Temporal Information Systems

SS 2015

Temporal Perspectives for SQL

Chapter 6

Beyond "Classical" SQL

> "The case studies in this book have amply demonstrated that SQL-92 does not look favorably upon time-oriented applications. Even the most simple tasks, such as specifying a primary key or joining two tables, become mired in complexity when time is introduced.
>
> Fortunately, the clouds part at the horizon. A minor language extension proposed for SQL3 dramatically simplifies coding such applications by providing support for periods, valid time, and transaction time."

(R. Snodgrass in „Developing Time-Oriented ...“ Chap. 12)

When Snodgrass wrote these lines (in 1998), there was a lot of optimism among a group of researchers headed by him who had submitted a detailed extension proposal to the SQL standardization bodies at ANSI and ISO based on their language TSQL2.

However, the proposal (called SQL/Temporal) never made it into the standard, mainly due to „disagreements“ within the ISO Committee in 2001, as Snodgrass tells us on a webpage he has devoted to these efforts:

http://www.cs.arizona.edu/people/rts/sql3.html

SQL/Temporal

- Nevertheless, the ideas developed for the SQL/Temporal proposal are well designed and meanwhile had a lot of influence on the development of relational DBMS on the market. Therefore, we summarize the main ideas in this (final) chapter of the lecture.

- There is a lot of additional literature on these extensions available via the webpage mentioned on the previous slide (including the proposals to ANSI/ISO). The book by Snodgrass discusses these issues in chapter 12. Please, help yourselves and read, if interested!

- This page also contains links to commercial DBMS offering limited and proprietary versions of the TSQL concepts, e.g.,
    - IBM DB2 10 for z/OS,
    - Oracle 11g as part of its Workspace Manager and using the Flashback Archive
    - Teradata 13.10

- A very interesting Java frontend to Oracle (using JDBC) is TimeDB still available on the web (developed by an academic group in Zurich):

http://www.timeconsult.com/Software/Software.html

PERIOD Data Type in SQL/Temporal (1)

In the SQL/Temporal proposal, the following characteristics of PERIOD have been chosen:

- Three data type variants are available:

  PERIOD(DATE), PERIOD(TIME), and PERIOD(TIMESTAMP)

- Four variants of PERIOD literals can be used, where [ ] indicates closed, and ( ) open time intervals. All combinations are possible: [ ), [ ], ( ], ( ).
  An example literal of type PERIOD(DATE) is

  PERIOD ' [2011-05-09 – 2011-05-14)'

- Period predicates are as follows in SQL/Temporal:
  - OVERLAPS is applicable to pairs of PERIOD values, too. It is equivalent to the following condition in terms of Allen operators:

    p overlaps q $\vee$ p overlaps^{-1} q $\vee$ p starts q $\vee$ p starts^{-1} q $\vee$ p finishes q $\vee$ p finishes^{-1} q $\vee$ p during q $\vee$ p during^{-1} q $\vee$ p equals q.

  - PRECEDES/SUCCEEDS stand for *before* and *before*$^{-1}$, resp.
  - p MEETS q implements *p meets q* $\vee$ *p meets*$^{-1}$ *q*
  - p CONTAINS q is short for *p during q* $\wedge$ $\neg$ *(p equals q)*

PERIOD Data Type in SQL/Temporal (2)

In addition, there is a number of PERIOD constructors available in SQL/Temporal:

(For details, see the Snodgrass book and Chap. 12)

Datetime Constructors:	
beginning(p)	BEGIN(p)
previous(p)	PRIOR(BEGIN(p))
last(p)	LAST(p)
ending(p)	END(p)
Interval Constructors:	
duration(p)	INTERVAL(p), INTERVAL(p AS *qual*)
extract_time_zone(p)	CAST(EXTRACT(TIMEZONE_HOUR FROM BEGIN(p)) AS HOUR) + CAST(EXTRACT(TIMEZONE_MINUTE FROM BEGIN(p)) AS MINUTE)
Period Constructors:	
p + i	PERIOD[BEGIN(p) + i , END(p) + i)
i + p	PERIOD[BEGIN(p) + i , END(p) + i)
p - i	PERIOD[BEGIN(p) - i , END(p) - i)
p *extend* q	not possible
p ∩ q	p P_INTERSECT q
p - q	p P_EXCEPT q
p ∪ q	p P_UNION q
p AT TIME ZONE i	PERIOD[BEGIN(p) AT TIME ZONE i , END(p) AT TIME ZONE i)
p AT LOCAL	PERIOD[BEGIN(p) AT LOCAL, END(p) AT LOCAL)
Other Operators:	
CAST(a AS PERIOD)	PERIOD[a , a]
CAST(p AS CHAR)	CAST(p AS CHAR)

(from: R. Snodgrass „Developing Time-Oriented ...“, p. 407)

PERIOD Data Type in SQL/Temporal (3)

(SQL3 used to be the name for that SQL standard intended to include SQL/Temporal.)

