION OF BUSINESS_TIME clause. If a row to be deleted has data that isn't fully contained within the specified time range, DB2 will ensure that the appropriate information from the row is preserved. Let's look at an example to clarify this.

Imagine that a client wants to suspend his car insurance policy from June 1, 2010 to Jan. 1, 2011. Assuming the client is referring to Policy 1414, the following DELETE statement will accomplish this task:

```
DELETE FROM policy
  FOR PORTION OF BUSINESS_TIME FROM '2010-06-01' TO '2011-01-01'
WHERE id = 1414;
```

Fig. 10 illustrates the resulting contents of the table. Note that DB2 altered the final row (shown in gray) to reflect the new BUS_END date for Policy 1414.

Fig. 10: POLICY table after DELETE involving a portion of business time

ID	VIN	annual_mileage	rental_car	coverage_amt	bus_start	bus_end
1111	A1111	10000	Y	500000	2010-01-01	2010-06-01
1111	A1111	10000	Y	900000	2010-06-01	2011-01-01
1111	A1111	10000	Y	900000	2011-01-01	2011-09-01
1111	A1111	10000	Y	750000	2011-09-01	9999-12-31
1414	B7777	14000	N	750000	2008-05-01	2010-03-01
1414	B7777	12000	N	600000	2010-03-01	2010-06-01

Querying a table with business time

Querying a table with business time is simple. Three optional clauses enable you to specify temporal queries so that you can assess past, current, and future business conditions. Of course, you can still write basic SELECT statements (i.e., non-temporal queries) against a table with a business time period, and DB2 processing of such queries will remain unchanged.

We'll explore a few examples so you can see how easy it is to write temporal queries involving business time. But first, we'll start with the most basic scenario – a situation where you don't need to consider any temporal conditions.

Imagine that your POLICY table contains the information shown in Fig. 11. (This is the same data shown in Fig. 7, immediately after we created the POLICY table and inserted four new rows.)

Fig. 11: Sample contents of POLICY table

ID	VIN	annual_mileage	rental_car	coverage_amt	bus_start	bus_end
1111	A1111	10000	Y	500000	2010-01-01	2011-01-01
1111	A1111	10000	Y	750000	2011-01-01	9999-12-31
1414	B7777	14000	N	750000	2008-05-01	2010-03-01
1414	B7777	12000	N	600000	2010-03-01	2011-01-01

To determine the total number of insurance records that you have for Policy 1111, you could write the following query:

```
SELECT COUNT(*)
FROM policy
WHERE id = 1111;
```

Since this query contains no temporal predicates, DB2 returns a value of 2.

What if you want to consider various temporal conditions for these insurance policies? You simply include one of three supported period specifications in your query's FROM clause, right after the table name:

- `FOR BUSINESS_TIME AS OF ...`
- `FOR BUSINESS_TIME FROM ... TO ...`
- `FOR BUSINESS_TIME BETWEEN ... AND ...`

Let's step through an example. To obtain information about the coverage in effect for Policy 1111 on Dec. 1, 2010, you could write the following query:

```
SELECT coverage_amt
FROM policy FOR BUSINESS_TIME AS OF '2010-12-01'
WHERE id = 1111;
```

DB2 returns a result of 500000.

To determine the terms applicable to Policy 1414 from Jan. 1, 2009 until Jan. 1, 2011, you could write the following query:

```
SELECT *
FROM policy FOR BUSINESS_TIME FROM '2009-01-01' TO '2011-01-01'
WHERE id = 1414;
```

DB2 returns two rows, as shown in Fig. 12.

Fig. 12: Query result

ID	VIN	annual_mileage	rental_car	coverage_amt	bus_start	bus_end
1414	B7777	14000	N	750000	2008-05-01	2010-03-01
1414	B7777	12000	N	600000	2010-03-01	2011-01-01

Temporal queries against tables with business time are internally re-written to a query with appropriate WHERE clause predicates on the appropriate date or timestamp columns defined for the start and end points of business time.

Bitemporal tables

As you’ve learned, DB2’s support for system and business time is straightforward. But DB2’s temporal data management capabilities don’t end there. Indeed, we mentioned earlier that DB2 enables you to maintain both system and business time in *bitemporal* tables. For example, you may decide to use business time to manage your application's logical notion of time, such as the validity periods of insurance policies, and also use system time to track the history and timestamps of changes that transactions make to these policies.

