New IBM® DB2® temporal data management technology enables companies 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, companies can use simple SQL statements to instruct DB2 10 to automatically manage multiple versions of their data, as well as track effective dates for changing business conditions. This article introduces you to the key concepts and capabilities of DB2 temporal data management support on z/OS® and Linux®, UNIX®, and Windows®. It explains when this technology can be useful and provides examples to help you understand how easy it can be to deploy it.

Introduction

If "time is money," then DB2 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 minimizes or eliminates such efforts through the introduction of table design options, and query syntax and semantics, based on the ANSI/ISO SQL:2011 standard.

When is temporal data management needed?

Before delving into the technical aspects of DB2 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 five 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 agency's resolution of a claim involving a car accident. The agency 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 situation, time is critical. The new temporal data management support in DB2 helps companies implement time-aware applications and queries with minimal effort, as we'll discuss. However, before exploring the new capabilities, we will introduce some 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 one table; such tables would be considered bitemporal.

How do these concepts apply to the previous application scenarios? Scenarios 1 and 2 require knowledge of system time because understanding the historical state(s) of one or more tables is important. Scenarios 3-6 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 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 31 Dec 2007, the policy would have been active until midnight, but not at midnight.

IBM has worked with the ANSI and ISO SQL standard committees to incorporate these extensions into the latest SQL:2011 standard. IBM is the first database vendor to support temporal data management based on this new SQL standard. Other database vendors use proprietary syntax for temporal operations and for the definition of temporal tables.
Benefits of temporal data management

The built-in support in DB2 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 companies 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.

An IBM internal study compared implementing a subset of business time capabilities with built-in temporal data management support in DB2 vs. hand-coding equivalent logic in two homegrown implementations. One implementation used triggers and stored procedures, while the second used triggers and Java™ application logic. All tests were done with DB2 for Linux, UNIX, and Windows. The cost savings provided by the DB2 built-in support were striking.

The built-in support in DB2 reduced coding requirements by more than 90 percent over both homegrown implementations. Implementing just the core business time logic in SQL stored procedures or in Java technology required 16 times and 45 times as many lines of code, respectively, as the equivalent simple SQL statements that use the new DB2 temporal features (see Figure 1). Furthermore, it took less than an hour to develop and test these few DB2 SQL statements. By contrast, the homegrown approaches required four to five weeks to code and test, and both provided only a subset of the temporal data management support built into DB2. Indeed, the homegrown logic didn't address system time, bitemporal data, views, and many other DB2 features. Thus, providing truly equivalent temporal data support through a homegrown implementation would likely take months. This is much longer than it would take to simply write the appropriate DB2 SQL statements, which we'll discuss shortly.

Figure 1. Development cost for implementing business time behavior

Sample scenario

To help you understand the temporal data management support in DB2, we use a common application scenario and sample data. 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), estimated annual vehicle mileage, whether a rental car will be provided if the car needs repair as a result of a claim, and the coverage amount (including accidental property damage, medical expenses, etc).

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

Table 1. Sample POLICY table (without temporal support)

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>500000</td>
</tr>
</tbody>
</table>

Let's explore how DB2 temporal support can help you manage such insurance policies.

Managing data versions with system time

The support in DB2 for system time enables you to automatically track and manage multiple versions of your data. 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 DB2 will automatically generate these values on INSERT, UPDATE, and DELETE. This relieves you of specifying values for these columns when writing to the database and ensures that the timestamps are accurate. Optionally, you may define these columns as IMPLICITLY HIDDEN so 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.)
A matter of time: Temporal data management in DB2 10

Listing 1. Creating a table with a SYSTEM_TIME period

```sql
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

Listing 2.