Table 12.1 Period operations in SQL3.

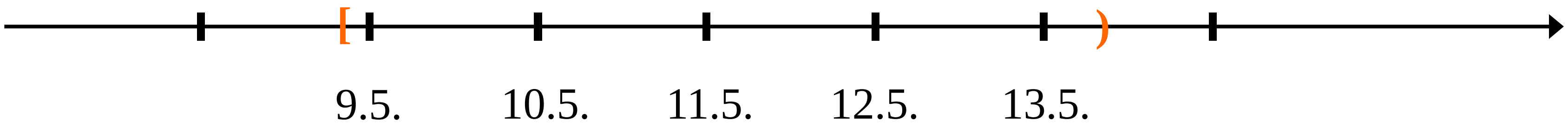
Period Operations	SQL3 Equivalent
Types:	
period	PERIOD(*datetime type*)
Predicates:	
p *equals* q	$p = q$
p *before* q	p PRECEDES q
p *before*$^{-1}$ q	p SUCCEEDS q
p *meets* q	END(p) = BEGIN(q)
p *meets*$^{-1}$ q	END(q) = BEGIN(p)
p *overlaps* q	BEGIN(p) < BEGIN(q) AND BEGIN(q) < END(p)
p *overlaps*$^{-1}$ q	BEGIN(q) < BEGIN(p) AND BEGIN(p) < END(q)
p *during* q	BEGIN(q) < BEGIN(p) AND END(p) < END(q)
p *during*$^{-1}$ q	BEGIN(p) < BEGIN(q) AND END(q) < END(p)
p *starts* q	BEGIN(p) = BEGIN(q) AND END(p) < END(q)
p *starts*$^{-1}$ q_2	BEGIN(p) = BEGIN(q) AND END(q) < END(p)
p *finishes* q	BEGIN(q) < BEGIN(p) AND END(p) = END(q)
p *finishes*$^{-1}$ q	BEGIN(p) < BEGIN(q) AND END(p) = END(q)
p OVERLAPS q	p OVERLAPS q
p IS NULL	p IS NULL

(from: R. Snodgrass „Developing Time-Oriented ...“, p. 407)

PERIOD Data Type in SQL/Temporal (4)

Each period corresponds to a set of instants:

PERIOD '[2011-05-09 – 2011-05-14)'



{2011-05-09 ,2011-05-10 ,2011-05-11 ,2011-05-12 ,2011-05-13}

The set is contiguous, i.e., for every two instants x and y in the set such that $x < y$ every other instant z such that $x < z < y$ is in the set, too (wrt the resp. granularity).

PERIOD Data Type in SQL/Temporal (5)

As periods are sets of instants, it makes sense to use set operators on periods as well. However, as periods are required to be contiguous („no gaps“), not every set of instants is in turn a period. Thus, there is a need for restrictions on usage of temporal set operators, in other words: Periods are not „closed“ under set operators!

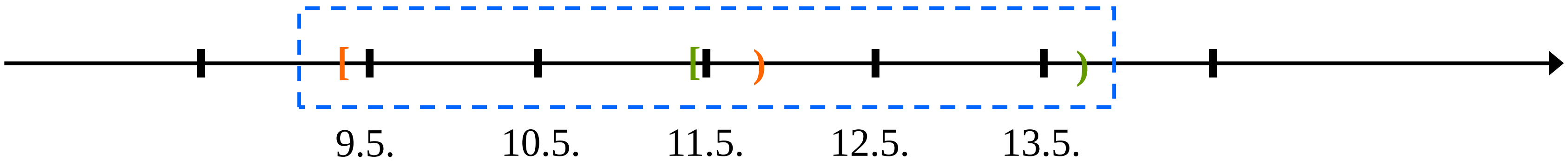
SQL/Temporal expresses temporal set operators as P-variants of their relational counter-parts in order to avoid overloading them.

Periods which either OVERLAPS or MEETS can be „combined“ into a single new period which corresponds to the union of the sets of instants contained in these periods, e.g.:

PERIOD ' [2011-05-09 – 2011-05-12) '

PERIOD ' [2011-05-11 – 2011-05-14)'

P_UNION



{2011-05-09 ,2011-05-10 ,2011-05-11} ∪ 2011-05-11,2011-05-12 ,2011-05-13}

{2011-05-09 ,2011-05-10 ,2011-05-11 ,2011-05-12 ,2011-05-13}

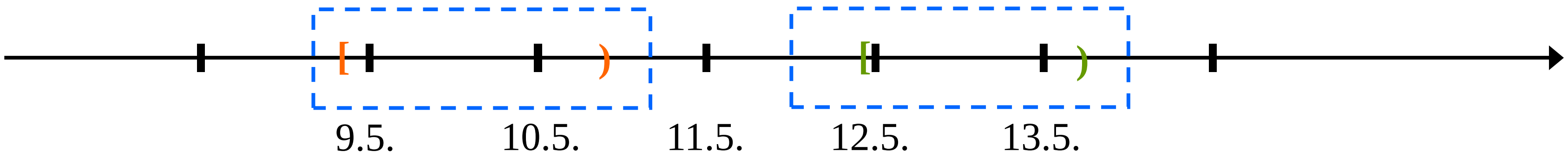
PERIOD Data Type in SQL/Temporal (6)