Administrators can easily create or alter a table to include both system and business time. For example, the following CREATE TABLE statement defines a bitemporal table with a BUSINESS_TIME period on the BUS_START and BUS_END columns as well as a SYSTEM_TIME period on the SYS_START and SYS_END columns.

```
CREATE TABLE policy (
  id               INT NOT NULL,
  vin              VARCHAR(10),
  annual_mileage   INT,
  rental_car       CHAR(1),
  coverage_amt     INT,
  bus_start        DATE NOT NULL,
  bus_end          DATE NOT NULL,
  sys_start        TIMESTAMP(12) GENERATED ALWAYS AS ROW BEGIN NOT NULL,
  sys_end          TIMESTAMP(12) GENERATED ALWAYS AS ROW END NOT NULL,
  trans_start      TIMESTAMP(12) GENERATED ALWAYS
                               AS TRANSACTION START ID IMPLICITLY HIDDEN,
  PERIOD SYSTEM_TIME (sys_start, sys_end),
  PERIOD BUSINESS_TIME(bus_start, bus_end),
  PRIMARY KEY(id, BUSINESS_TIME WITHOUT OVERLAPS)
);
```

After creating this bitemporal table, you need to create a compatible history table and enable versioning, as we discussed earlier in the section on system time. Then you can insert, update, delete, and query rows in this table using the syntax described earlier for system time and business time. We'll review a short example here.

Imagine that you have a bitemporal POLICY table and associated history table as shown in Fig. 13. (The TRANS_START column is omitted for simplicity.)

Fig. 13: Sample bitemporal POLICY and POLICY_HISTORY tables

POLICY table

ID	VIN	annual_ mileage	rental _car	coverage_amt	bus_start	bus_end	sys_start	sys_end

POLICY_HISTORY table (contains historical data)

ID	VIN	annual_ mileage	rental _car	coverage_amt	bus_start	bus_end	sys_start	sys_end

Imagine that client service representatives performed the following activities:

- On Nov. 15, 2011, someone created Policy 1111 for vehicle A1111. The policy, with a coverage amount of 500000, was set to go into effect on Jan. 1, 2012.

  ```
  INSERT INTO policy(id, vin, annual_mileage, rental_car,
                     coverage_amt, bus_start, bus_end)
  VALUES(1111,'A1111',10000,'Y',500000,'2012-01-01','9999-12-31');
  ```

- On March 1, 2012, someone changed the terms of Policy 1111 effective June 1, 2012. The change lowered the coverage amount and removed the rental car benefit. The following update statement was used:

```
UPDATE policy
  FOR PORTION OF BUSINESS_TIME FROM '2012-06-01' TO '9999-12-31'
SET coverage_amt = 250000, rental_car='N'
WHERE id = 1111;
```

Fig. 14 illustrates the contents of the POLICY and POLICY_HISTORY tables after these operations. (For simplicity, only the date portions of the timestamps for SYS_START and SYS_END are shown.)

Fig. 14: Effect of the UPDATE on the bitemporal table

POLICY table

ID	VIN	annual_mileage	rental_car	coverage_amt	bus_start	bus_end	sys_start	sys_end
1111	A1111	10000	Y	500000	2012-01-01	2012-06-01	2012-03-01	9999-12-31
1111	A1111	10000	N	250000	2012-06-01	9999-12-31	2012-03-01	9999-12-31

POLICY_HISTORY table (contains historical data)

ID	VIN	annual_mileage	rental_car	coverage_amt	bus_start	bus_end	sys_start	sys_end
1111	A1111	10000	Y	500000	2012-01-01	9999-12-31	2011-11-15	2012-03-01

Now imagine that a claim is filed against Policy 1111 for a car accident that occurred on June 20, 2012. A client service representative can determine coverage eligibility with the following query:

```
SELECT vin, rental_car, coverage_amt
FROM policy FOR BUSINESS_TIME AS OF '2012-06-20'
WHERE id = 1111;
```

DB2 will return the information shown in Fig. 15, which indicates that the vehicle is covered but that the client isn't eligible for a rental car.

Fig. 15: Query results

VIN	rental_car	coverage_amt
A1111	N	250000

If the client calls on July 10, 2012 to complain, demanding a full account of changes made to his policy for the past two years, a service representative can issue this query:

```
SELECT id, vin, annual_mileage, rental_car, coverage_amt,
       bus_start, bus_end, sys_start, sys_end
FROM policy FOR SYSTEM_TIME FROM TIMESTAMP('2010-07-10')
                              TO TIMESTAMP('2012-07-11')
WHERE id = 1111;
```

DB2 will return the results shown in Fig. 16. The client service representative can explain when changes occurred to the policy as well as when these changes were in effect (or set to take effect).