```sql
CREATE TABLE policy_history LIKE policy;
```

Step 3: Enable versioning

Listing 3.

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

Table 2 and Table 3 illustrate 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.

Table 2. POLICY table (contains current data)

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
<th>sys_start</th>
<th>sys_end</th>
<th>trans_start</th>
</tr>
</thead>
</table>

Table 3. POLICY_HISTORY table (contains historical data)

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
<th>sys_start</th>
<th>sys_end</th>
<th>trans_start</th>
</tr>
</thead>
</table>

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.

DB2 for Linux, UNIX, and Windows supports automatic schema evolution across a user table and the associated history table. For example, if you add a column to the POLICY table, DB2 will also add the same column to the POLICY_HISTORY table.

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 15 Nov 2010 you needed to enter two new car insurance policy records into your POLICY table. The following statements accomplish this:
Listing 4. Inserting data into a table with system time

```
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 system time columns and transaction start time. Note that none of these were referenced in the `INSERT` statements; DB2 automatically records the necessary information. Table 4 and Table 5 depict the contents of the POLICY and POLICY_HISTORY tables as a result of this query. (To make our example easier to follow, Table 4 and Table 5 show 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 Table 4 and Table 5 as well.)

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
<th>sys_start</th>
<th>sys_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>500000</td>
<td>2010-11-15</td>
<td>9999-12-30</td>
</tr>
<tr>
<td>1414</td>
<td>B7777</td>
<td>14000</td>
<td>N</td>
<td>750000</td>
<td>2010-11-15</td>
<td>9999-12-30</td>
</tr>
</tbody>
</table>

The `SYS_START` values in the POLICY table reflect when the rows were inserted (on 15 Nov 2010 in our example). The `SYSTEM_END` values are set to 30 Dec 9999 to indicate that these rows have not expired (i.e., 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 31 Jan 2011 to change the coverage amount for Policy 1111 to 750000:

Listing 5. Updating data in a table with system time

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

Let's explore how DB2 processes this statement. As shown in Table 6 and Table 7, 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 and transparently to the user. (Although not shown in Table 6 and Table 7, DB2 also records the transaction start time in both tables.)
In effect, the processing by DB2 of this UPDATE statement records that Policy 1111 had a coverage amount of 500000 set from 15 Nov 2010 to 31 Jan 2011; thereafter, the coverage amount was set to 750000.

Table 6. Current table contents after UPDATE on 31 Jan 2011

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
<th>sys_start</th>
<th>sys_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>750000</td>
<td>2011-01-31</td>
<td>9999-12-30</td>
</tr>
<tr>
<td>1414</td>
<td>B7777</td>
<td>14000</td>
<td>N</td>
<td>750000</td>
<td>2010-11-15</td>
<td>9999-12-30</td>
</tr>
</tbody>
</table>

Table 7. History table contents after UPDATE on 31 Jan 2011

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
<th>sys_start</th>
<th>sys_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>500000</td>
<td>2010-11-15</td>
<td>2011-01-31</td>
</tr>
</tbody>
</table>

As you might expect, any subsequent updates to policies are handled in a similar manner. For example, assume that Policy 1111 is updated on 31 Jan 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:

Listing 6. Subsequent updates

```
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 Table 8 and Table 9.

Table 8. Current table contents after UPDATE on 31 Jan 2012

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
<th>sys_start</th>
<th>sys_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>A1111</td>
<td>5000</td>
<td>N</td>
<td>250000</td>
<td>2012-01-31</td>
<td>9999-12-30</td>
</tr>
<tr>
<td>1414</td>
<td>B7777</td>
<td>14000</td>
<td>N</td>
<td>750000</td>
<td>2010-11-15</td>
<td>9999-12-30</td>
</tr>
</tbody>
</table>

Table 9. History table contents after UPDATE on 31 Jan 2012

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
<th>sys_start</th>
<th>sys_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>500000</td>
<td>2010-11-15</td>
<td>2011-01-31</td>
</tr>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>750000</td>
<td>2011-01-31</td>
<td>2012-01-31</td>
</tr>
</tbody>
</table>

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., old data versions) through queries that contain an appropriate time period specification.

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

**Listing 7.**
```
DELETE FROM policy WHERE id = 1414;
```

As shown in Table 10 and Table 11, 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 (31 March 2012).

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
<th><code>sys_start</code></th>
<th><code>sys_end</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>A1111</td>
<td>5000</td>
<td>N</td>
<td>250000</td>
<td>2012-01-31</td>
<td>9999-12-30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
<th><code>sys_start</code></th>
<th><code>sys_end</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>10000</td>
<td>Y</td>
<td>500000</td>
<td>2010-11-15</td>
<td>2011-01-31</td>
</tr>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>750000</td>
<td>2011-01-31</td>
<td>2012-01-31</td>
</tr>
<tr>
<td>1414</td>
<td>B7777</td>
<td>14000</td>
<td>N</td>
<td>750000</td>
<td>2010-11-15</td>
<td>2012-03-31</td>
</tr>
</tbody>
</table>

**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 affected by adding system time support to 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. 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 Table 10 and Table 11. Consider the following query:

**Listing 8.**
```
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 ...** enables you to query data as of a certain point in time.
• **FOR SYSTEM_TIME FROM ... TO ...** enables you to query 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 ...** enables you to query 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 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 1 Dec 2010, you could write the following query:

**Listing 9.**

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

To resolve this query, DB2 transparently accesses data in the history table to retrieve the correct information (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 30 Nov 2011, you could write the following query:

**Listing 10.**

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

Given our sample data, this query returns the value of 2. As shown in Table 10 and Table 11, 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 mentioned, 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 another month. Or a credit card may have an interest rate of 16 percent one year and 18 percent 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.

**Listing 11. Creating a table with business time**

```sql
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 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:

**Listing 12. Inserting data into a table with business time**

```sql
INSERT INTO policy VALUES(1111,'A1111',10000,'Y',500000,'2010-01-01','2011-01-01');
INSERT INTO policy VALUES(1111,'A1111',14000,'N',750000,'2008-05-01','2010-03-01');
INSERT INTO policy VALUES(1111,'A1111',12000,'N',600000,'2010-03-01','2011-01-01');
```
Table 12 illustrates the resulting contents of this table.

### Table 12. POLICY table after INSERT statements

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
<th>bus_start</th>
<th>bus_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>500000</td>
<td>2010-01-01</td>
<td>2011-01-01</td>
</tr>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>750000</td>
<td>2011-01-01</td>
<td>9999-12-30</td>
</tr>
<tr>
<td>1414</td>
<td>B7777</td>
<td>14000</td>
<td>N</td>
<td>750000</td>
<td>2008-05-01</td>
<td>2010-03-01</td>
</tr>
<tr>
<td>1414</td>
<td>B7777</td>
<td>12000</td>
<td>N</td>
<td>600000</td>
<td>2010-03-01</td>
<td>2011-01-01</td>
</tr>
</tbody>
</table>

It may help to summarize the contents of this table in business terms. The data shows that Policy 1111 had a coverage amount of 500000 in effect from 1 Jan 2010 to 1 Jan 2011. From 1 Jan 2011 onward, a coverage amount of 750000 is in effect. Similarly, the table shows that from 1 May 2008 to 1 Mar 2010, Policy 1414 had coverage of 750000 for a specific vehicle with an estimated annual mileage of 14000. Effective 1 Mar 2010 to 1 Jan 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 effect of the temporal uniqueness constraint defined on this table (with the `BUSINESS_TIME WITHOUT OVERLAPS` clause). Imagine that we issued the following INSERT statement:

**Listing 13.**

```sql
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 1 Jun 2010 to 1 Sep 2011, we would accomplish this with an appropriate UPDATE statement.

### Updating data in a table with business time

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 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 1 Jun 2010 to 1 Sep 2011 — specifically, you want to alter the coverage amount for that period of time. Here's how you could write the UPDATE statement:

**Listing 14.**

```sql
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 Figure 2, there were originally two rows for Policy 1111. Both are affected by our UPDATE statement because the portion of business time being updated overlaps partially with the business period of each row. This overlap is illustrated in the upper part of Figure 2. When DB2 applies the update, each of the two original rows is split into two rows, as illustrated in the lower part of Figure 2. DB2 adjusts the effective dates of the rows automatically.

**Figure 2. Row splits caused by the UPDATE statement**

Before the update (Table 12)

![Diagram showing row splits before update]

After the update (Table 13)

![Diagram showing row splits after update]

Table 13 shows the resulting state of the POLICY table. The first row from Table 12 gets split into two new rows, shown in Table 13. The second row from Table 12 also gets split into two new rows, shown in Table 13.

**Table 13. POLICY table after Policy 1111 UPDATE**

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
<th>bus_start</th>
<th>bus_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>500000</td>
<td>2010-01-01</td>
<td>2010-06-01</td>
</tr>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>900000</td>
<td>2010-06-01</td>
<td>2011-01-01</td>
</tr>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>900000</td>
<td>2011-01-01</td>
<td>2011-09-01</td>
</tr>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>750000</td>
<td>2011-09-01</td>
<td>9999-12-30</td>
</tr>
<tr>
<td>1414</td>
<td>B7777</td>
<td>14000</td>
<td>N</td>
<td>750000</td>
<td>2008-05-01</td>
<td>2010-03-01</td>
</tr>
<tr>
<td>1414</td>
<td>B7777</td>
<td>12000</td>
<td>N</td>
<td>600000</td>
<td>2010-03-01</td>
<td>2011-01-01</td>
</tr>
</tbody>
</table>

**Deleting data from a table with business time**

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 1 Jun 2010 to 1 Jan 2011. Assuming the client is referring to Policy 1414, the following `DELETE` statement will accomplish this:

**Listing 15.**

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

Table 14 illustrates the resulting contents of the table. Note that DB2 altered the final row to reflect the new BUS_END date for Policy 1414.

**Table 14. POLICY table after DELETE involving a portion of business time**

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
<th>bus_start</th>
<th>bus_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>500000</td>
<td>2010-01-01</td>
<td>2010-06-01</td>
</tr>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>900000</td>
<td>2010-06-01</td>
<td>2011-01-01</td>
</tr>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>900000</td>
<td>2011-01-01</td>
<td>2011-09-01</td>
</tr>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>750000</td>
<td>2011-09-01</td>
<td>9999-12-30</td>
</tr>
<tr>
<td>1414</td>
<td>B7777</td>
<td>14000</td>
<td>N</td>
<td>750000</td>
<td>2008-05-01</td>
<td>2010-03-01</td>
</tr>
<tr>
<td>1414</td>
<td>B7777</td>
<td>12000</td>
<td>N</td>
<td>600000</td>
<td>2010-03-01</td>
<td>2010-06-01</td>
</tr>
</tbody>
</table>

**Querying a table with business time**

Querying a table with business time is simple. Three optional clauses enable you to specify temporal queries so 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 Table 15. (This is the same data shown in Table 12, immediately after we created the POLICY table and inserted four rows.)

**Table 15. POLICY table after DELETE involving a portion of business time**

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
<th>bus_start</th>
<th>bus_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>500000</td>
<td>2010-01-01</td>
<td>2011-01-01</td>
</tr>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>750000</td>
<td>2011-01-01</td>
<td>9999-12-30</td>
</tr>
<tr>
<td>1414</td>
<td>B7777</td>
<td>14000</td>
<td>N</td>
<td>750000</td>
<td>2008-05-01</td>
<td>2010-03-01</td>
</tr>
<tr>
<td>1414</td>
<td>B7777</td>
<td>12000</td>
<td>N</td>
<td>600000</td>
<td>2010-03-01</td>
<td>2011-01-01</td>
</tr>
</tbody>
</table>
To determine the total number of insurance records you have for Policy 1111, you could write the following query:

**Listing 16.**

```sql
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 1 Dec 2010, you could write the following query:

**Listing 17.**

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

DB2 returns a result of 500000.

To identify the terms applicable to Policy 1414 from 1 Jan 2009 to 1 Jan 2011, you could write the following SQL statement, which produces the result shown in **Table 16**.

