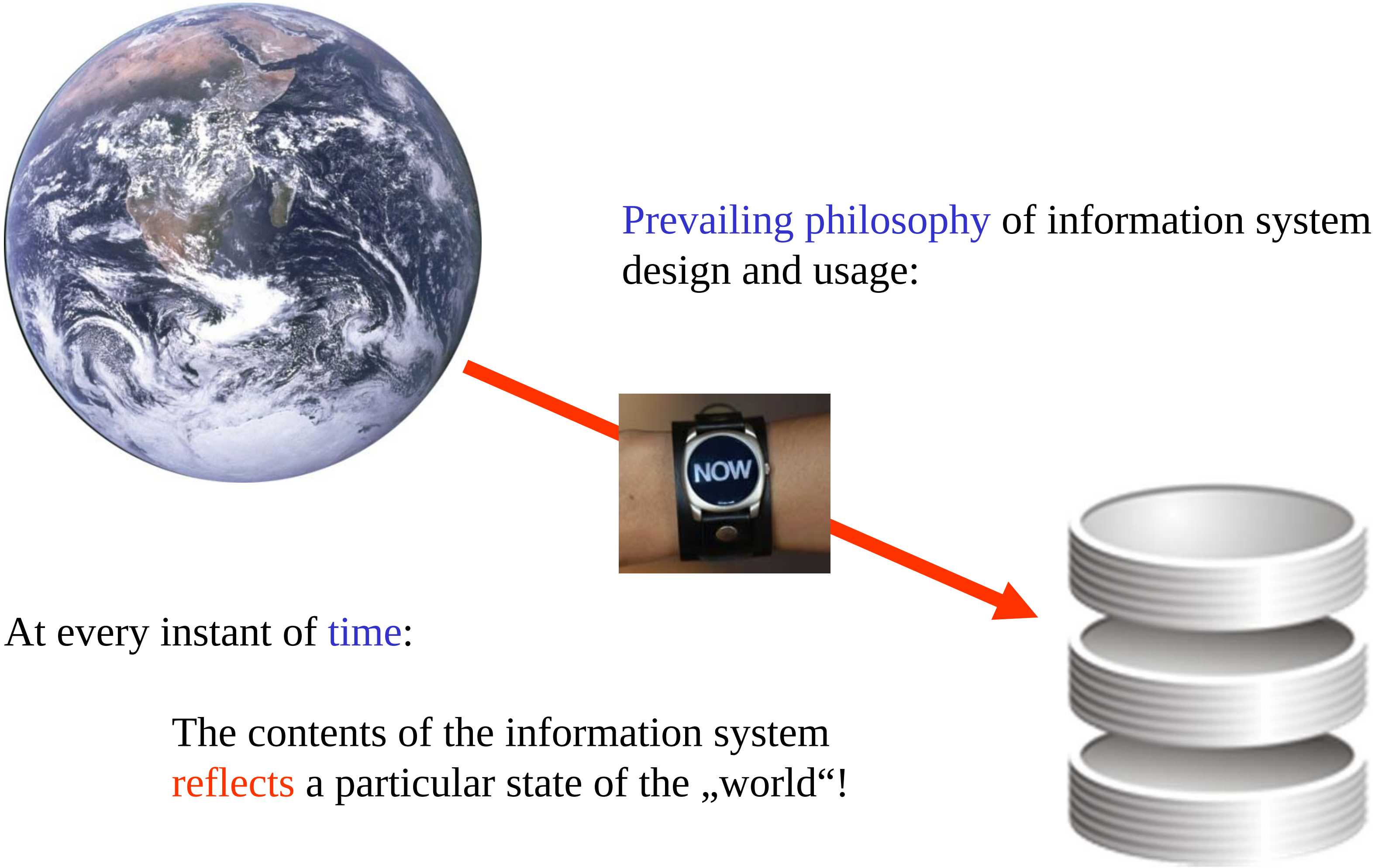
Temporal Information Systems

SS 2015

„Time About Data“ – Keeping a History of Change

Chapter 3

IS and Change: Some Philosophy Ahead (1)



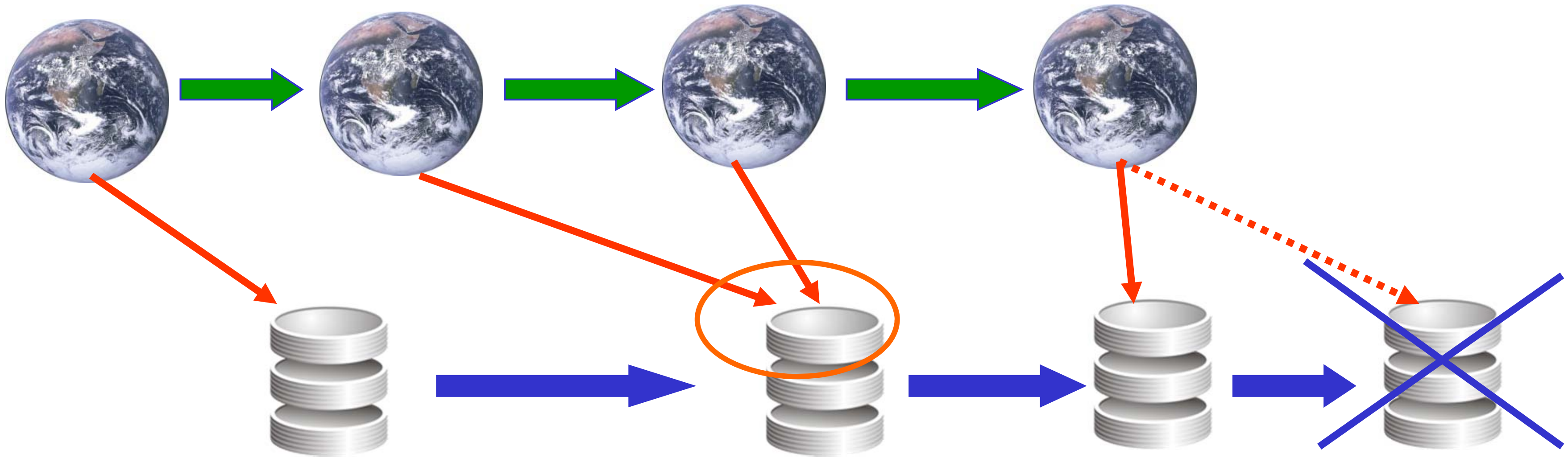
Prevailing philosophy of information system design and usage:

At every instant of time:

The contents of the information system reflects a particular state of the „world“!

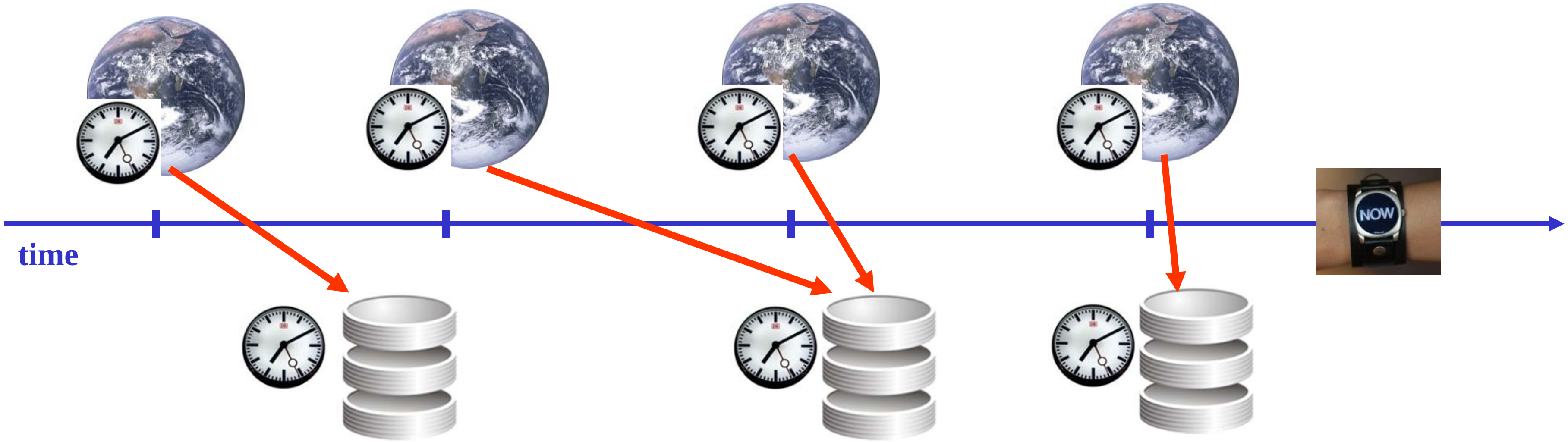
Databases represent snapshots of the world (ideally, as it is just now)!

IS and Change: Some Philosophy Ahead (2)



- Changes of „the world“ ought to be reflected by changes of the IS (we can‘t be sure, though!).
- IS evolution is always delayed wrt world evolution, but it often changes synchronously.
- However, there may be cases where the world changes more often than the IS:
  Several changes of the world may be reflected by a single change of the IS.
- An IS evolves only if the world has changed before –
  IS never change without a cause originating from „the world“.

IS and Change: Some Philosophy Ahead (3)



- Every change in the world takes place at a particular moment in time. So does every change of the database!
- This moment in time might be unknown, or considered irrelevant for the IS.
- Thus, the time of change is not always recorded in the IS.
- Even the sequence of changes occurring might be lost once reflected in the IS.
- Many (most?) IS do just represent the current state of the world, but no history at all!

Databases That Never Forget

> „The database is not the database – the log is the database,
> and the database is just an optimized access path to the
> most recent version of the log“.

(B.-M. Schueler in „Update Reconsidered“, 1977)

- This chapter will be concerned with techniques of keeping track of all changes of certain tables of a temporal database by logging each change and keeping all versions of the resp. tables.

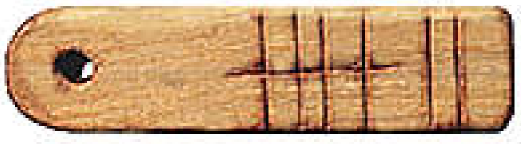
- Such versioned databases are required nowadays in a wide variety of application domains, in particular when legal problems (e.g., liability and auditing) have to be expected. More and more often, the provenance (history of origin) of data has to be proved.

- The technical key decision for such services is to automatically record all changes without human users being able to influence this process (administrators included). Key problem is how to properly query such databases (and, sometimes, how to update them properly).

Reminder (From Chapter 0): Logs

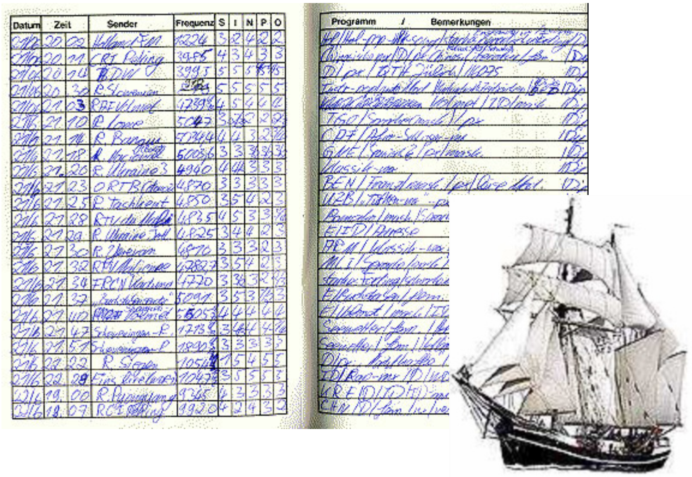
(The „log lady“
from the TV series
„Twin Peaks“)

Log: originally „part of the stem of a tree“



Later: „Tally stick“, used for recording consumption

Even later:
Ship‘s logbook, used for recording changes of direction and relevant events of a trip



Another Reminder: Log and Logging in Transaction Management

- The terms log/logging are well-known in the DB context, as they have been around in the context of transaction management in a DBMS since many decades.

- Transactions are sequences of change statements treated as a unit, i.e., they are either performed entirely (and successfully), or not at all, not even partially, in case of „failure“ of at least one of the component operations (atomicity of a transaction).

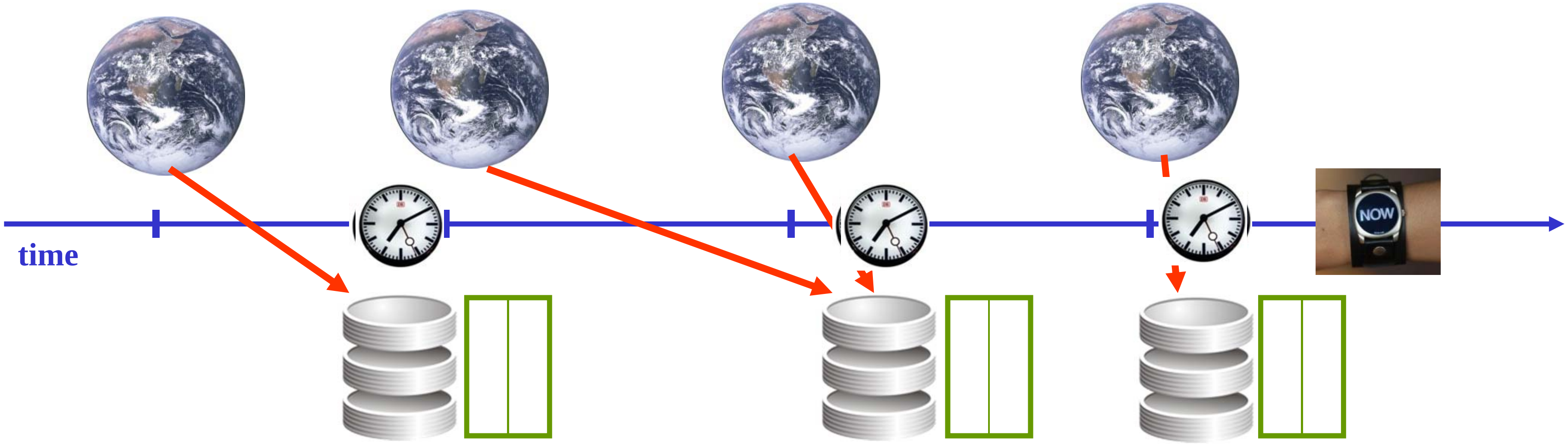
- Every DBMS controls every transaction wrt physical and logical consistency and protects and controls execution order in case of „competing“ transactions simultaneously trying to modify the same data (synchronisation).