Fig. 16: Query results (the final row, shown in gray, was from the history table)

ID	VIN	annual_mileage	rental_car	coverage_amt	bus_start	bus_end	sys_start	sys_end
1111	A1111	10000	Y	500000	2012-01-01	2012-06-01	2012-03-01	9999-12-31
1111	A1111	10000	N	250000	2012-06-01	9999-12-31	2012-03-01	9999-12-31
1111	A1111	10000	Y	500000	2012-01-01	9999-12-31	2011-11-15	2012-03-01

Migration to DB2 temporal tables

The temporal support in DB2 has been designed to allow easy migration of existing database tables to the new temporal capabilities. There are two common scenarios to consider.

First, if you have existing tables without timestamp columns and you want to turn these into temporal tables with system time or business time periods, you can use ALTER TABLE statements to add the required timestamp columns and period definition to the table. For a system period temporal table you would then also create a history table "like" your original table and use another ALTER TABLE statement to enable versioning.

The second scenario involves migrating tables that already have timestamp columns. For example, you might already be using triggers to set timestamp columns and populate a history table. In that case you can reuse the existing timestamp columns and history table. You simply perform ALTER TABLE statements to declare that the existing timestamp columns are now interpreted as a SYSTEM_TIME period. You can then drop your triggers and issue another ALTER TABLE statement to enable versioning between your base table and history table. Similar migration options exist for business time as well.

Conclusion

DB2's new temporal data support provides simple yet sophisticated capabilities for managing multiple versions of your data and tracking effective business dates. Through simple SQL enhancements (based on emerging ANSI/ISO standards efforts), DB2 enables database professionals to work with temporal data in an efficient manner, saving considerable time and effort when compared with hard-coding temporal logic into triggers, stored procedures, or homegrown applications.

About the authors

Cynthia M. Saracco is a senior solutions architect at IBM's Silicon Valley Laboratory who specializes in emerging technologies and database management topics. She has 23 years of software industry experience, has written 3 books and more than 70 technical papers, and holds 7 patents.

Matthias Nicola is senior software engineer for DB2 at IBM's Silicon Valley Lab. His work focuses on XML, temporal data management, data warehousing, and performance. Matthias also works closely with customers and business partners, assisting them in the design, implementation, and optimization of DB2 solutions. Matthias has published more than 40 articles on various database management topics and is a frequent speaker at DB2 conferences. Prior to joining IBM, Matthias worked on data warehousing performance for Informix Software.

Lenisha Gandhi is a senior software development manager who manages a DB2 software engineering organization at IBM's Silicon Valley Lab. Lenisha focuses on temporal data management technology in DB2 for Linux, UNIX, and Windows (LUW). Prior to joining DB2, Lenisha worked in query development for the IBM Content Management organization.

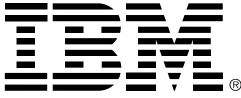
Acknowledgments

Thanks to those who assisted with this paper. In alphabetical order, this includes Xiaohong Fu, Shruti Gopinath, Bala Iyer, Krishna Kulkarni, Eileen Lin, Claire McFeely, and Martin Sommerlandt.

Resources

If you have questions or comments about DB2's temporal technology, please post them to the DB2 Temporal forum on IBM developerWorks. (You'll find the DB2 Temporal forum by visiting http://www.ibm.com/developerworks/forums/im_forums.jspa, which lists all the forums related to IBM Information Management.)

Learn more about DB2 10 z/OS at IBM's web site:
http://www.ibm.com/software/data/db2/zos/



© Copyright IBM Corporation, 2010
IBM
Armonk, NY
USA

All Rights Reserved.

Neither this documentation nor any part of it may be copied or reproduced in any form or by any means or translated into another language, without the prior consent of the IBM Corporation.

DB2, DB2 Universal Database, IBM, and the IBM logo are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries or both.

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

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.

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 measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

The information contained in this document is subject to change without any notice. IBM reserves the right to make any such changes without obligation to notify any person of such revision or changes. IBM makes no commitment to keep the information contained herein up to date.

References in this publication to IBM products or
services do not imply that IBM intends to make them
available in all countries in which IBM operates.