**Listing 18.**

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

**Table 16. Query result**

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
<th>bus_start</th>
<th>bus_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>1414</td>
<td>B7777</td>
<td>14000</td>
<td>N</td>
<td>750000</td>
<td>2008-05-01</td>
<td>2010-03-01</td>
</tr>
<tr>
<td>1414</td>
<td>B7777</td>
<td>12000</td>
<td>N</td>
<td>600000</td>
<td>2010-03-01</td>
<td>2011-01-01</td>
</tr>
</tbody>
</table>

Temporal queries against tables with business time are internally rewritten 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.
Advanced concepts

As you've learned, DB2 support for system and business time is straightforward. But the DB2 temporal data management capabilities don't end there. We mentioned that DB2 enables you to maintain system and business time in bitemporal tables. Furthermore, DB2 on Linux, UNIX, and Windows supports temporal views and a register setting that allows your database to function much like a time machine. While it's beyond the scope of this article to provide a full discussion of these and other advanced temporal topics, we'll describe each briefly.

Bitemporal tables

Bitemporal tables allow you to manage data with system time and business time simultaneously, combining the benefits of both. 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.

Listing 19. Creating a bitemporal table

```sql
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. 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 Table 17 and Table 18. (The TRANS_START column is omitted for simplicity.)

Table 17. Sample bitemporal POLICY table

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
<th>bus_start</th>
<th>bus_end</th>
<th>sys_start</th>
<th>sys_end</th>
</tr>
</thead>
</table>

Imagine that client service representatives performed the following activities:

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

```sql
Listing 20.

INSERT INTO policy(id, vin, annual_mileage, rental_car, coverage_amt, bus_start, bus_end)
VALUES(1111, 'A1111', 10000, 'Y', 500000, '2012-01-01', '9999-12-30');
```

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

```sql
Listing 21.

UPDATE policy
FOR PORTION OF BUSINESS_TIME FROM '2012-06-01' TO '9999-12-30'
SET coverage_amt = 250000, rental_car='N'
WHERE id = 1111;
```

Table 19 and Table 20 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.)

### Table 19. Effect of UPDATE on the bitemporal POLICY table

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
<th>bus_start</th>
<th>bus_end</th>
<th>sys_start</th>
<th>sys_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>500000</td>
<td>2012-01-01</td>
<td>2012-06-01</td>
<td>2012-03-01</td>
<td>9999-12-30</td>
</tr>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>N</td>
<td>250000</td>
<td>2012-06-01</td>
<td>9999-12-30</td>
<td>2012-03-01</td>
<td>9999-12-30</td>
</tr>
</tbody>
</table>

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

### Listing 22.

```sql
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 Table 21, which indicates that the vehicle is covered, but that the client isn't eligible for a rental car.
Table 21. Query results

<table>
<thead>
<tr>
<th>VIN</th>
<th>rental_car</th>
<th>coverage_amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1111</td>
<td>N</td>
<td>250000</td>
</tr>
</tbody>
</table>

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

Listing 23. Querying bitemporal data

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

DB2 will return the results shown in Table 22. The customer service representative can explain when changes occurred to the policy, as well as when these changes were in effect (or set to take effect).

Table 22. Query results (the final row comes from the history table)

<table>
<thead>
<tr>
<th>ID</th>
<th>VIN</th>
<th>annual_mileage</th>
<th>rental_car</th>
<th>coverage_amt</th>
<th>bus_start</th>
<th>bus_end</th>
<th>sys_start</th>
<th>sys_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>500000</td>
<td>2012-01-01</td>
<td>2012-06-01</td>
<td>2012-03-01</td>
<td>9999-12-30</td>
</tr>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>N</td>
<td>250000</td>
<td>2012-06-01</td>
<td>9999-12-30</td>
<td>2012-03-01</td>
<td>9999-12-30</td>
</tr>
<tr>
<td>1111</td>
<td>A1111</td>
<td>10000</td>
<td>Y</td>
<td>500000</td>
<td>2012-01-01</td>
<td>9999-12-30</td>
<td>2011-11-15</td>
<td>2012-03-01</td>
</tr>
</tbody>
</table>

Views

DB2 for Linux, UNIX, and Windows allows you to use two types of views with temporal tables. Such views provide flexibility for application design and allow you to expose data for different points or periods in time to different users.