- If any unresolvable problem occurs, execution of the affected transaction is stopped, and all changes to the DB already performed are rolled back till a consistent previous state has been reached (recovery).

- In order to be able to perform rollback, a temporary log of all performed operations of each active transaction is kept by the transaction manager of the DBMS.

The log we are speaking about here, is a different one – kept permanently!!

Keeping Data About History



- In this chapter, we will clearly separate the data part of a relational tuple from the history part of that same tuple:

  Timestamps added to tuples (representing facts in reality) denote those periods during which the resp. tuple was current in the database.

- Current changes in reality (called logical modifications) are always translated into physical modifications of the history DB preserving past data and recording the time of modification by means of start/end timestamps.

- Without explicitly mentioning, we assume that timestamps refer to the time of DB modification rather than to the time of „change in the real world“.

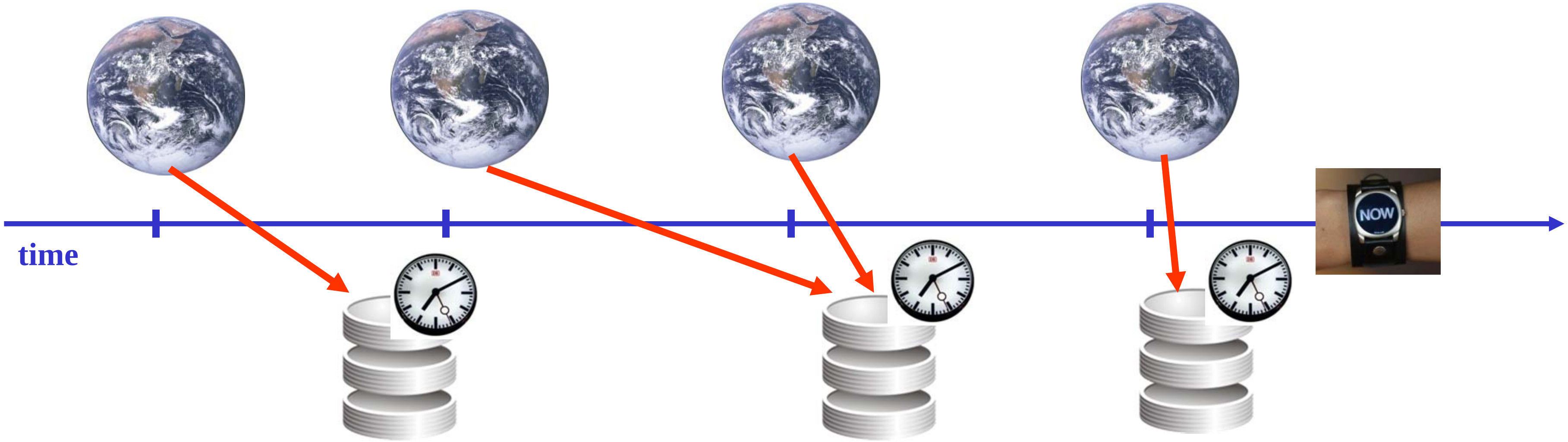
Timestamp: Dual Meaning – Beware of the Ambiguity!

The term „timestamp“ has been used on the previous slide in a different sense than in chapter 2 – both forms of usage are common in TDB research!

1) In SQL, there is the data type TIMESTAMP consisting of DATE-TIME values.

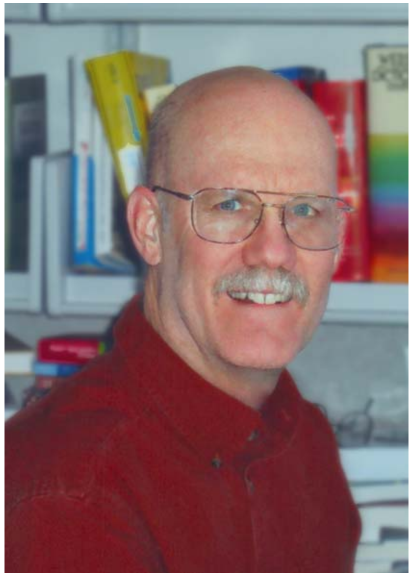
2) In databases keeping history of change of tables, the value of some temporal data type added to each tuple (representing facts in reality) in order to indicate when these tuples were „valid“ in reality or in the database, the additional temporal value is called the timestamp of the resp. tuple.

Transaction Time



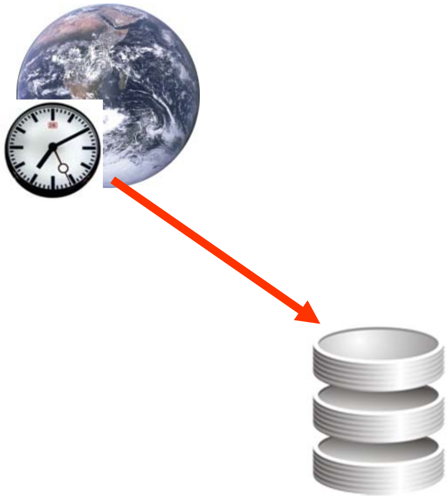
- In temporal DB research, storing those instants when data change in the DB, resp., those periods when data were current in the DB, is called keeping transaction time.
- Ideally, transaction timestamps are referring to the system clock, not to the watches of humans issuing modification commands. Again ideally, transaction time timestamps are generated automatically by the DBMS.
- Even more ideally, TT timestamps cannot be modified by human users later on anymore!
- In comparison, time of change in reality will be called valid time.

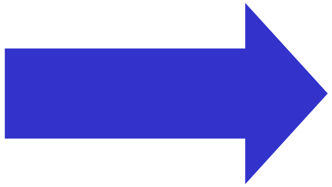
ATTENTION: We are Using Transaction Time for History Keeping!

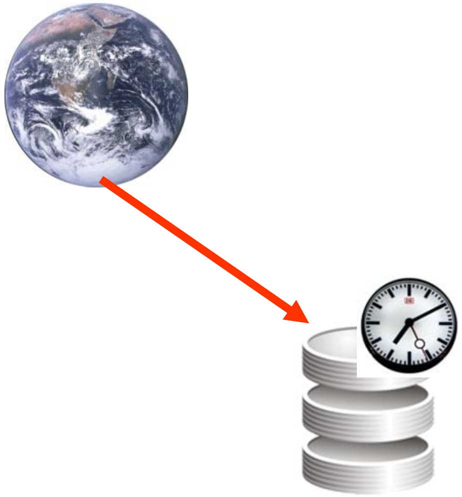


- This chapter mainly follows chapters 5-7 in the book by Snodgrass, but . . .
- whereas Snodgrass discusses history keeping in terms of timestamps referring to the clock of „the world“ (valid time),
- we do so in terms of the clock of the DBMS (transaction time)!

© adpic







Reminder: Exams Database with Potential Mistakes and Corrections

Student	**Class**	**Signed_up**	**Dropped**	**Grade**	**Exam Date**
John	1203	11.11.2010		1,3	13.2.2011
Jack	1203	19.11.2010	2.1.2011		
Tim	1203	21.11.2010		3,0	18.3.2011
Pete	1203	27.11.2010	3.2.2011	5,0	18.3.2011
John	2201	11.11.2010		1,7	19.2.2011
Jack	2201		2.1.2011		
Tim	3203	2.12.2010		3,7	1.4.2010

Failed or dropped?

Sign-up date missing?

Mistyped?

Snapshot of the DB as of April 1st, 2011

After Several Corrections (Done, but not Recorded)

All being valid time values!

Student	Class	Signed_up	Dropped	Grade	Exam Date
John	1203	11.11.2010		1,3	13.2.2011
Jack	1203	19.11.2010	2.1.2011		
Tim	1203	21.11.2010		**2,7**	18.3.2011
Pete	1203	27.11.2010	3.2.2011		
John	2201	11.11.2010		1,7	19.2.2011
Tim	3203	2.12.2010		3,7	1.4.2011

Snapshot of the DB as of April 8th, 2011

Keeping Track of All Changes (Using Simulated Periods as Timestamps)

Student	Class	Signed_up	Dropped	Grade	Exam Date	From	To
John	1203	11.11.2010				11.11.2010	14.2.2011
John	1203	11.11.2010		1,3	13.2.2011	14.2.2011	
Jack	1203	19.11.2010				19.11.2010	2.1.2011
Jack	1203	19.11.2010	2.1.2011			2.1.2011	[illegible]
Tim	1203	21.11.2010				21.11.2010	20.3.2011
Tim	1203	21.11.2010		3,0	18.3.2011	20.3.2011	8.4.2011
Tim	1203	21.		2,7	18.3.2011	8.4.2011	
Pete	1203	27.11.2010				27.11.2010	3.2.2011
Pete	1203	27.11.2010	3.2.2011			3.2.2011	21.3.2011
Pete	1203	27.11.2010	3.2.2011	5,0	18.3.2011	[illegible]	[illegible]
Pete	1203	27.11.2010	3.2.2011			6.4.2011	
John	2201	11.11.2010				11.11.2010	21.2.2011
John	2201	11.11.2010		1,7	19.2.2011	21.2.2011	
Jack	2201		2.1.2011			2.11.2010	7.4.2011
Tim	3203	2.12.2010		3,7	1.4.2010	1.4.2011	4.4.2011
Tim	3203	2.12.2010		3,7	1.4.2011	4.4.2011	

Valid time

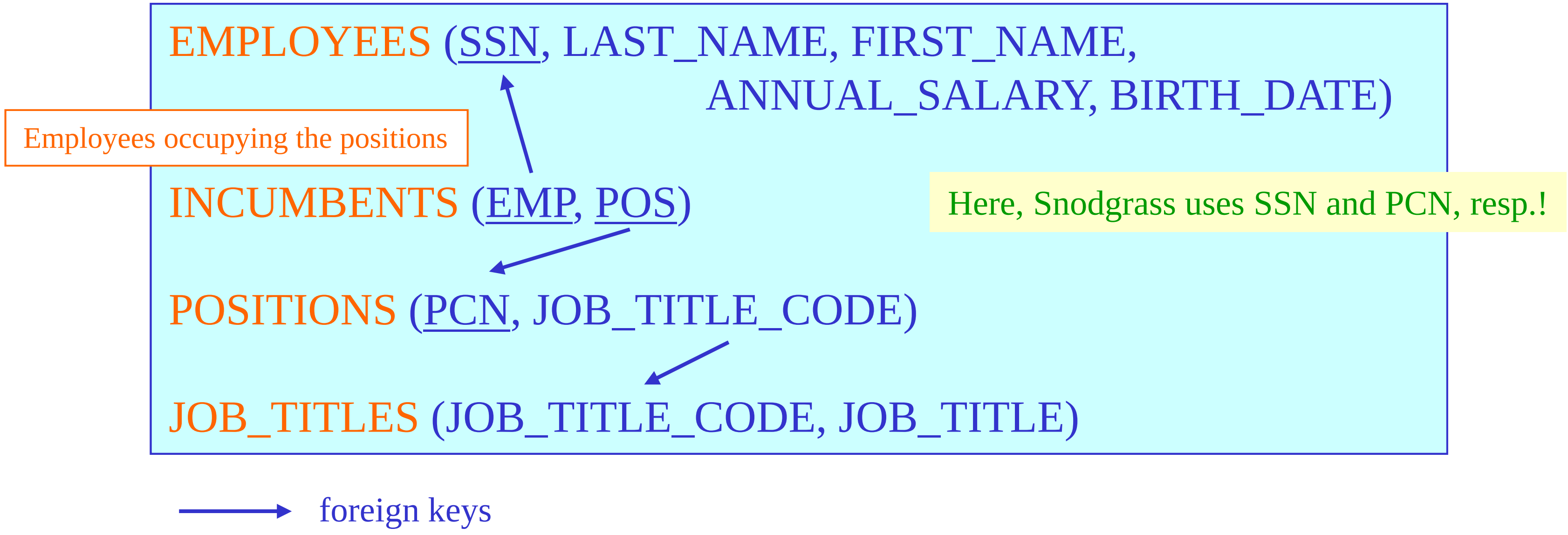
Transaction time

Data part

History part

Motivating Discussion for History Keeping

In the following, we will discuss issues, implications and alternative solutions for the problem of keeping track of all changes using the running example used in the Snodgrass book in chapters 5-7:



EMPLOYEES (SSN, LAST_NAME, FIRST_NAME, ANNUAL_SALARY, BIRTH_DATE)

Employees occupying the positions

INCUMBENTS (EMP, POS)

Here, Snodgrass uses SSN and PCN, resp.!

POSITIONS (PCN, JOB_TITLE_CODE)

JOB_TITLES (JOB_TITLE_CODE, JOB_TITLE)

foreign keys

How to keep history of all changes by extending these tables?

(not by storing separate archives – we want to query past data like present data)

INCUMBENTS Extended with a Date Timestamp

How to extend the INCUMBENTS table in view of being able to record the history of position assignments in the company?

INCUMBENTS

EMP	**POS**	**SINCE**
111223333	900225	1996-01-01
111223333	120033	1996-06-01
111223333	137112	1996-10-01
444332222	120033	1997-01-01

1st idea:

Add a single column for recording the start date of any assignment!

Obvious disadvantages:

- It is not possible to record if somebody lost his position without being reassigned a new one, e.g.:
  111223333 is fired on 1996-10-31
- Gaps in assignment cannot be represented either:
  111223333 is without assignment during September 1996, and only reassigned a position on 1996-10-01.

INCUMBENTS Extended with a Period Timestamp

Both problems can be avoided if the period of assignment is represented in full – in Snodgrass‘ book, periods are „simulated“ using two date columns. Throughout the following, we will represemt periods as [close, open) intervals:

INCUMBENTS

EMP	**POS**	**START_DATE**	**END_DATE**
111223333	900225	1996-01-01	1996-06-01
111223333	120033	1996-06-01	1996-09-01
111223333	137112	1996-10-01	1996-10-31
444332222	120033	1997-01-01	?

1-month gap

end of employment

A new problem arises though: How to deal with current assignments?