If any of the two operand periods of a union is BEFORE the other, the union set is no longer contiguous, and thus not a period any more – for such input, period P_UNION is undefined:

PERIOD ' [2011-05-09 – 2011-05-11) '

PERIOD ' [2011-05-12 – 2011-05-14) '

P_UNION



{2011-05-09 ,2011-05-10} ∪ {2011-05-12 ,2011-05-13}

{2011-05-09 ,2011-05-10 ,~~2011-05-11~~ ,2011-05-12 ,2011-05-13}

Similarly, the P_INTERSECT operator for periods is defined if and only if its two operands OVERLAPS, e.g.:

PERIOD ' [2011-05-09 – 2011-05-13)' **P_INTERSECT** PERIOD ' [2011-05-11 – 2011-05-14)'

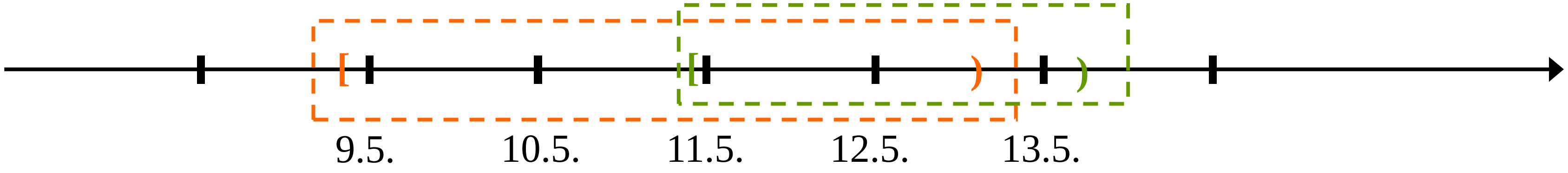
results in PERIOD ' [2011-05-11 – 2011-05-13)'

PERIOD Data Type in SQL/Temporal (7)

As in relational algebra, the difference operator is not commutative for periods, too:

PERIOD ' [2011-05-09 – 2011-05-13)' **P_EXCEPT** PERIOD ' [2011-05-11 – 2011-05-14)'

results in PERIOD ' [2011-05-09 – 2011-05-11)'



What is the necessary restriction in this case which ensures that the result of applying a P_EXCEPT to two periods is a period again?

SQL/Temporal: VALIDTIME and TRANSACTIONTIME Tables and Constraints

- The most important extension to „ordinary“ SQL wrt temporal support is to allow tables to be specified as VALIDTIME and/or TRANSACTIONTIME tables already at creation time. All timestamp columns are automatically created and maintained (i.e., hidden) and have PERIOD values, e.g., for a bitemporal table:

```
CREATE TABLE  student
   matrNr   integer,
   name   text,
   . . .
AS VALIDTIME PERIOD(DATE)
   AND TRANSACTIONTIME
```

(Granularity for TT timestamps is provided by the DBMS.)

- Keys, foreign keys and other constraints expressed in „normal“ SQL style are interpreted as current constraints in SQL/Temporal.

- Sequenced constraints can be declared using the prefix VALIDTIME and/or TRANSACTIONTIME, resp., e.g.:

```
CREATE TABLE  student
   matrNr   integer VALIDTIME PRIMARY KEY,
   . . .
```

(Again VALIDTIME AND TRANSACTION TIME in case of both dimensions.)

SQL/Temporal: VALIDTIME Queries (1)

For illustrating the extended querying capabilites of SQL/Temporal, we just discuss the case of valid time queries. Transaction time queries are quite analogous, for bi-temporal queries please consult the sources mentioned.

All queries directed to tables with temporal support but not mentioning any temporal dimension explicitly are considered current queries. This decision is characteristic of SQL/Temporal in order to ease introduction of timestamps during the lifetime of a database („temporal upward compatibility“).