First, you can define views on a temporal table where the view definition contains a `FOR SYSTEM_TIME` or `FOR BUSINESS_TIME` clause to restrict the view to a certain point or period in time. Then you can perform regular SQL queries against these views. A query against these views can't contain a `FOR SYSTEM_TIME` or `FOR BUSINESS_TIME` clause because the time constraint in the query might conflict with the time constraint in the view or lead to ambiguity.

Second, you can define views on a temporal table without a `FOR SYSTEM_TIME` or `FOR BUSINESS_TIME` constraint in the view definition. Such views expose data for all points in time and can be accessed with queries that include a `FOR SYSTEM_TIME` or `FOR BUSINESS_TIME` clause. Such a clause is then automatically applied to every table in the view definition that contains a `SYSTEM_TIME` or `BUSINESS_TIME` period, respectively.

In summary, DB2 offers you considerable flexibility when working with views and temporal data. You can use temporal constraints in a view definition or in queries against views over temporal tables.
Register settings for "time travel"

New register settings enable you to run existing applications against data from a certain point in time without changing the application itself. For example, imagine that you have an existing application that contains many SQL queries or a set of reporting queries you need to run from time to time. With the new temporal features in DB2, you may want to run those against a past point in system time, or against a past or future point in business time. However, enhancing all existing SQL statements with a `FOR SYSTEM_TIME` or `FOR BUSINESS_TIME` clause can be labor-intensive. Similarly, creating views for all affected temporal tables can also be tedious. Therefore, DB2 for Linux, UNIX, and Windows provides special registers to set a database session to a specific point in time.

For example, you could use the following `SET` commands to set the session's system time to 10 a.m. 1 Jan 2008:

**Listing 24.**

```
SET CURRENT TEMPORAL SYSTEM_TIME = '2008-01-01 10:00:00';
```

Subsequently, any queries against system period temporal tables (or bitemporal tables) performed in the same session will see data as of 10 a.m. 1 Jan 2008. That is, all queries in the session are internally rewritten to use the clause `FOR SYSTEM_TIME AS OF '2008-01-01 10:00:00'` that you saw earlier. DB2 performs this rewrite automatically for you; you don't need to change your application and SQL statements in any way.

Similarly, you could set the following special register to look at the data from one month ago:

**Listing 25.**

```
SET CURRENT TEMPORAL SYSTEM_TIME = current timestamp - 1 MONTH;
```

When the `CURRENT TEMPORAL SYSTEM_TIME` special register is set to a value other than NULL, any DML operations (insert, update, delete, merge) against system period temporal tables are not meaningful and are disallowed.

For business time, you can set the `CURRENT TEMPORAL BUSINESS_TIME` special register to apply queries, updates, deletes, etc. to past or future points in business time. For example, consider the following register setting:

**Listing 26.**

```
SET CURRENT TEMPORAL BUSINESS_TIME = '2012-06-01';
```

It means that queries and DML statements against tables with business time (or bitemporal tables) are rewritten to use the clause `FOR BUSINESS_TIME AS OF '2012-06-01'`. Again, DB2 performs this rewrite automatically for you.

Similar to the usage guideline for views, you must avoid applying two time constraints at the same time. That is, you either set a system time (business time) special register or you use system time
(business time) constraints your SQL statements, but not both at the same time. The time machine can accept only one destination at a time (no pun intended).

**Migration to DB2 temporal tables**

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

1. 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.

2. 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**

The new DB2 temporal data support provides simple yet sophisticated capabilities for managing multiple versions of your data and tracking effective business dates. Based on the temporal features in the SQL:2011 standard, 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.
Related topics

- In the DB2 area on developerWorks, get the resources you need to advance your DB2 skills.
- Now you can use DB2 for free. Download DB2 Express-C, a no-charge version of DB2 Express Edition for the community that offers the same core data features as DB2 Express Edition and provides a solid base to build and deploy applications.

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