- If the contract provides a specified end date, this may be used for delimiting the assignment period, even though it ranges into the future.
- But what if there is no assignment end in the contract, i.e. if assignment is „until changed“ or „until employee fired“?

Representing Open-Ended Assignments: Four Alternatives

a) Using a very much earlier date, e.g., begin of times:

EMP	**POS**	**START_DATE**	**END_DATE**
444332222	120033	1997-01-01	**0001-01-01**

b) Using today as end date – changing every day at midnight:

EMP	**POS**	**START_DATE**	**END_DATE**
444332222	120033	1997-01-01	**2015-05-14**

c) Using a date very much in the future, e.g. end of times:

EMP	**POS**	**START_DATE**	**END_DATE**
444332222	120033	1997-01-01	**9999-12-31**

d) Leaving the respective field empty, i.e., using a null value implicitly:

EMP	**POS**	**START_DATE**	**END_DATE**
444332222	120033	1997-01-01	*NULL*

Representing Open-Ended Assignments (2)

a) Using a very much earlier date, e.g., begin of times:

EMP	POS	START_DATE	END_DATE
444332222	120033	1997-01-01	0001-01-01

- current queries: WHERE END_DATE = '0001-01-01'
- unintuitive: Convention has to be communicated to everybody!
- not really recommended

b) Using today as end date – changing every day at midnight:

EMP	POS	START_DATE	END_DATE
444332222	120033	1997-01-01	2011-05-23

- current queries: WHERE END_DATE = CURRENT_DATE,
- Effort for keeping up to date is prohibitively high!
- not at all recommended

Representing Open-Ended Assignments (3)

c) Using a date very much in the future, e.g. end of times:

EMP	**POS**	**START_DATE**	**END_DATE**
444332222	120033	1997-01-01	**9999-12-31**

- current queries: WHERE END_DATE = '9999-12-31'
- Semantically problematic if „real“ future timestamps are to be expected!
- Probably the best choice among the four less than ideal choices!

d) Leaving the respective field empty, i.e., using a null value implicitly:

EMP	**POS**	**START_DATE**	**END_DATE**
444332222	120033	1997-01-01	*NULL*

- current queries: WHERE END_DATE IS NULL
- Prevents other usage of NULL (with different meaning)!
- END_DATE not comparable with other dates!
- not really recommended

Potential Trouble with Overlapping or Meeting Assignment Periods

A state like the following is possible (if by mistake or carelessness) in the INCUMBENTS table – it is not very pleasant (and troublesome for query answering) to admit overlapping or meeting periods of assignment, at least for the same position (maybe double assignments for different positions is admissible):

EMP	**POS**	**START DATE**	**END DATE**
111223333	900225	1996-01-01	1996-05-01
111223333	900225	1996-03-01	1996-06-01
111223333	120033	1996-06-01	1996-08-01
111223333	120033	1996-08-01	1996-10-01

overlaps

meets

Obviously, the following „non-redundant“ representation is to be preferred:

EMP	**POS**	**START DATE**	**END DATE**
111223333	900225	1996-01-01	1996-06-01
111223333	120033	1996-06-01	1996-10-01

On first glance, the „redundancy“ in the first case seems to come from violating the primary key constraint in the extended table. But is this really the solution?

Primary Keys and Timestamps (1)

What if an employee is reassigned to a position (s)he already occupied earlier, as in the following case:

SSN	PCN	START_DATE	END_DATE
111223333	900225	1996-01-01	1996-06-01
111223333	120033	1996-06-01	1996-09-01
111223333	900225	1996-09-01	1996-10-01
111223333	137112	1996-10-01	1996-10-31

The original primary key (SSN, PCN) only states, that at every point in time the table INCUMBENTS is free of duplicate rows.

After adding a period timestamp to each row, the original key would in addition prevent any reassignments ever – which might be far too strong. Thus, it is necessary to include the timestamp into the primary key as well! We call this a „temporal key“.

There are three alternatives how to design a temporal key for this table:

(SSN, PCN, **START_DATE**), or (SSN, PCN, **END_DATE**), or
(SSN, PCN, **START_DATE, END_DATE**)

Primary Keys and Timestamps (2)

A slight modification of the previous example (introducing an overlap again) shows that none of the three candidate keys prevents this situation:

SSN	PCN	START_DATE	END_DATE
111223333	900225	1996-01-01	1996-05-01
111223333	900225	1996-04-01	1996-06-01
111223333	120033	1996-06-01	1996-09-01
111223333	137112	1996-10-01	1996-10-31

overlaps

(SSN, PCN, START_DATE):	satisfied
(SSN, PCN, END_DATE):	satisfied
(SSN, PCN, START_DATE, END_DATE):	satisfied

The purpose of extending the key was to prevent any duplicate row in any of the temporal snapshots of this historical table.

However, for every day in April, the assignment of 111223333 to 900225 is represented twice.

How to avoid this?

Sequenced or Snapshot Timestamping

SSN	**PCN**	**START_DATE**	**END_DATE**
111223333	900225	1996-01-01	1996-06-01

- A key idea of history keeping is to be able to represent the presence of a particular fact in the DB at every moment in time. The period representation is just a shorthand for the more detailed instant („snapshot“) representation:

SSN	**PCN**	**DATE**
111223333	900225	1996-01-01
111223333	900225	1996-01-02
111223333	900225	1996-01-03
. . .		
111223333	900225	1996-05-31

Don‘t forget;
Periods are intervals written in [close,open) style!

- We call this „expanded“ representation, which uses instant timestamping, the sequenced version of the history.

Expanding Period Timestamps and Duplicates (1)

SSN	PCN	START_DATE	END_DATE
111223333	900225	1996-01-01	1996-05-01
111223333	900225	1996-04-01	1996-06-01

overlaps

When expanding several timestamped rows into sequenced form, each period is expanded separately. Thus, if timestamp periods overlap, duplicate rows will result in the sequenced representation:

1st row expanded

2nd row expanded

SSN	PCN	DATE
111223333	900225	1996-01-01
. . .		
111223333	900225	1996-04-01
111223333	900225	1996-04-02
. . .		
111223333	900225	1996-04-30
111223333	900225	1996-04-01
. . .		
111223333	900225	1996-04-30
111223333	900225	1996-05-01
. . .		
111223333	900225	1996-05-31

Duplicate April days

Thus, no key exists that solves the uniqueness problem!

Expanding Period Timestamps and Duplicates (2)

SSN	PCN	START_DATE	END_DATE
111223333	900225	1996-01-01	1996-**05**-01
111223333	900225	1996-**05**-01	1996-06-01

meets

Situations where two timestamped rows are identical in the non-temporal part, but have meeting timestamps is different in this respect (as compared to the overlaps case)! Even though combining both rows by „merging“ the two timestamps is still preferable, no duplicate (and thus key) problems arise, however, in the sequenced representation of the data:

SSN	PCN	DATE
111223333	900225	1996-01-01
. . .		
111223333	900225	1996-04-01
111223333	900225	1996-04-02
. . .		
111223333	900225	1996-04-30
111223333	900225	1996-05-01
. . .		
111223333	900225	1996-05-31

1st row expanded

2nd row expanded

No duplicate entries after expansion!

Primary Keys and Timestamps (3)

- The problem we just started to investigate (How to avoid duplicates in a „table with history keeping“?) does not have an easy solution in traditional SQL!
- For now, we will postpone a further discussion of the temporal key problem to the end of this lecture.
- The related problem of temporal foreign keys will also be discussed later.
- Instead, we start treatment of queries on tables with TT timestamps . This will be continued in three weeks, followed by discussion of modifications of this kind of tables.

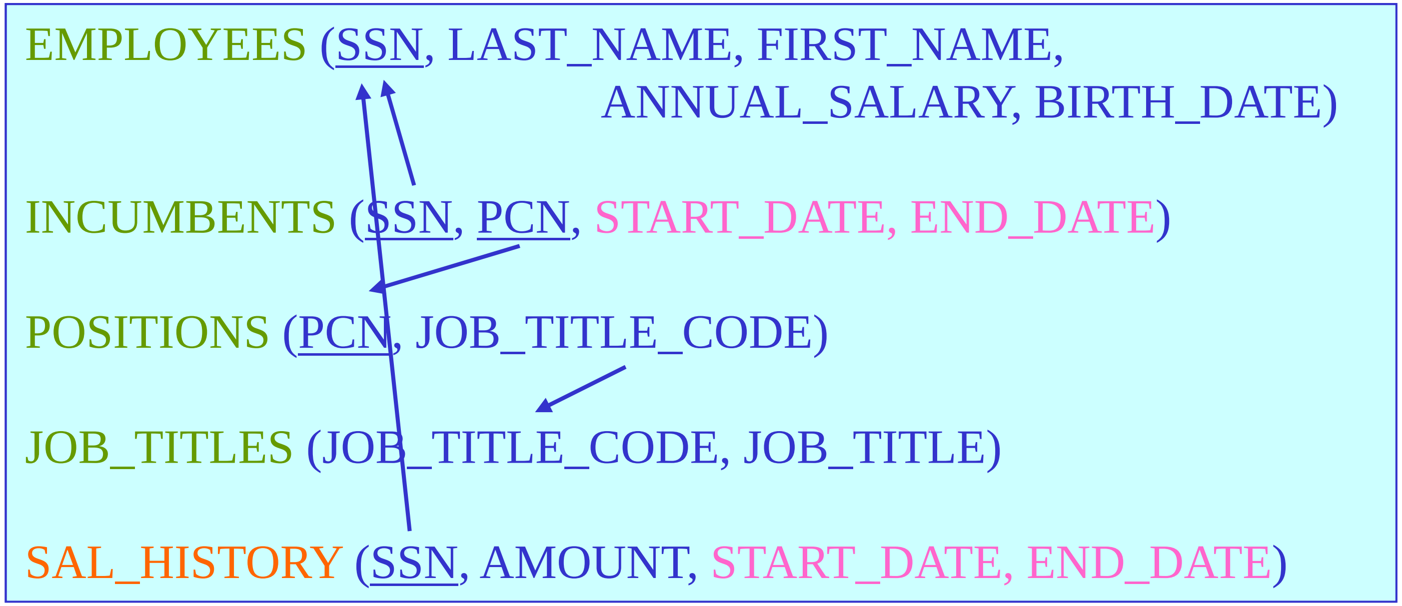
Classifying Queries on Historical Tables

As already discussed in the context of temporal keys, there are 4 different kinds of queries that can be distinguished wrt their relationship to temporal aspects:

- non-temporal queries: ignoring timestamps
- temporal queries: referring to timestamps
    - time-slice queries: evaluated at one specific point in time
        - current queries: timestamp period overlapping „now“
        - past/future queries: timestamp period overlapping some specific past/future instant
    - sequenced queries: evaluated at each point in time and referring to the entire history
    - non-sequenced queries: evaluated over a history but treating timestamps as „ordinary“ columns

Example Extended and „Temporalized“

For the following discussion, assume that there is an additional table recording the salary history of each employee – both, this one and INCUMBENTS being historical:



EMPLOYEES (SSN, LAST_NAME, FIRST_NAME, ANNUAL_SALARY, BIRTH_DATE)

INCUMBENTS (SSN, PCN, START_DATE, END_DATE)

POSITIONS (PCN, JOB_TITLE_CODE)

JOB_TITLES (JOB_TITLE_CODE, JOB_TITLE)

SAL_HISTORY (SSN, AMOUNT, START_DATE, END_DATE)

Let us turn to query answering now:

What is Bob's current position?

Current Queries (1)

What is Bob's current position?

```
SELECT P.JOB_TITLE_CODE
FROM    EMPLOYEES       AS  E,
        INCUMBENTS      AS  I,
        POSITIONS       AS  P
WHERE  E.FIRST_NAME = 'Bob'
   AND  E.SSN = I.SSN
   AND  I.PCN = P.PCN
   AND                  (CURRENT_DATE,
                         CURRENT_DATE + INTERVAL '1' DAY)
        OVERLAPS   (I.START_DATE, I.END_DATE)
```

This query works, if the „until changed“ end date is represented by a far away future date (e.g., end of time).

As only INCUMBENTS is a historical table, no other tests for „now“ are required.

Current Queries (2)

What is Bob's current position and salary?

```
SELECT JOB_TITLE_CODE1, AMOUNT
FROM   EMPLOYEES       AS  E,
       INCUMBENTS      AS  I,
       POSITIONS       AS  P,
       SAL_HISTORY     AS  H
WHERE E.FIRST_NAME = 'Bob'
   AND E.SSN = I.SSN
   AND I.PCN = P.PCN
   AND           (CURRENT_DATE, CURRENT_DATE + INTERVAL '1' DAY)
     OVERLAPS  (I.START_DATE, I.END_DATE)
   AND H.SSN = E.SSN
   AND           (CURRENT_DATE, CURRENT_DATE + INTERVAL '1' DAY)
     OVERLAPS  (H.START_DATE, H.END_DATE)
```

As now two historical tables are joined, two tests for validity „now“ are required!

Past Queries

What was Bob's position at the beginning of 1997?

```
SELECT JOB_TITLE_CODE1
FROM   EMPLOYEES      AS  E,
       INCUMBENTS     AS  I,
       POSITIONS      AS  P
WHERE E.FIRST_NAME = 'Bob'
  AND  E.SSN = I.SSN
  AND  I.PCN = P.PCN
  AND           (I.START_DATE, I.END_DATE)
     OVERLAPS   (DATE '1997-01-01', DATE '1997-01-02')
```

This works exactly like asking for the current date!

Queries asking for the state of affairs on a particular date are called snapshot or time-slice queries.

Sequenced Queries

Who makes or has made more than $50,000 annually?

sequenced projection

sequenced selection

```
SELECT   *
FROM     SAL_HISTORY
WHERE    AMOUNT > 50000
```

This returns each high-paid employee from current and past including timestamps, i.e., the answer table is a historical one, too.