Sequenced queries (wrt the valid time dimension) are very easily expressed by preceeding the non-temporal version with the keyword VALIDTIME. The following, e.g., is the SQL/Temporal equivalent of the complex temporal join discussed in chapter 3 of this lecture (involving four different cases):

Provide the salary and position history for all employees.

```
VALIDTIME
SELECT S.SSN, AMOUNT, PCN
FROM SAL_HISTORY AS S, INCUMBENTS
WHERE S.SSN = INCUMBENTS.SSN
```

SQL/Temporal: VALIDTIME Queries (2)

- Nonsequenced queries are expressed by prefixing VALIDTIME with the additional keyword NONSEQUENCED, e.g.:

List all the salaries, past and present, of employees who had been a hazardous waste specialist (20730) at some time.

```
NONSEQUENCED VALIDTIME
SELECT    AMOUNT
FROM      INCUMBENTS, POSITIONS, SAL_HISTORY
WHERE     INCUMBENTS.SSN = SAL HISTORY.SSN
   AND    INCUMBENTS.PCN = POSITIONS.PCN
   AND    JOB_TITLE_CODE = 20730
```

- A new temporal function VALIDTIME( ) is available for accessing the VALIDTIME timestamp of each row, e.g.:

When did employees receive salary raises?

```
NONSEQUENCED VALIDTIME
SELECT    S2.SSN, BEGIN(VALIDTIME(S2)) AS RAISE_DATE
FROM      SAL HISTORY AS S1, SAL HISTORY AS S2
WHERE     S2.AMOUNT > S1.AMOUNT
   AND    S1.SSN = S2.SSN
   AND    VALIDTIME(S1) MEETS VALIDTIME(S2)
```

- A timeslice query is expressed using this function in the WHERE part, e.g.:

OVERLAPS extended: Accepts instant literals as one of ist arguments, too!

```
. . WHERE  VALIDTIME(S) OVERLAPS DATE '2010-12-31'
```

SQL/Temporal: VALIDTIME Modifications

Analogously, modifications of VALIDTIME tables not mentioning timestamps are considered current modifications – a timestamp for new rows is automatically added at insertion (always PERIOD , [CURRENT_DATE, 9999-12-31) ').

Current deletions and updates of VALIDTIME tables can be expressed in their logical form, i.e, without worrying about transforming them into several physical changes in order to implement the „not forgetting anything idea“! The implementation of such updates performed by the DBMS, however, still is the physical one, but the transformation remains hidden from the user.

Sequenced modifications are expressed by preceding the resp. logical modification by the dimension keyword VALIDTIME followed by a period literal representing the period of applicability of the modification, e.g.:

Remove Bob as associate director of the Computer Center for all of 1997.

```
VALIDTIME PERIOD ' [1997-01-01 - 1997-12-31] '
DELETE FROM INCUMBENTS
WHERE   SSN=111223333
    AND   PCN = 999071
```

SQL/Temporal: VALIDTIME and TRANSACTIONTIME Modifications

- Nonsequenced modifications require a preceding NONSEQUENCED again, indicating that the timestamp columns are treated as „normal columns“ in the following change statement, e.g.:

Extend Bob's position as associate director of the Computer Center for an additional year.

```
NONSEQUENCED VALIDTIME
UPDATE    INCUMBENTS
SET       VALIDTIME = PERIOD [BEGIN(VALIDTIME(INCUMBENTS)),
                              (END(VALIDTIME(INCUMBENTS))
                               + INTERVAL 1 YEAR) )
WHERE     SSN = 111223333
   AND    PCN = 999071
```

For the transaction time dimension, sequenced and nonsequenced modifications are <u>not</u> allowed (in order to preserve „faithfulness“ of the logging nature of this dimension). Only current modifications are accepted, which are expressed in their logical form, proper maintenance of TT timestamps is taken care of by the DBMS. TT timestamps can be accessed using the function TRANSACTIONTIME(…), delivering a period.

Pro:	Complex and lengthy temporal queries and updates are completely avoided!
Con:	Using the compact keywords properly requires exact knowledge of the hidden semantics of each construct (in order to know what really happens)!

SQL/Temporal: Querying Bitemporal Tables

In case of a bitemporal table, it is particularly important to indicate the intended type of query in the prefix of each query statement, e.g.:

- Past TT time slice query:

  What was known in the DB about the owner of 7797 on 1.1.1998?

```
VALIDTIME AND NONSEQUENCED TRANSACTIONTIME
SELECT    customer
FROM      Owner
WHERE     property = 7797  AND TRANSACTIONTIME(Owner) OVERLAPS DATE '1998-01-01'
```

- Current TT time slice query (condition and type mainly implicit):

```
VALIDTIME   SELECT  customer  FROM  Owner  WHERE  property = 7797
```

- Past VT+TT time slice query:

  When was the information about the owner on ... entered into the DB?

```
NONSEQUENCED  VALIDTIME AND TRANSACTIONTIME
SELECT    customer
FROM      Owner
WHERE     property = 7797  AND VALIDTIME(Owner) OVERLAPS DATE '1998-01-04'
```

IBM

developerWorks.

A matter of time: Temporal data management in DB2 10

Cynthia M. Saracco
Senior Software Engineer
IBM

Matthias Nicola (mnicola@us.ibm.com)
Senior Technical Staff Member
IBM Silicon Valley Lab

Lenisha Gandhi (lenisha@us.ibm.com)
Senior Software Development Manager
IBM

Skill Level: Intermediate

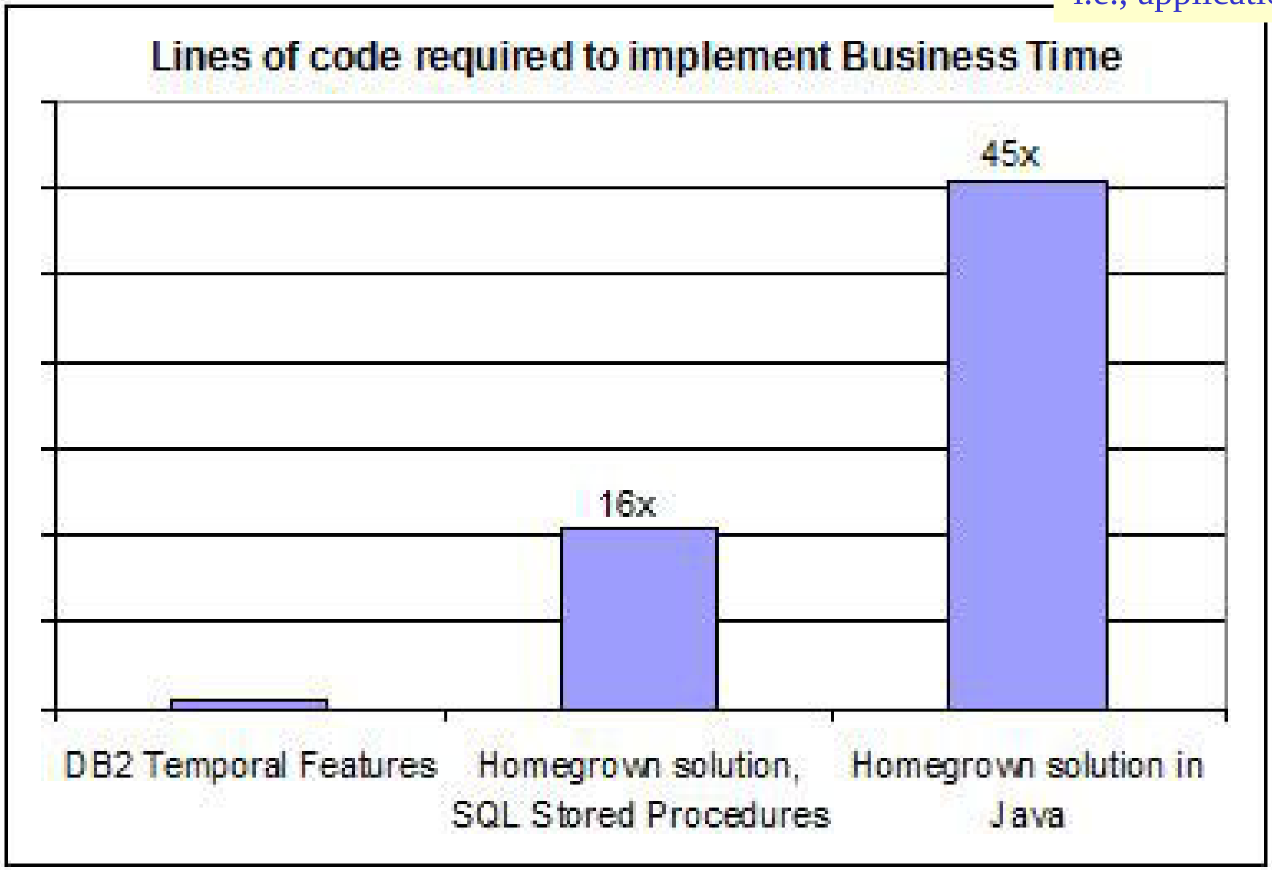
Date: 03 Apr 2012

„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."

http://www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/index.html

DB2 Temporal DB Support (1)

i.e., application/valid time



from Sarocco, Nicola, Gandhi: „A matter of time: Temporal data management in DB2 10“, IBM 2012

Sample scenario for the following slides:

Table for car insurance policies!

- id: Policy ID
- vin: Vehicle identification number
- annual_mileage: Estimated mileage of car
- rental_car: Rental car provided on repairs (Y/N)
- coverage_amount: Maximal damage amount covered by insurance

Table 1. Sample POLICY table (without temporal support)

ID	VIN	annual_mileage	rental_car	coverage_amt
1111	A1111	10000	Y	500000

DB2 Temporal DB Support (2)

Step 1: Create a table with a SYSTEM_TIME period

— (Our definition specifies that the TRANS_START column will be hidden.)

Listing 1. Creating a table with a SYSTEM_TIME period

current table

```
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)
 );
```

Timestamps automatically generated on every change.