Who makes or has made more than $50,000 or less than $10,000 annually?

sequenced union

```
(SELECT *
 FROM    SAL_HISTORY
 WHERE  AMOUNT > 50000)
       UNION ALL
(SELECT *
 FROM    SAL_HISTORY
 WHERE  AMOUNT < 10000)
```

Sequenced Join (1)

Provide the salary and position history for all employees.

```
SELECT H.SSN, H.AMOUNT, I.PCN, ?.START_DATE, ?.END_DATE
FROM    SAL_HISTORY     AS  H,
        INCUMBENTS      AS  I
WHERE  H.SSN=I.SSN  AND . . .
```

sequenced join

SAL HISTORY

SSN	AMOUNT	START DATE	END DATE
111223333	15.75	1995-03-01	1996-05-01
111223333	16.25	1996-05-01	1997-06-01
111223333	17.00	1997-06-01	1998-01-01

(two historical tables)

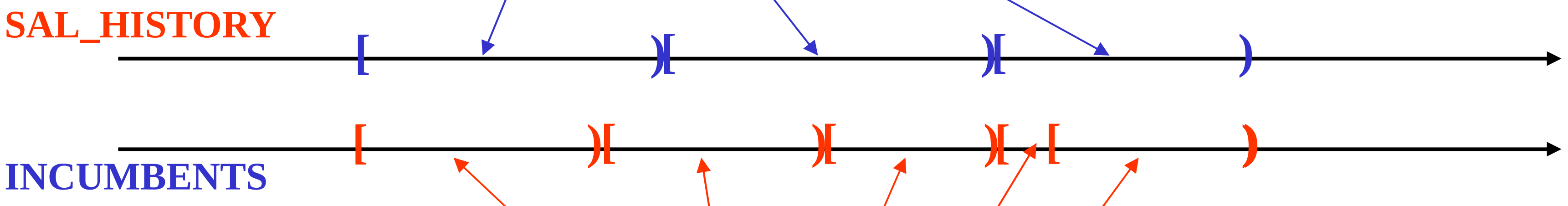
INCUMBENTS

SSN	PCN	START DATE	END DATE
111223333	341288	1995-03-01	1996-01-01
111223333	137112	1996-01-01	1996-10-01
111223333	120033	1996-10-01	1997-06-01
111223333	908654	1997-06-01	1998-01-01
111223333	723401	1997-10-01	1998-01-01

Sequenced Join (2)

SAL HISTORY

SSN	AMOUNT	START DATE	END DATE
111223333	15.75	1995-03-01	1996-05-01
111223333	16.25	1996-05-01	1997-06-01
111223333	17.00	1997-06-01	1998-01-01



INCUMBENTS

SSN	PCN	START DATE	END DATE
111223333	341288	1995-03-01	1996-01-01
111223333	137112	1996-01-01	1996-10-01
111223333	120033	1996-10-01	1997-06-01
111223333	908654	1997-06-01	1998-01-01
111223333	723401	1997-10-01	1998-01-01

Sequenced Join (3)

SAL HISTORY

SSN	AMOUNT	START DATE	END DATE
111223333	15.75	1995-03-01	1996-05-01
111223333	16.25	1996-05-01	1997-06-01
111223333	17.00	1997-06-01	1998-01-01

INCUMBENTS

SSN	PCN	START DATE	END DATE
111223333	341288	1995-03-01	1996-01-01
111223333	137112	1996-01-01	1996-10-01
111223333	120033	1996-10-01	1997-06-01
111223333	908654	1997-06-01	1998-01-01
111223333	723401	1997-10-01	1998-01-01

salary and position history

SSN	AMOUNT	PCN	START DATE	END DATE
111223333	15.75	341288	1995-03-01	1996-01-01
111223333	15.75	137112	1996-01-01	1996-05-01
111223333	16.25	137112	1996-05-01	1996-10-01
111223333	16.25	120033	1996-10-01	1997-06-01
111223333	17.00	908654	1997-06-01	1998-01-01
111223333	17.00	723401	1997-10-01	1998-01-01

Sequenced Join (4)

Provide the salary and position history for all employees.

A proper SQL formulation of this query requires to catch all possible situations where an INCUMBENTS row and a SAL_HISTORY row are jointly valid on at least one day. 9 of the 13 Allen relationships between intervals satisfy this condition:

starts, starts^{-1}, finishes, finishes^{-1}
during, during^{-1}, overlaps, overlaps^{-1}
equals

These 9 cases can be summarized using 4 disjoint conditions, such that the overall query consists of four subqueries (guaranteed not to have any answer in common):

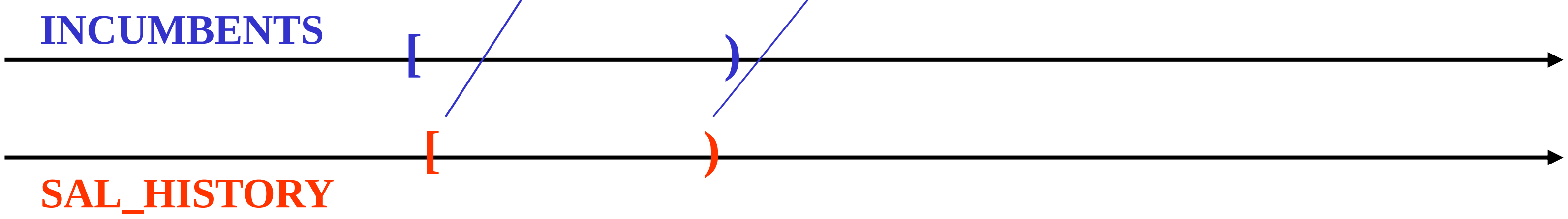
```
(SELECT  . . .  FROM  . . .)
            UNION ALL
(SELECT  . . .  FROM  . . .)
            UNION ALL
(SELECT  . . .  FROM  . . .)
            UNION ALL
(SELECT  . . .  FROM  . . .)
```

Sequenced Join (5)

Provide the salary and position history for all employees.

```
SELECT H.SSN, H.AMOUNT, I.PCN, H.START_DATE, H.END_DATE
FROM    SAL_HISTORY     AS  H,
        INCUMBENTS      AS  I
WHERE  . . .
```

```
 . . .     H.SSN = I.SSN
AND       I.START_DATE <= H.START_DATE
AND       H. END_DATE   <= I.END_DATE
```



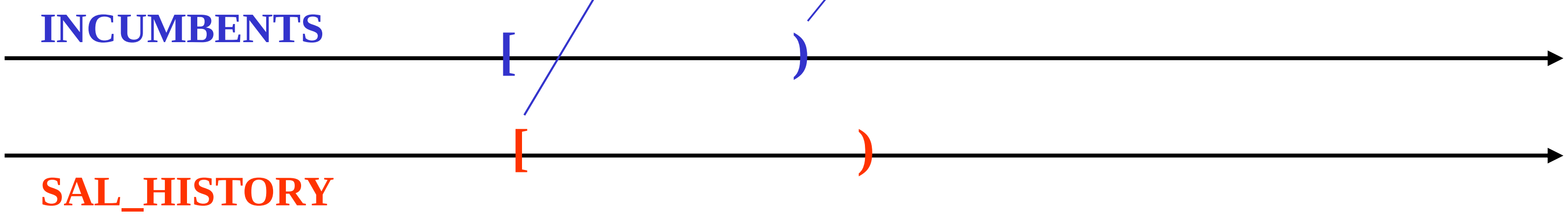
h starts i ∨ h during i ∨ h finishes i ∨ h equals i

Sequenced Join (6)

Provide the salary and position history for all employees.

```
SELECT H.SSN, H.AMOUNT, I.PCN, H.START_DATE, I.END_DATE
FROM     SAL_HISTORY     AS  H,
         INCUMBENTS      AS  I
WHERE  . . .
```

```
 . . .     H.SSN = I.SSN
AND        I.START_DATE <= H.START_DATE
AND        I.END_DATE < H.END_DATE
AND        H. START_DATE < I.END_DATE
```



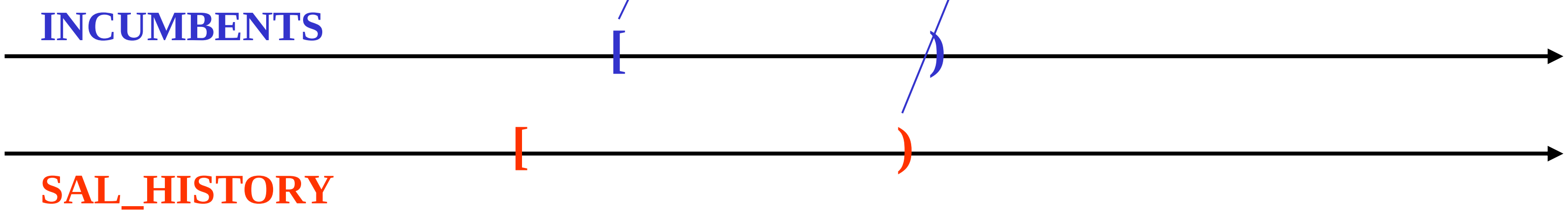
$h\ \text{overlaps}^{-1}\ i \ \vee\ h\ \text{starts}^{-1}\ i$

Sequenced Join (7)

Provide the salary and position history for all employees.

```
SELECT H.SSN, H.AMOUNT, I.PCN, I.START_DATE, H.END_DATE
FROM     SAL_HISTORY      AS  H,
         INCUMBENTS       AS  I
WHERE  . . .
```

```
. . .        H.SSN = I.SSN
    AND      I.START_DATE > H. START_DATE
    AND      H. END_DATE <= I.END_DATE
    AND      I.START_DATE < H.END_DATE
```



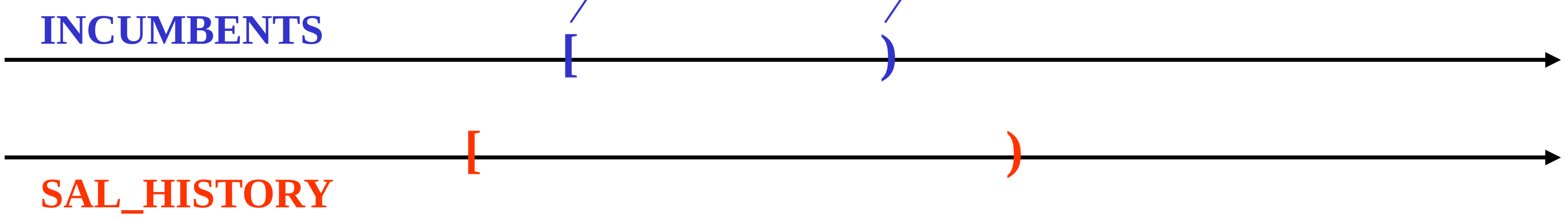
$h \text{ overlaps } i \vee h \text{ finishes}^{-1} i$

Sequenced Join (8)

Provide the salary and position history for all employees.

```
SELECT H.SSN, H.AMOUNT, I.PCN, I. START_DATE, I. END_DATE
FROM     SAL_HISTORY      AS  H,
         INCUMBENTS       AS  I
WHERE  . . .
```

```
 . . .      H.SSN = I.SSN
AND         I.START_DATE  > H.START_DATE
AND         I.END_DATE      <  H. END_DATE
```



h during^{-1} i

Sequenced Join (9)

Why four complicated cases – if the situations are exactly covered by OVERLAPS?

starts
during
equals
starts^{-1}
during^{-1}
finishes
overlaps
finishes^{-1}
overlaps^{-1}

==

OVERLAPS

The problem is to associate the correct choice of start and end point in the SELECT clause with the particular case (represented by the WHERE condition), e.g.:

```
SELECT H.SSN, H.AMOUNT, I.PCN, I. START_DATE, I. END_DATE
…
WHERE  …
                I.START_DATE  > H.START_DATE
AND             I.END_DATE    <  H. END_DATE
```

But there is a very elegant way out if using CASE in SELECT, …

Sequenced Join: Ultra Short Version

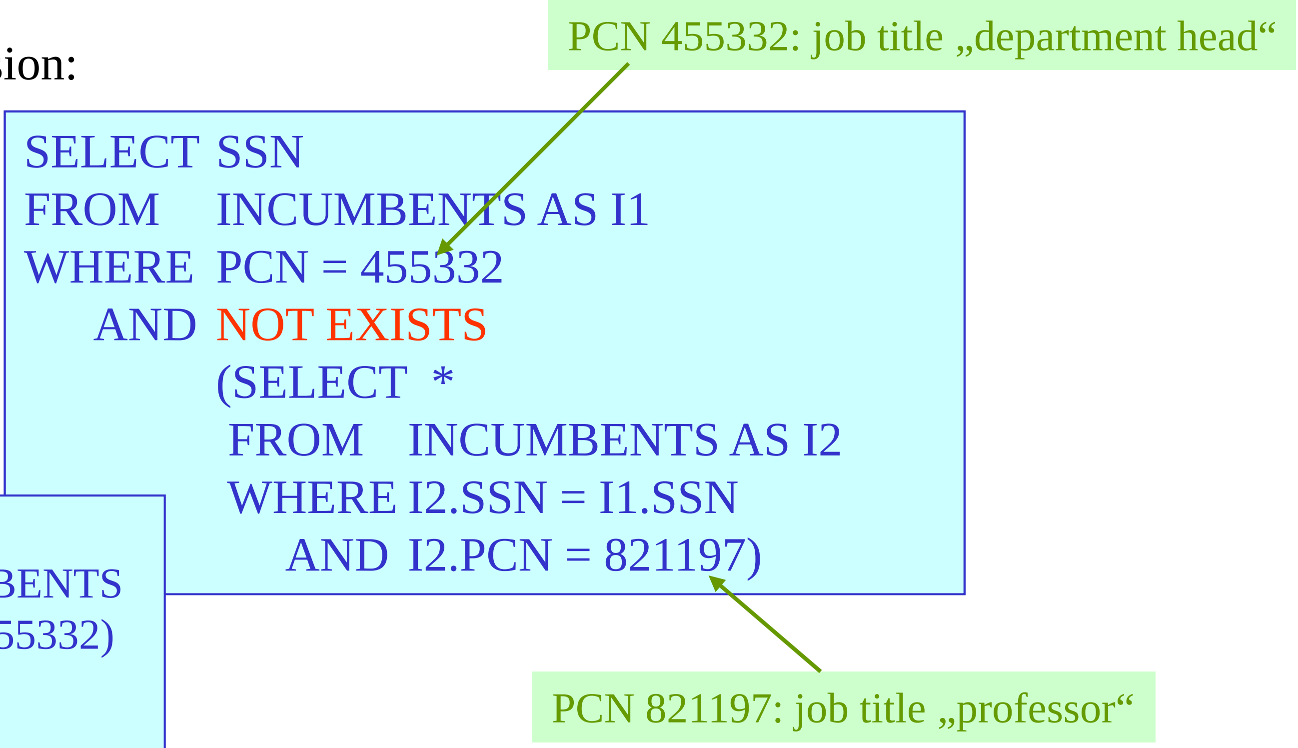
```
SELECT H.SSN, H.AMOUNT,
       (CASE   WHEN I.START_DATE < H.START.DATE
               THEN  H.START_DATE
               ELSE   I.START_DATE
         END)  AS START_DATE,
       (CASE   WHEN I.END_DATE < H.END.DATE
               THEN  I.END_DATE
               ELSE   H.END_DATE
         END)  AS END_DATE
FROM   SAL_HISTORY AS H,
       INCUMBENTS  AS I
WHERE  I.SSN = H.SSN
  AND  (I.START_DATE, I.END_DATE)
               OVERLAPS
       (H.START_DATE, H.END_DATE)
```

Thanks for this to Stephan Zacharias in TIS 2011!