Step 2: Create an associated history table

Listing 2.

history table

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

Step 3: Enable versioning

Listing 3.

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

from Sarocco, Nicola, Gandhi: „A matter of time: Temporal data management in DB2 10“, IBM 2012

DB2 Temporal DB Support (2a)

- System time: Details deviate from SQL:2011!

- Two tables rather than one, if system time is used:
    - current table
    - history table
    - Explicit creation of both tables is needed!
    - Separate step for linking them and switching automatic versioning on (rather than WITH SYSTEM VERSIONING clause)

- PERIOD SYSTEM_TIME rather than PERIOD FOR SYSTEM_TIME.

- Possibility to introduce additional system-generated timestamp using transaction start time of that transaction which contains the change command affecting this table.

DB2 Temporal DB Support (3)

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);
```

Table 4. Current table contents after `INSERT`s on 15 Nov 2010

automatically generated

POLICY						
ID	VIN	annual_mileage	rental_car	coverage_amt	sys_start	sys_end
1111	A1111	10000	Y	500000	2010-11-15	9999-12-30
1414	B7777	14000	N	750000	2010-11-15	9999-12-30

Table 5. History table contents after `INSERT`s on 15 Nov 2010

POLICY_HISTORY (empty)						
ID	VIN	annual_mileage	rental_car	coverage_amt	sys_start	sys_end

from Sarocco, Nicola, Gandhi: „A matter of time: Temporal data management in DB2 10“, IBM 2012

DB2 Temporal DB Support (4)

Listing 5. Updating data in a table with system time

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

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

POLICY						
ID	VIN	annual_mileage	rental_car	coverage_amt	sys_start	sys_end
1111	A1111	10000	Y	750000	2011-01-31	9999-12-30
1414	B7777	14000	N	750000	2010-11-15	9999-12-30

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

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

from Sarocco, Nicola, Gandhi: „A matter of time: Temporal data management in DB2 10“, IBM 2012

Listing 6. Subsequent updates

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

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

POLICY						
ID	VIN	annual_mileage	rental_car	coverage_amt	sys_start	sys_end trans_start
1111	A1111	5000	N	250000	2012-01-31	9999-12-30
1414	B7777	14000	N	750000	2010-11-15	9999-12-30

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

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

from Sarocco, Nicola, Gandhi: „A matter of time: Temporal data management in DB2 10“, IBM 2012

DB2 Temporal DB Support (6)

Listing 7.

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

Table 10. Current table contents after `DELETE` on 31 March 2012

POLICY						
ID	VIN	annual_mileage	rental_car	coverage_amt	sys_start	sys_end
1111	A1111	5000	N	250000	2012-01-31	9999-12-30

Table 11. History table contents after `DELETE` on 31 March 2012

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

from Sarocco, Nicola, Gandhi: „A matter of time: Temporal data management in DB2 10“, IBM 2012

DB2 Temporal DB Support (7)

Listing 8.

current query

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

answer: 250.000 (over current table on previous slide)

Listing 9.

past (timeslice) query

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

answer: 500.000 (over history table on previous slide)

Listing 10.

sequenced query

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

answer: 2 (over current <u>and</u> history table on previous slide)

(two variants: FROM … TO – [close, open) period, BETWEEN … AND – [close, close] period)

from Sarocco, Nicola, Gandhi: „A matter of time: Temporal data management in DB2 10“, IBM 2012

Listing 11. Creating a table with business time

```
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) );
```

- Different from standard:
  - Business time rather than application time
  - No user-defined period name, but PERIOD BUSINESS_TIME throughout.
- As in standard:
  - No separate history table.
  - Timestamp values to be supplied by users in insert statements.
  - Short form for temporal PK (and FK).

from Sarocco, Nicola, Gandhi: „A matter of time: Temporal data management in DB2 10“, IBM 2012

DB2 Temporal DB Support (9)

Listing 12. Inserting data into a table with business time

```
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','9999-12-30');
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');
```

Table 12. POLICY table after `INSERT` statements

POLICY						
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	9999-12-30
1414	B7777	14000	N	750000	2008-05-01	2010-03-01
1414	B7777	12000	N	600000	2010-03-01	2011-01-01

from Sarocco, Nicola, Gandhi: „A matter of time: Temporal data management in DB2 10“, IBM 2012

DB2 Temporal DB Support (10)

state before UPDATE:

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	9999-12-30
1414	B7777	14000	N	750000	2008-05-01	2010-03-01
1414	B7777	12000	N	600000	2010-03-01	2011-01-01

Listing 14.

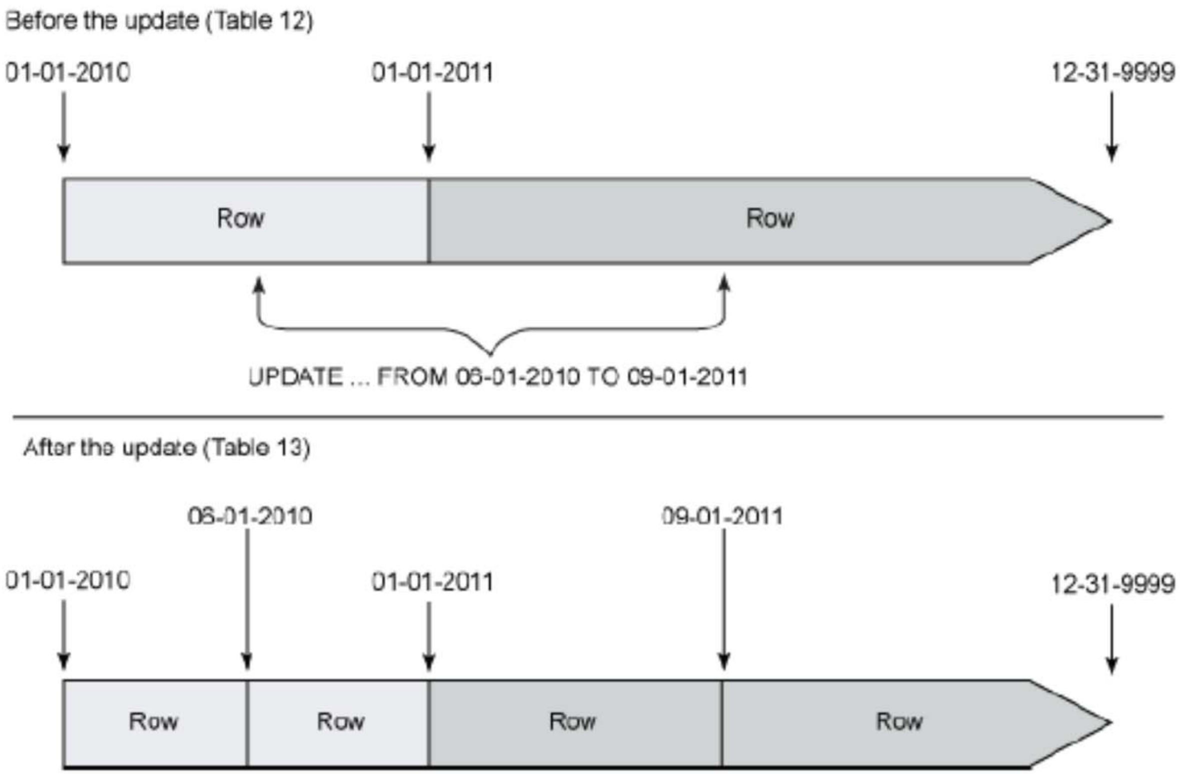
```
UPDATE policy
  FOR PORTION OF BUSINESS_TIME FROM '2010-06-01' TO '2011-09-01'
SET coverage_amt = 900000
WHERE id = 1111;
```

Table 13. POLICY table after Policy 1111 UPDATE

POLICY						
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	9999-12-30
1414	B7777	14000	N	750000	2008-05-01	2010-03-01
1414	B7777	12000	N	600000	2010-03-01	2011-01-01

from Sarocco, Nicola, Gandhi: „A matter of time: Temporal data management in DB2 10“, IBM 2012

Figure 2. Row splits caused by the UPDATE statement



from Sarocco, Nicola, Gandhi: „A matter of time: Temporal data management in DB2 10", IBM 2012

Listing 15.

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

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

POLICY						
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	9999-12-30
1414	B7777	14000	N	750000	2008-05-01	2010-03-01
1414	B7777	12000	N	600000	2010-03-01	2010-06-01

from Sarocco, Nicola, Gandhi: „A matter of time: Temporal data management in DB2 10“, IBM 2012

DB2 Temporal DB Support (13)

state of table:

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	9999-12-30
1414	B7777	14000	N	750000	2008-05-01	2010-03-01
1414	B7777	12000	N	600000	2010-03-01	2011-01-01

overlaps

finishes

Listing 18.

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

Table 16. 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

from Sarocco, Nicola, Gandhi: „A matter of time: Temporal data management in DB2 10“, IBM 2012

DB2 Temporal DB Support (14)

Back to the „old“ terminology!

Listing 19. Creating a bitemporal table

```
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)
);
```

from Sarocco, Nicola, Gandhi: „A matter of time: Temporal data management in DB2 10“, IBM 2012

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');
```