Sequenced Difference (1)

List the employees who are department heads but are not also professors!

non-temporal version:



PCN 455332: job title „department head“

```
SELECT SSN
FROM   INCUMBENTS AS I1
WHERE  PCN = 455332
   AND NOT EXISTS
       (SELECT  *
        FROM   INCUMBENTS AS I2
        WHERE I2.SSN = I1.SSN
            AND  I2.PCN = 821197)
```

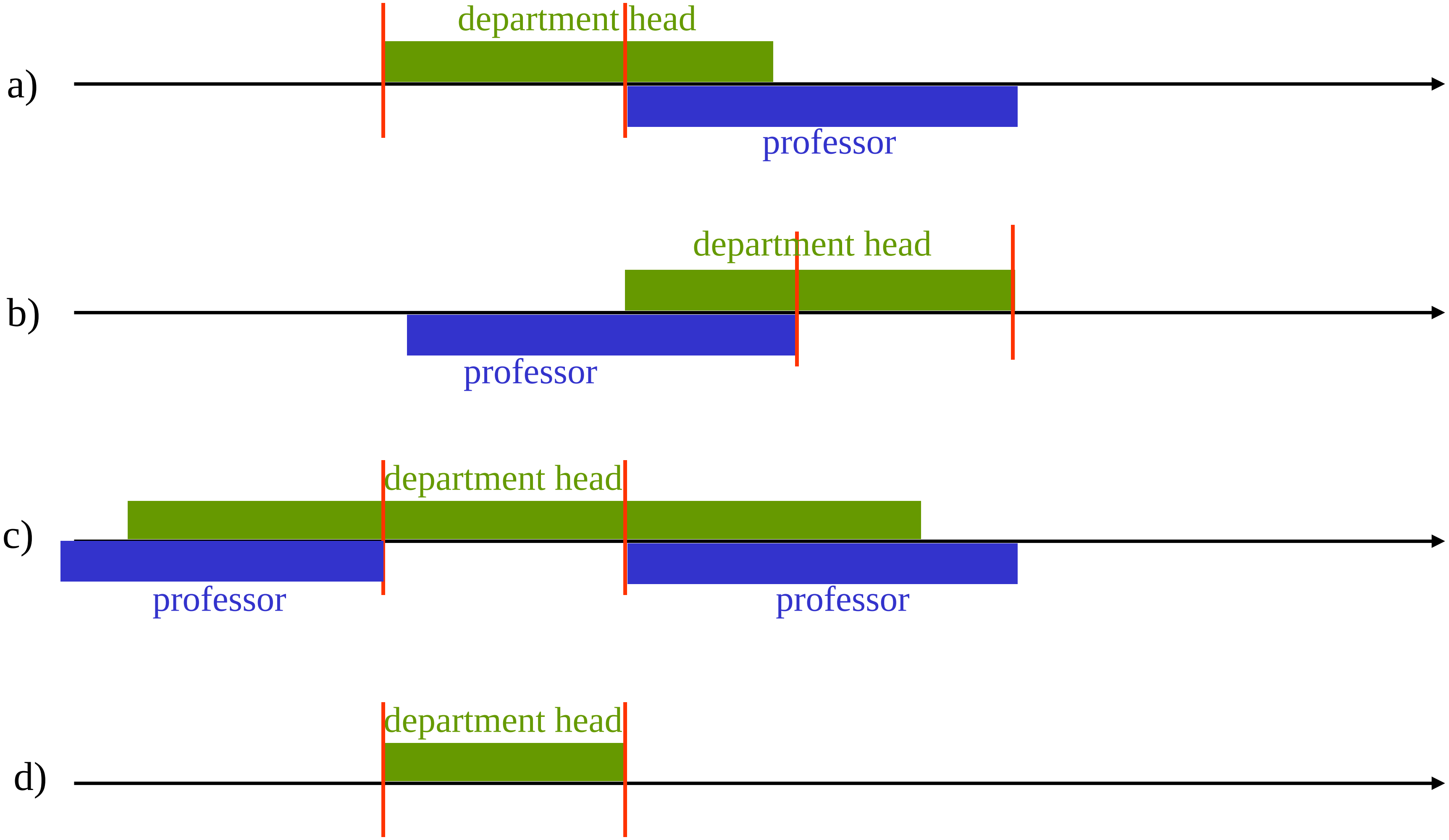
PCN 821197: job title „professor“

equivalent:

```
(SELECT SSN
 FROM   INCUMBENTS
 WHERE PCN = 455332)
EXCEPT
(SELECT SSN
 FROM   INCUMBENTS
 WHERE PCN = 821197)
```

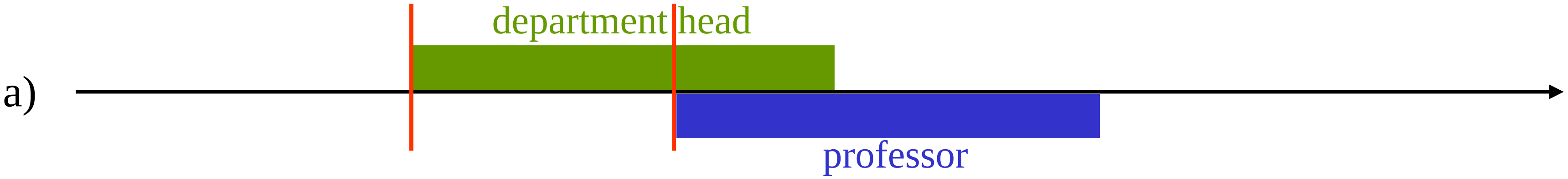
Sequenced Difference (2)

There are four different cases how the „except“ situation may have arisen in history:



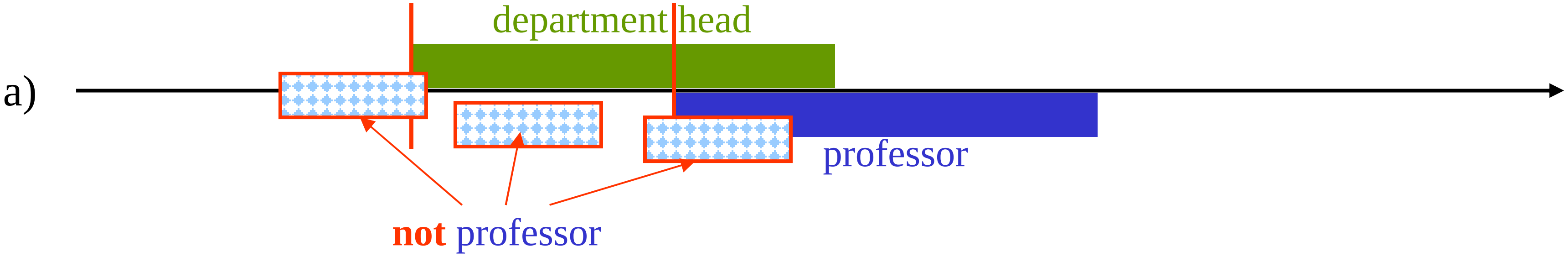
Sequenced Difference (3)

This time, however, it is not sufficient to simply specify the depicted situation in terms of SQL expressions. e.g.:

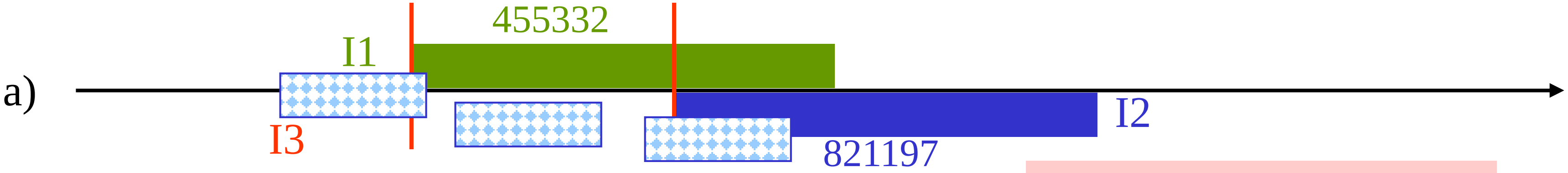


Instead, two specifications are required:

- The situation specific to the case has to take place, i.e., in case a), the period of being department head overlaps the period of being a professor, and the former starts earlier.
- No prior professorship period overlaps this department head period.



Sequenced Difference (4a)



```
SELECT I1.SSN,
       I1.START_DATE,
       I2.START_DATE AS END_DATE
FROM   INCUMBENTS AS I1,
       INCUMBENTS AS I2
WHERE  I1.PCN = 455332
   AND I2.PCN = 821197
   AND I1.SSN = I3.SSN
   AND I1. START_DATE =< I2.START_DATE
   AND I2. START_DATE =< I1.END_DATE
   AND I1. END_DATE =< I2.END_DATE
   AND -- No other professorship period OVERLAPS the department head
          period „from the left“ (to be continued)
```

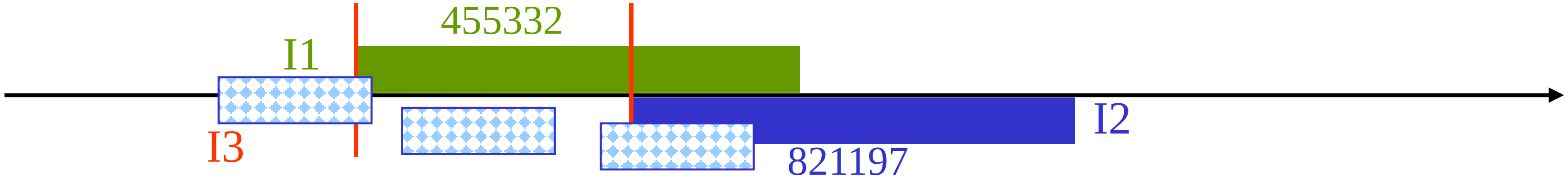
Specifies period during which the „difference answer“ is valid in „Case a“.

Specifies „Case a“ exactly!

Surprising: Difference can be expressed (in the sequenced case) without negation!

Sequenced Difference (4b)

a)



```
SELECT I1.SSN,
       I1.START_DATE, I2.START_DATE AS END_DATE
FROM   INCUMBENTS AS I1, INCUMBENTS AS I2
WHERE  …
   AND NOT EXISTS
           (SELECT *
            FROM   INCUMBENTS AS I3
            WHERE  I3.SSN = I1.SSN
               AND I3.PCN = 821197
               AND I1.START_DATE < I3.END_DATE
               AND I3.START_DATE < I2.START_DATE)
```

Specifies „overlapping from the left“!

Sequenced Difference (5)

List the employees who are (or were) department heads but are (or were) not also professors (at that time) !

The entire query is once again composed of 4 different subqueries, each specifying one of the 4 cases a) – d).

This time, however, there are potential overlaps of the four cases, thus duplicates may arise and UNION is required instead of UNION ALL (as for sequenced joins):

```
(SELECT  . . .  FROM  . . .)
          UNION
(SELECT  . . .  FROM  . . .)
          UNION
(SELECT  . . .  FROM  . . .)
          UNION
(SELECT  . . .  FROM  . . .)
```

Nonsequenced (Temporal) Queries (1)

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

```
SELECT AMOUNT
FROM    INCUMBENTS      AS I,
        POSITIONS       AS P,
        SAL_HISTORY     AS H
WHERE   I.SSN = H.SSN
  AND   I.PCN = P.PCN
  AND   P.JOB_TITLE_CODE = 20730
```

Queries like this one refer to historical tables (all three of them are) and retrieves answers from past data, too, but do not mention timestamp values in their output or treat timestamp columns as „ordinary“ ones without requiring an expansion of period timestamps into sequenced form.

This kind of query is called a nonsequenced query – the term does not mean the same thing as non-temporal query, though!

Nonsequenced Queries (2)

When did employees receive salary raises?

```
SELECT S2.SSN,
       S2.HISTORY_START_DATE AS RAISE_DATE
FROM   SAL_HISTORY AS S1,
       SAL_HISTORY AS S2
WHERE  S2.AMOUNT > S1.AMOUNT
   AND S1.SSN = S2.SSN
   AND S1.HISTORY_END_DATE = S2.HISTORY_START_DATE
```

This query is a nonsequenced one, too, as it refers to the entire history „as stored“ without being evaluated at each point in time (which would be sequenced).

It is not always that easy to recognize nonsequenced queries and to distinguish them from nontemporal and/or sequenced ones!

Eliminating Duplicates From Answers to Queries (1)

```
SELECT DISTINCT  SSN, PCN
FROM  INCUMBENTS
```

eliminating non-temporal duplicates

non-temporal duplicates

SSN	PCN	START_DATE	END_DATE
111223333	120033	1996-01-01	1996-06-01
111223333	120033	1996-04-01	1996-10-01
111223333	120033	1996-04-01	1996-10-01
111223333	120033	1996-10-01	1998-01-01
111223333	120033	1997-12-01	1998-01-01

non-sequenced duplicates

containing sequenced duplicates

```
SELECT DISTINCT  *
FROM  INCUMBENTS
```

eliminating non-sequenced duplicates

How to eliminate sequenced duplicates?

(Temporal key property doesn‘t help – answers to queries don‘t have keys!)

Eliminating Duplicates (2)

1.