Insert performed on 15.11.2011

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;
```

Update performed on 1.3.2012

Table 19. Effect of UPDATE on the bitemporal POLICY table

POLICY								
ID	VIN	annual_mile	rental_car	coverage_an	bus_start	bus_end	sys_start	sys_end
1111	A1111	10000	Y	500000	2012-01-01	2012-06-01	2012-03-01	9999-12-30
1111	A1111	10000	N	250000	2012-06-01	9999-12-30	2012-03-01	9999-12-30

Table 20. Effect of UPDATE on the bitemporal POLICY_HISTORY table

POLICY_HISTORY								
ID	VIN	annual_mileage	rental_car	coverage_amt	bus_start	bus_end	sys_start	sys_end
1111	A1111	10000	Y	500000	2012-01-01	9999-12-30	2011-11-15	2012-03-01

from Sarocco, Nicola, Gandhi: „A matter of time: Temporal data management in DB2 10“, IBM 2012

current:

ID	VIN	annual_mile	rental_car	coverage_ar	bus_start	bus_end	sys_start	sys_end
1111	A1111	10000	Y	500000	2012-01-01	2012-06-01	2012-03-01	9999-12-30
1111	A1111	10000	N	250000	2012-06-01	9999-12-30	2012-03-01	9999-12-30

history:

ID	VIN	annual_mileage	rental_car	coverage_amt	bus_start	bus_end	sys_start	sys_end
1111	A1111	10000	Y	500000	2012-01-01	9999-12-30	2011-11-15	2012-03-01

Listing 23. Querying bitemporal data

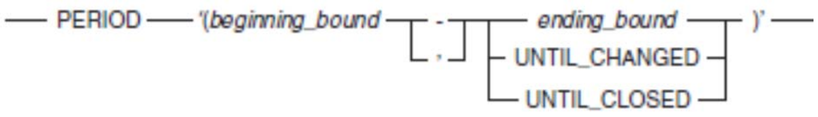
```
SELECT id, vin, annual_mileage, rental_car, coverage_amt,
       bus_start, bus_end, sys_start, sys_end
FROM policy FOR SYSTEM_TIME FROM '2010-07-10' TO '2012-07-11'
WHERE id = 1111;
```

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

ID	VIN	annual_mile	rental_car	coverage_ar	bus_start	bus_end	sys_start	sys_end
1111	A1111	10000	Y	500000	2012-01-01	2012-06-01	2012-03-01	9999-12-30
1111	A1111	10000	N	250000	2012-06-01	9999-12-30	2012-03-01	9999-12-30
1111	A1111	10000	Y	500000	2012-01-01	9999-12-30	2011-11-15	2012-03-01

from Sarocco, Nicola, Gandhi: „A matter of time: Temporal data management in DB2 10“, IBM 2012

PERIOD Data Type in Teradata SQL



Data Types

Other vendors are active as well – the new standard features will soon be widely available (cross fingers)!

Just one recent example:

Teradata is already ahead of SQL:2011 in supporting a data type PERIOD (well in the tradi-tion of SQL/Temporal)

IF the format of the Period literal is ...	THEN the data type is ...
hh:mi:ss	PERIOD(TIME(0)).
hh:mi:sssignhh:mi	PERIOD(TIME(0) WITH TIME ZONE).
hh:mi:ss.ssssss	PERIOD(TIME(n)), where *n* is the maximum number of fractional seconds digits in the beginning and ending bound values, or, if the ending bound value is UNTIL_CHANGED, the number of fractional seconds digits in the beginning bound value.
hh:mi:ss.sssssssignhh:mi	PERIOD(TIME(n) WITH TIME ZONE), where *n* is the maximum number of fractional seconds digits in the beginning and ending bound values, or, if the ending bound value is UNTIL_CHANGED, the number of fractional seconds digits in the beginning bound value.
YYYY-MM-DD	PERIOD(DATE).
YYYY-MM-DD hh:mi:ss	PERIOD(TIMESTAMP(0)).
YYYY-MM-DD hh:mi:sssignhh:mi	PERIOD(TIMESTAMP(0) WITH TIME ZONE).
YYYY-MM-DD hh:mi:ss.ssssss	PERIOD(TIMESTAMP(n)), where *n* is the maximum number of fractional seconds digits in the beginning and ending bound values, or, if the ending bound value is UNTIL_CHANGED, the number of fractional seconds digits in the beginning bound value.

Ultima

- Time for this lecture has come to an end by now.

- It was the main goal of this lecture to make you aware of the complex and subtle nature of time and of maintaining temporal information in a database properly.

- The lecture leaves you with a dilemma:
  - On the one hand, you learned a lot about the tedious and intricate way how temporal issues can be handled in „classical“ SQL without specific support for managing time.
  - On the other hand, you learned about really attractive new syntactical SQL extensions which meanwhile made it into the standard (and already partially into products). So why bother with the „old style“?

- In order to understand the semantics of these extensions, however, (which is really complex), knowing the „hard way“ is nearly inevitable.

- Another (intended) effect of knowing about the complex way of handling time in SQL „as of now“ is that you have a kind of measure against which to judge current and future new (?) features of commercial products – if you have a vision, you can see how short they still come in many respects.