SSN	PCN	START_DATE	END_DATE
111223333	120033	1996-01-01	1996-06-01
111223333	120033	1996-04-01	1996-10-01
~~111223333~~	~~120033~~	~~1996-04-01~~	~~1996-10-01~~
111223333	120033	1996-10-01	1998-01-01
111223333	120033	1997-12-01	1998-01-01

After eliminating all non-sequenced duplicates, „merging“ all overlapping periods for the same non-temporal fact into one, results in a version free of sequenced duplicates:

2.

SSN	PCN	START_DATE	END_DATE
111223333	120033	1996-01-01	1996-10-01
111223333	120033	1996-10-01	1998-01-01

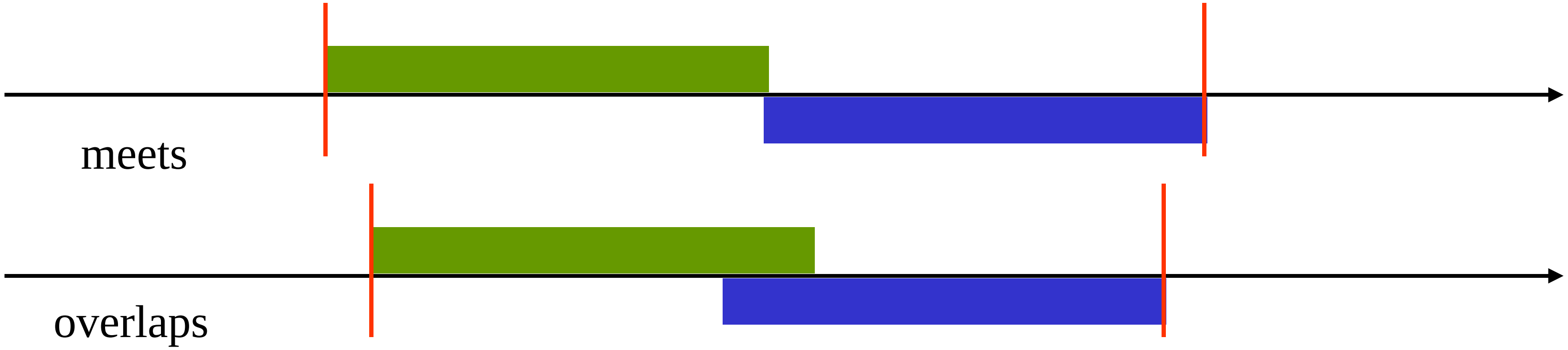
The two remaining rows have periods of validity which meet – so why not „merge“ them into a single row as well (even though this is not strictly necessary for eliminating duplicates):

3.

SSN	PCN	START_DATE	END_DATE
111223333	120033	1996-01-01	1998-01-01

Coalescing

The operation of combining two overlapping or meeting periods into a single one that comprises both (without extending any of them) is called coalescing in most research papers. (from lat. alescere = grow up, co-alescere = grow together into one)



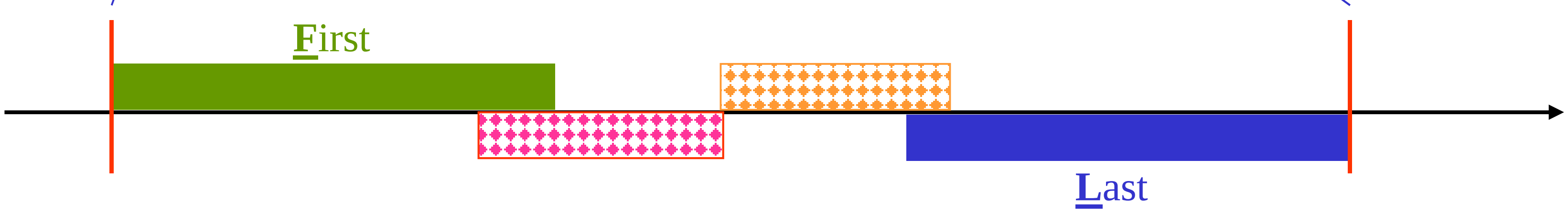
In order to reach a duplicate free (no overlapping periods for the same fact) or even a „non-redundant“ (no meeting periods) representation of a piece of history, repeated coalescing is required which continues until no further coalescing is possible any more.

This seems to require an imperative specification of sequenced duplicate freeness – but in fact a declarative specification is possible (which does not make the iteration explicit)!

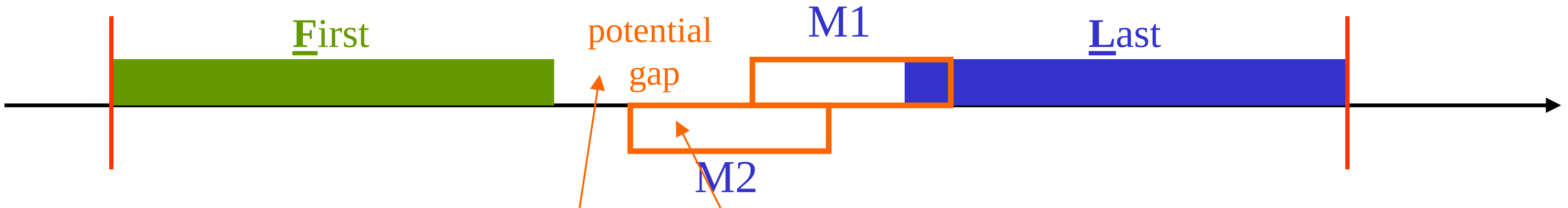
Eliminating Duplicates (3)

„Coalescing“ the INCUMBENTS table by means of a single SQL query is quite a tricky business (which took researchers many years to discover). It constructs gap-free periods of validity with maximal length for each non-temporal fact:

```
SELECT DISTINCT
            F.SSN, F.PCN,
            F.START_DATE, L.END_DATE
FROM        INCUMBENTS  AS  F,
            INCUMBENTS  AS  L
WHERE  F.START_DATE <= L.END_DATE
    AND  F.END_DATE <= L.START_DATE
    AND  F.SSN = L.SSN   AND   F.PCN = L.PCN
    AND  -- No gaps between F.END_DATE and L.START_DATE
    AND  -- Can‘t be extended further
```

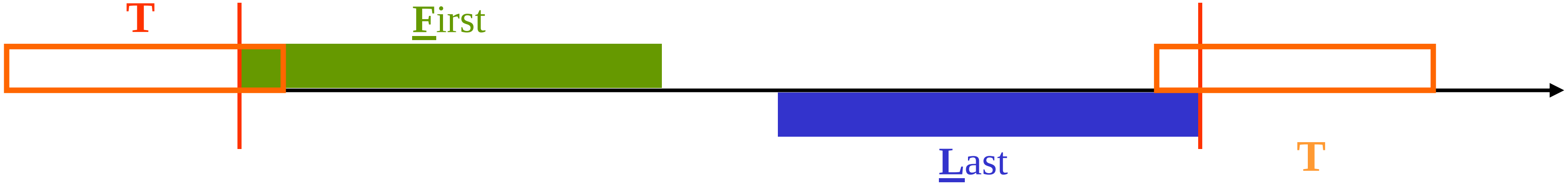


Eliminating Duplicates (4)



```
-- No gaps between F.END_DATE and L.START_DATE
NOT EXISTS
   ∀      (SELECT *
           FROM     INCUMBENTS AS M1
           WHERE  M1.SSN = F.SSN
               AND  M1.PCN = F.PCN
               AND  F.END_DATE          < M1.START_DATE
               AND  M1.START_DATE < L.START_DATE
               AND  NOT EXISTS
                    ∃        (SELECT *
                              FROM     INCUMBENTS AS M2
                              WHERE  M2.SSN = F.SSN
                                 AND  M2.PCN = F.PCN
                                 AND   M2.START_DATE  <   M1.START_DATE
                                 AND   M1.START_DATE  <= M2.END_DATE))
```

Eliminating Duplicates (5)



```
-- Can‘t be extended further
NOT EXISTS
            (SELECT *
             FROM     INCUMBENTS AS T
             WHERE  T.SSN = F.SSN
                  AND T.PCN = F.PCN
                  AND (    (T.START_DATE    <   F.START_DATE  AND
                              F.START_DATE    <= T.END_DATE)
                       OR (T.START_DATE   <= L.END_DATE      AND
                              L.END_DATE       <   T.END_DATE)))
```

T meets F
or T overlaps F

T meets^{-1} F
or T overlaps^{-1} F

Current Insertions

- Turning to modifications of historical tables now, let us assume for the time being, that only current modifications are allowed, i.e., that history cannot be changed in a transaction time table.

- If a table like INCUMBENTS were still non-temporal, all modifications would be physical changes (i.e., really performed like that), e.g.:

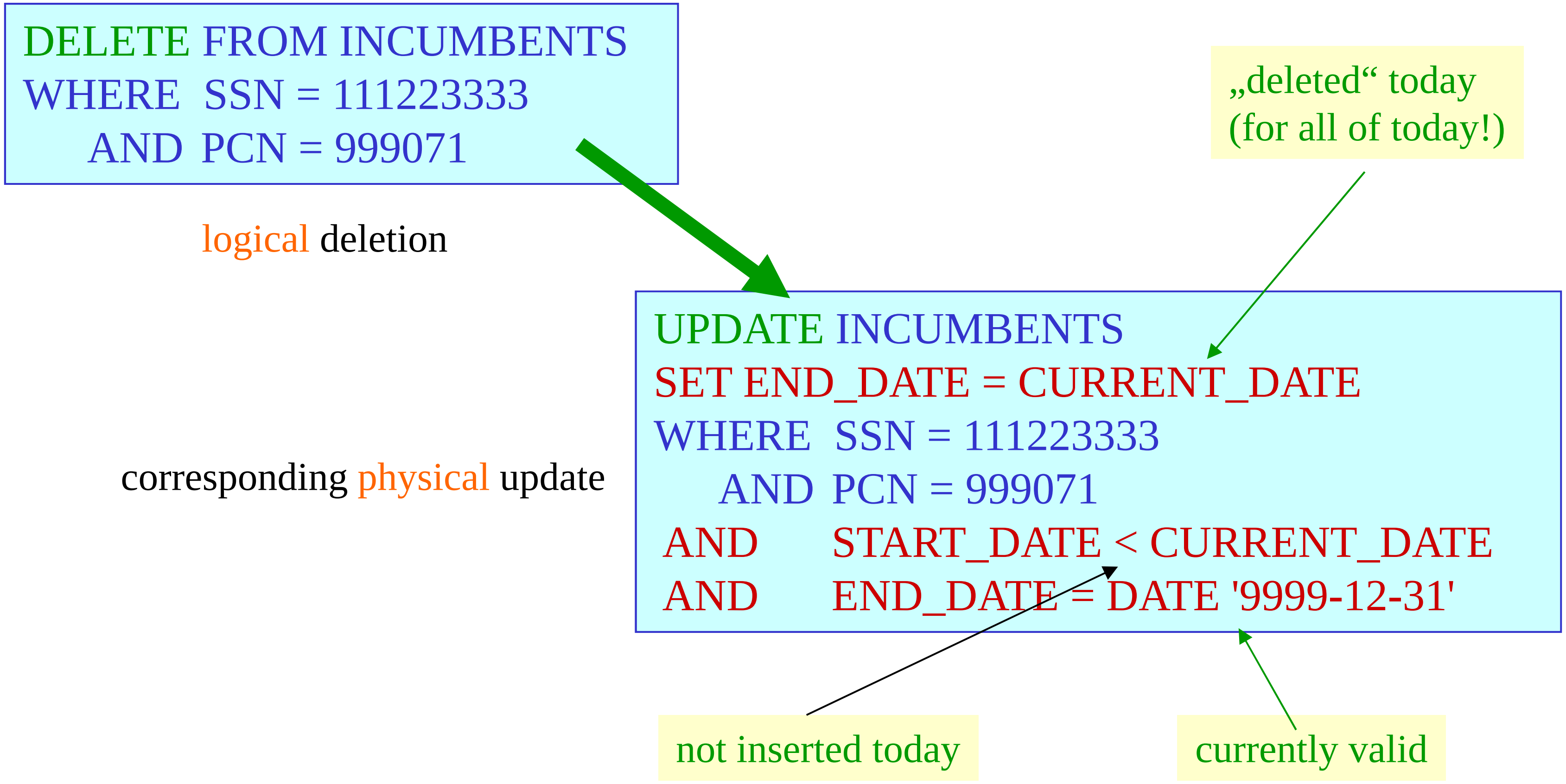
```
INSERT  INTO INCUMBENTS
VALUES (111223333, 999071)
```

- After turning INCUMBENTS into a historical one, the above insertion would become a logical insertion, which is to be implemented (as current insertion) by means of the following physical insertion, automatically adding the Start/End timestamps:

```
INSERT  INTO INCUMBENTS
VALUES (111223333, 999071, CURRENT_DATE, DATE '9999-12-31')
```

Current Deletions (1)

A (current) logical deletion, on the other hand, will have to be implemented by means of a physical update, which „closes“ the validity period of the deleted row:



```
DELETE FROM INCUMBENTS
WHERE  SSN = 111223333
     AND  PCN = 999071
```

logical deletion

corresponding physical update

```
UPDATE INCUMBENTS
SET END_DATE = CURRENT_DATE
WHERE  SSN = 111223333
     AND  PCN = 999071
 AND      START_DATE < CURRENT_DATE
 AND      END_DATE = DATE '9999-12-31'
```

„deleted“ today
(for all of today!)

not inserted today

currently valid

Current Deletions (2)

- Here, we deviate from the „translation“ of the logical deletion to a physical update as given by Snodgrass in his book (p. 183) and include the same additional condition he is using himself on p. 185 when „translating“ the deletion part of a current (logical) update:

```
UPDATE INCUMBENTS
SET END_DATE = CURRENT_DATE
WHERE  SSN = 111223333
     AND   PCN = 999071
 AND       START_DATE < CURRENT_DATE
 AND       END_DATE = DATE '9999-12-31'
```

- Thus, we avoid applying the deletion to facts being (logically) inserted on the same day as the (logical) deletion. If the fact (111223333, 999071) was inserted and subsequently deleted ***today***, we end up with the following physical state of INCUMBENTS:

SSN	**PCN**	**START_DATE**	**END_DATE**	
111223333	999071	*some day in the past*	***today***	deleted
111223333	999071	***today***	9999-12-31	inserted

- The fact (111223333, 999071) is true during all of ***today*** – we can‘t detect if the insertion was before or after the deletion, and we can‘t detect a „gap“ in validity either. This is due to the granularity DAY of the timestamps.

Current Updates (1)

A logical update such as . . .

```
UPDATE INCUMBENTS
SET       PCN = 908739
WHERE     SSN = 111223333
```

. . . corresponds to a physical insertion of the updated row(s) plus a logical deletion, i.e., a physical update of the previous version of the affected row(s) – in this order:

```
INSERT    INTO INCUMBENTS (SSN, PCN, START_DATE, END_DATE)
SELECT    DISTINCT SSN, 908739, CURRENT_DATE, DATE '9999-12-31'
FROM      INCUMBENTS
WHERE     SSN = 111223333
     AND  START_DATE < CURRENT_DATE
     AND  END_DATE = DATE '9999-12-31‘ ;

UPDATE    INCUMBENTS
SET       END_DATE = CURRENT_DATE
WHERE     SSN = 111223333
    AND   START_DATE < CURRENT_DATE
    AND   END_DATE = DATE '9999-12-31‘ ;
```

Current Updates (2)

- Again, we deviate from Snodgrass (p. 185) who doesn‘t include the END_DATE condition this time. However, without this condition „historical“ facts (the END_DATE of which is in the past) would be updated, too. This is not what is meant, obviously:

```
UPDATE  INCUMBENTS
SET     END_DATE = CURRENT_DATE
WHERE   SSN = 111223333
   AND  START_DATE < CURRENT_DATE
   AND  END_DATE = DATE '9999-12-31‘ ;
```

- The new physical state of INCUMBENTS is as follows:

SSN	PCN	START_DATE	END_DATE	
111223333	999071	*some day in the past*	***today***	old/deleted
111223333	908739	***today***	9999-12-31	new/inserted

- Again, we have both facts being true ***today*** – and again we can‘t reconstruct what happened ***today*** in which order (unless we use a finer granularity of timestamp).

SQL:2011 – A New Launch of Old Ideas

- In 2008, a (new) attempt of including temporal features into the SQL standard was launched, this time not aiming at an (overambitious) separate part of the standard (as done by previous attempts, that all failed), but at an inclusion of such features into SQL/Foundation (the main part of the standard).

- First, a new time dimension called „system time“ was included (resembling transaction time), in 2010 another dimension „application time“ (resembling valid time) was included, too.

- New modification and query syntax clauses have been added to SQL. No new syntax yet for more complex query types (e.g., no sequenced join, no coalescing)!

- A PERIOD data type for time intervals, however, was still not proposed. Periods still have to be simulated using pairs of instants (with implicit [close, open)-semantics).

- On December 15, 2011, the new SQL standard SQL:2011 was published. It‘s foundation part including the new temporal features comprises 1434 pages.

- By now, first major relational DBMS vendors are following SQL:2011 and have been including the new language features into „their SQL dialect“ (sometimes „in dialect“).

SQL:2011 vs. Previously Used Terminology in Comparison

Research Terminology	**SQL:2011 Terminology**
valid time	application time
transaction time	system time
timestamping	versioning
valid time table	application time period table
transaction time table	system-versioned table
bitemporal table	system-versioned application time period table

The following introduction to the new temporal language features of SQL:2011 have been taken from an early introductory lecture on this issue prepared and presented by a leading researcher of IBM who has been the head of the committee which proposed these extensions to the SQL standards committee. They will be continued in the next chapter (on valid resp. application time).

SQL:2011 Tutorial by Krishna Kulkarni (IBM)

WG2 N1536
WG3: KOA-046

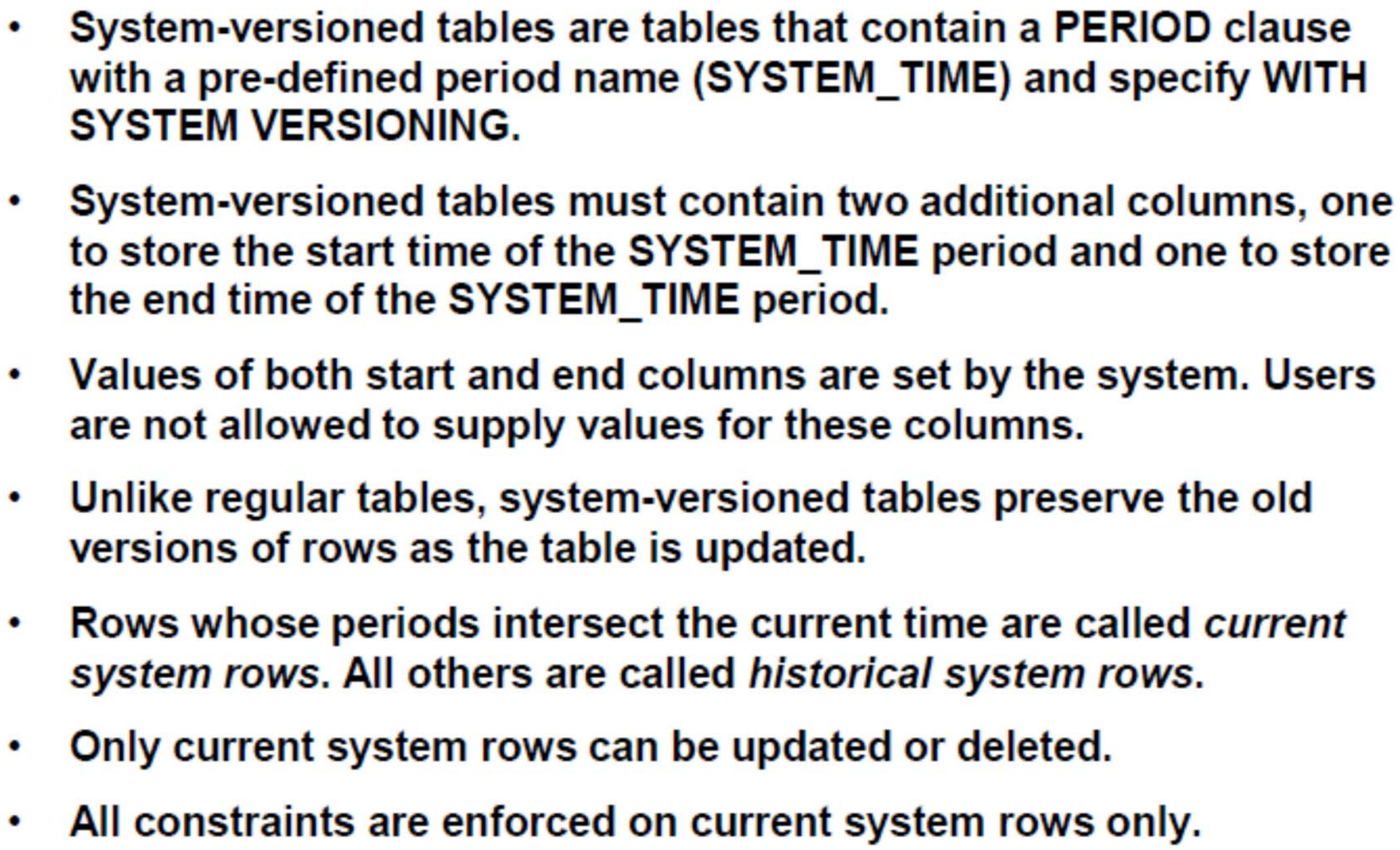
Temporal Features in SQL standard

Krishna Kulkarni,
IBM Corporation
krishnak@us.ibm.com
May 13, 2011

1

The following slides have been taken from this tutorial available online.

SQL:2011: System-Versioned Tables

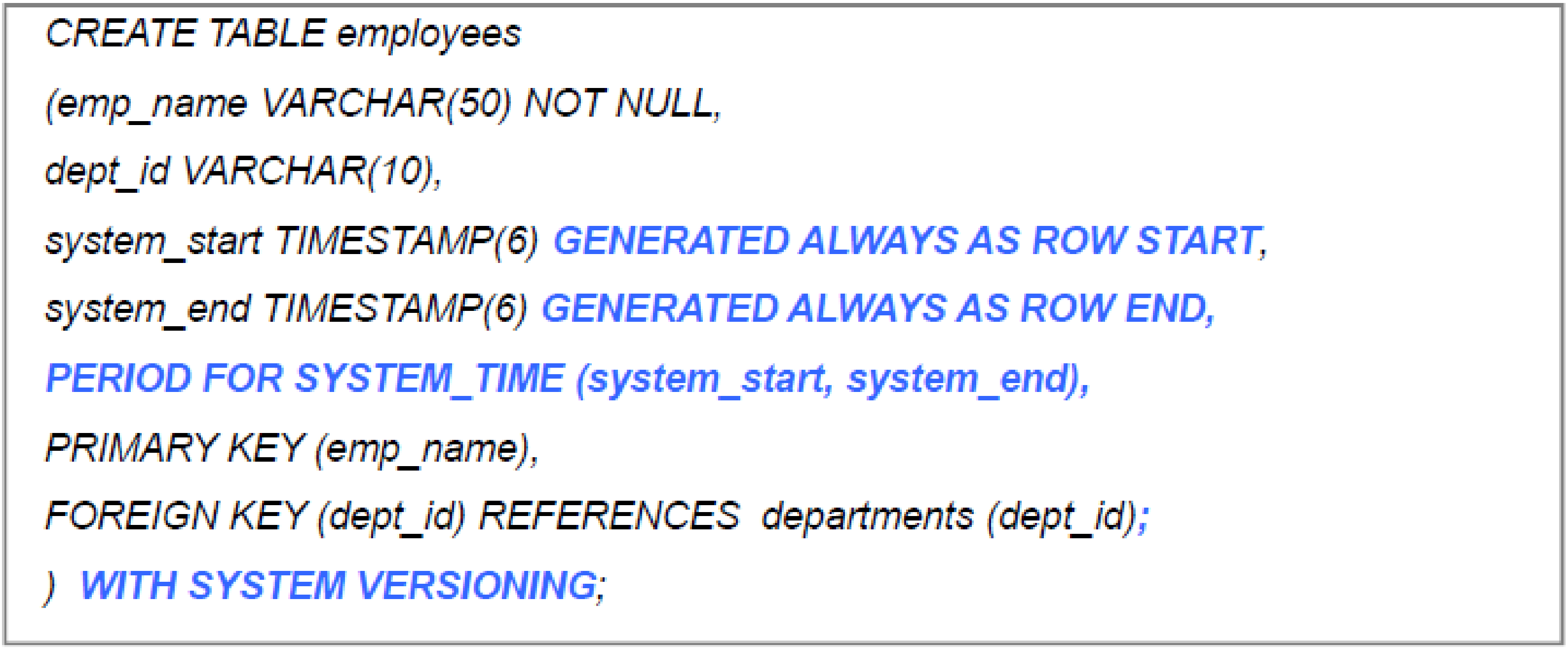


- System-versioned tables are tables that contain a **PERIOD** clause with a pre-defined period name (**SYSTEM_TIME**) and specify **WITH SYSTEM VERSIONING**.
- System-versioned tables must contain two additional columns, one to store the start time of the **SYSTEM_TIME** period and one to store the end time of the **SYSTEM_TIME** period.
- Values of both start and end columns are set by the system. Users are not allowed to supply values for these columns.
- Unlike regular tables, system-versioned tables preserve the old versions of rows as the table is updated.
- Rows whose periods intersect the current time are called *current system rows*. All others are called *historical system rows*.
- Only current system rows can be updated or deleted.
- All constraints are enforced on current system rows only.

(example from K. Kulkarni „Temporal Features in SQL Standard“)

SQL:2011: System-Versioned Tables (1)

Creating a system-versioned table:



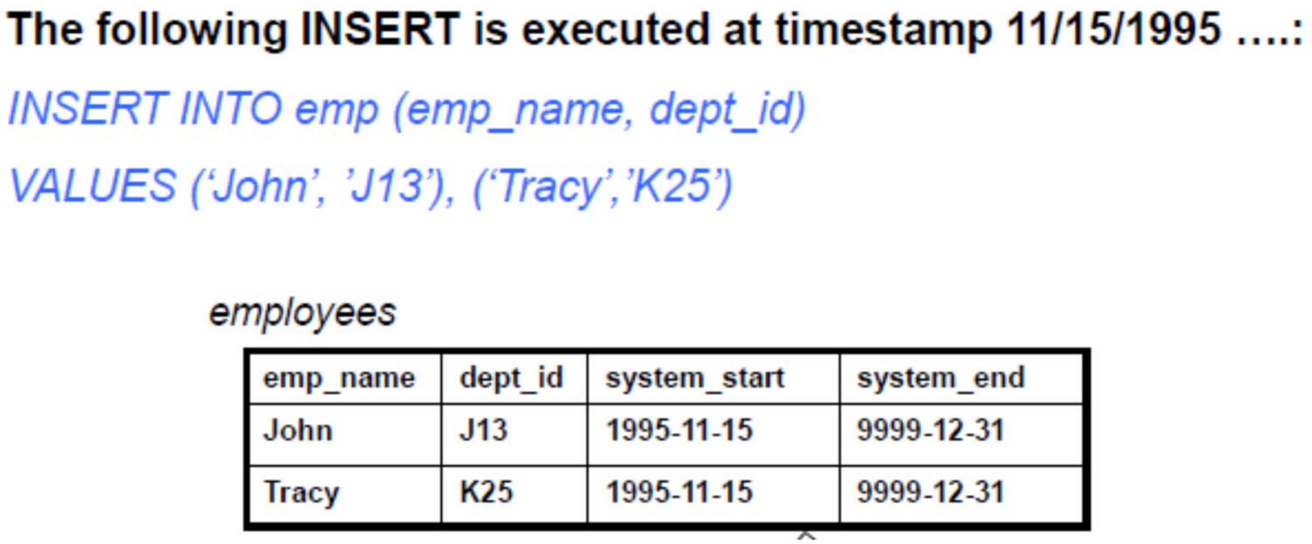
```
CREATE TABLE employees
(emp_name VARCHAR(50) NOT NULL,
dept_id VARCHAR(10),
system_start TIMESTAMP(6) GENERATED ALWAYS AS ROW START,
system_end TIMESTAMP(6) GENERATED ALWAYS AS ROW END,
PERIOD FOR SYSTEM_TIME (system_start, system_end),
PRIMARY KEY (emp_name),
FOREIGN KEY (dept_id) REFERENCES  departments (dept_id);
)  WITH SYSTEM VERSIONING;
```

(example from K. Kulkarni „Temporal Features in SQL Standard“)

SQL:2011: System-Versioned Tables (2)

Inserting rows into a system-versioned table – period values provided by the system:

current (at that time)!



The following INSERT is executed at timestamp 11/15/1995 ….:

```
INSERT INTO emp (emp_name, dept_id)
VALUES ('John', 'J13'), ('Tracy','K25')
```

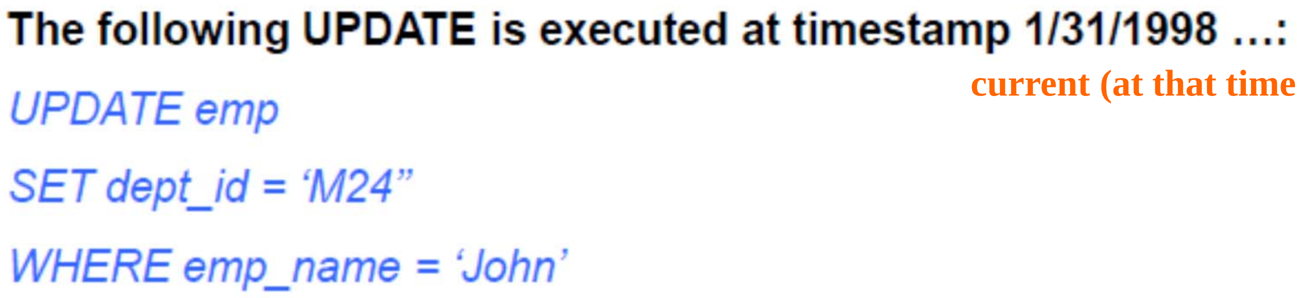
employees

emp_name	dept_id	system_start	system_end
John	J13	1995-11-15	9999-12-31
Tracy	K25	1995-11-15	9999-12-31

(example from K. Kulkarni „Temporal Features in SQL Standard“)

SQL:2011: System-Versioned Tables (3)

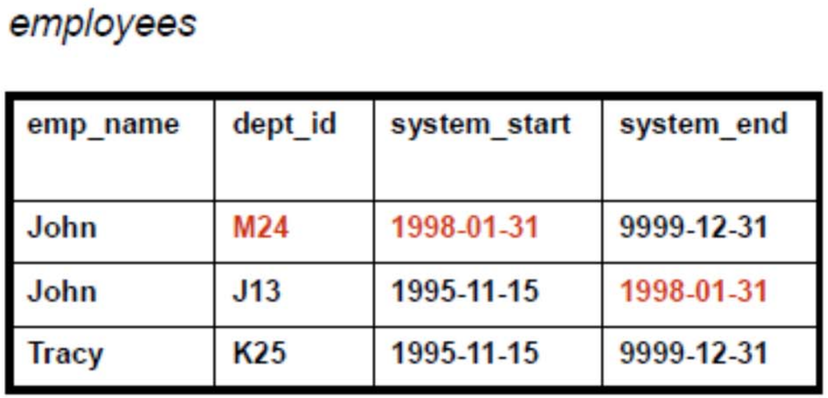
Updating fields in a system-versioned table – timestamps updated automatically:



The following UPDATE is executed at timestamp 1/31/1998 …:

```
UPDATE emp
SET dept_id = ‘M24”
WHERE emp_name = ‘John’
```

current (at that time)!



employees

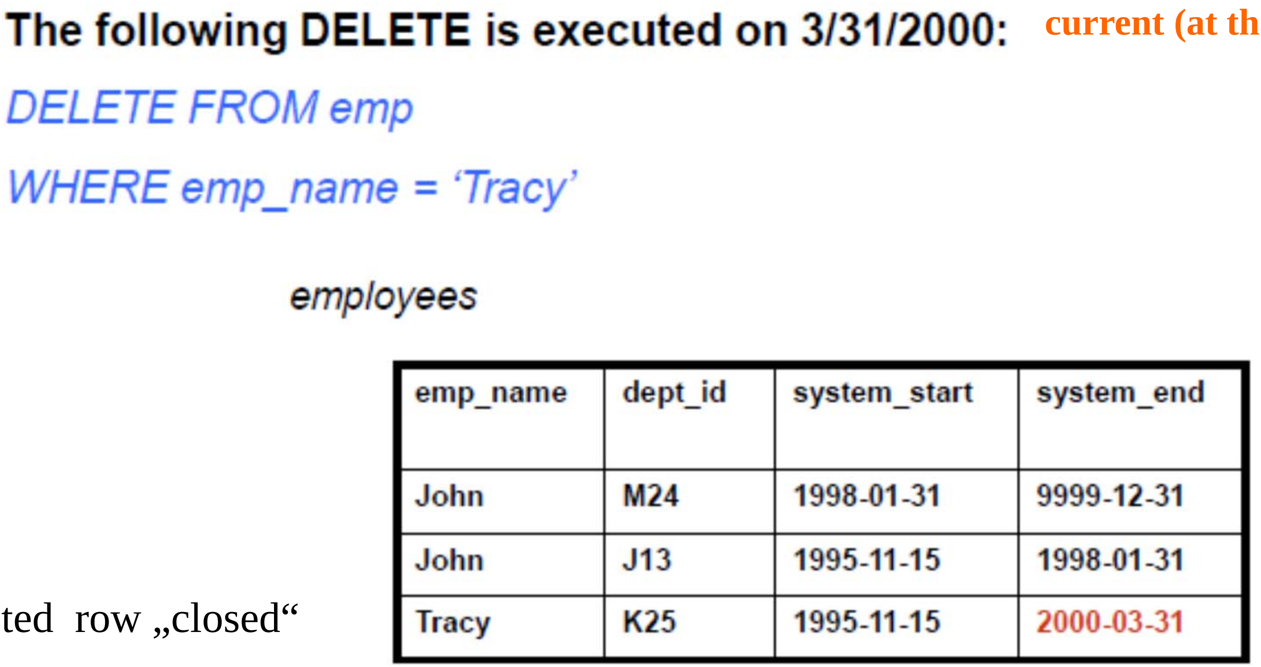
emp_name	dept_id	system_start	system_end
John	M24	1998-01-31	9999-12-31
John	J13	1995-11-15	1998-01-31
Tracy	K25	1995-11-15	9999-12-31

new value: new row
old value: row „closed“

(example from K. Kulkarni „Temporal Features in SQL Standard“)

SQL:2011: System-Versioned Tables (4)

Deleting rows from a system-versioned table – timestamps updated automatically:



The following DELETE is executed on 3/31/2000:

current (at that time)!

```
DELETE FROM emp
WHERE emp_name = ‘Tracy’
```

employees

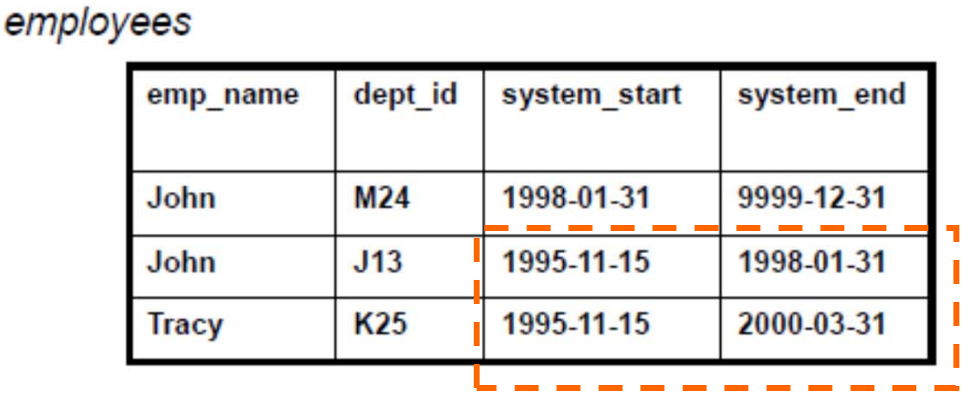
emp_name	dept_id	system_start	system_end
John	M24	1998-01-31	9999-12-31
John	J13	1995-11-15	1998-01-31
Tracy	K25	1995-11-15	2000-03-31

deleted row „closed“

(example from K. Kulkarni „Temporal Features in SQL Standard“)

SQL:2011: System-Versioned Tables (5)

Querying a system-versioned table – a system time timeslice (past) query:



employees

emp_name	dept_id	system_start	system_end
John	M24	1998-01-31	9999-12-31
John	J13	1995-11-15	1998-01-31
Tracy	K25	1995-11-15	2000-03-31

1. Which department was John in on Dec. 1, 1997?

```
SELECT Dept
FROM employees FOR SYSTEM_TIME AS OF DATE '1997-12-01'
WHERE emp_name = 'John'
```

J13

AS OF identifies timeslice queries

(example from K. Kulkarni „Temporal Features in SQL Standard“)

SQL:2011: System-Versioned Tables (6)

Querying a system-versioned table – a system time current query:

employees

emp_name	dept_id	system_start	system_end
John	M24	1998-01-31	9999-12-31
John	J13	1995-11-15	1998-01-31
Tracy	K25	1995-11-15	2000-03-31

1. Which department is John in currently?

```
SELECT Dept
FROM employees
WHERE emp_name = ‘John’
```

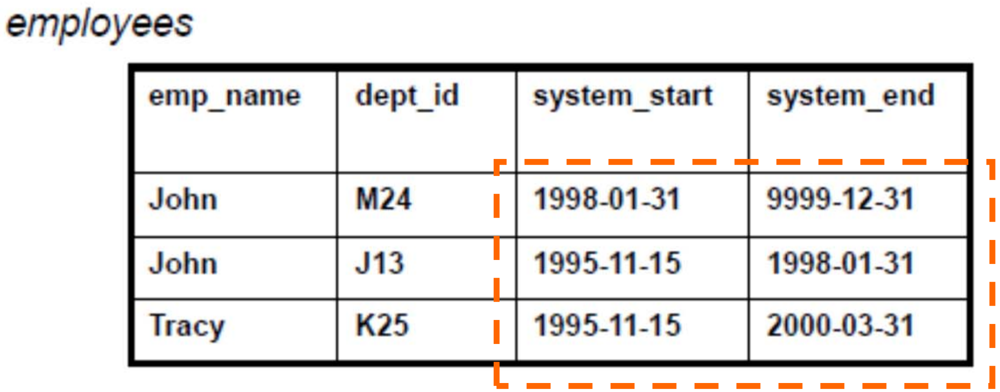
M24

No explicit mentioning of „current“! So every „normal“ SQL query is considered a system time current query.

(example from K. Kulkarni „Temporal Features in SQL Standard“)

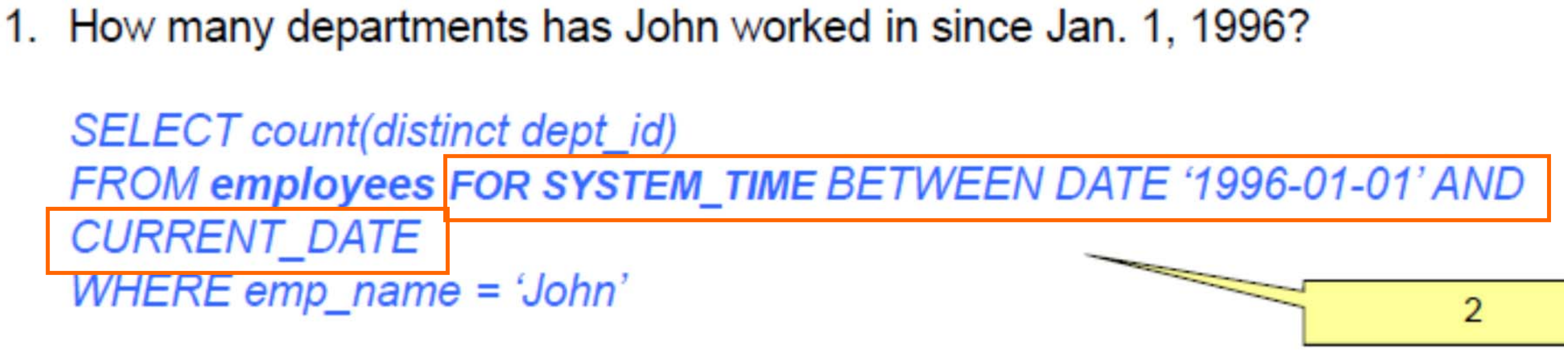
SQL:2011: System-Versioned Tables (7)

Querying a system-versioned table – a system time sequenced query:



employees

emp_name	dept_id	system_start	system_end
John	M24	1998-01-31	9999-12-31
John	J13	1995-11-15	1998-01-31
Tracy	K25	1995-11-15	2000-03-31



1. How many departments has John worked in since Jan. 1, 1996?

```
SELECT count(distinct dept_id)
FROM employees FOR SYSTEM_TIME BETWEEN DATE '1996-01-01' AND
CURRENT_DATE
WHERE emp_name = 'John'
```

2

(example from K. Kulkarni „Temporal Features in SQL Standard“)

SQL:2011: System-Versioned Tables (8)

- The new FOR SYSTEM_TIME-clause selects a „period of past data“ (possibly including the current timestamp) over which the given query is to be evaluated.

- In our example, the BETWEEN .. AND … syntax already present in SQL for arbitrary data types has been used. Unfortunately, BETWEEN .. AND … includes both, start and end value, i.e., it specifies [close, close] intervals. If this is intended for a particular query, the old syntax can be used.

- A new variant of delimiters has been introduced since SQL:2011 which corresponds to [close, open)-intervals – as required for temporal intervals, called periods:
  FROM … TO … does not include the end point!

- Nevertheless, the range of expressivity of the FOR SYSTEM_TIME clause is still rather limited! Even though it restricts the range of rows to be considered as input to the query, it does not automatically generate the syntactical „constructs“ needed for sequenced binary operators (e.g., join and difference) – these are still the job of the SQL programmer. Automatic coalescing is also not yet supported.