# Temporal Data and The Relational Model

Hugh Darwen
hugh@dcs.warwick.ac.uk
www.dcs.warwick.ac.uk/~hugh

Based on the book of the same title,
by C.J. Date, Hugh Darwen, and Nikos A. Lorentzos

summarised in C.J. Date: Introduction to Database Systems (8th edition, Addison-Wesley, 2003), Chapter 23.

for
Warwick University
CS253

## Temporal Data and The Relational Model

Authors: C.J. Date, Hugh Darwen,
Nikos A. Lorentzos

*A detailed investigation into the application of* ***interval*** *and* ***relation*** *theory to the problem of* ***temporal database*** *management*

Morgan-Kaufmann, 2002
ISBN 1-55860-855-9

*Caveat:* not about technology available anywhere today!

But **MighTyD** deserves a mention!

2

## The Book's Aims

- Describe a **foundation** for inclusion of support for **temporal data** in a **truly relational database** management system (TRDBMS)
- Focussing on problems related to data representing beliefs that hold ***throughout*** given **intervals** (usually, of time).
- Propose additional operators on **relations** and **relation variables** ("relvars") having **interval-valued attributes**.
- Propose additional constraints on **relation variables** having **interval-valued attributes**.
- All of the above to be definable in terms of existing operators and constructs.
- And explore some interesting side issues.

3

## Contents (Parts I and II)



4

## Contents (Part III)



5

## Appendixes



6

## Part I: Preliminaries

**Chapter 1:** A Review of Relational Concepts

Introduction; The running example (based on Date's familiar "suppliers and parts" database); Types; Relation values; Relation variables; Integrity constraints; Relational operators; The relational model; Exercises (as for every chapter).

**Chapter 2:** An Overview of **Tutorial D**

A relational database language devised for tutorial purposes by Date and Darwen in "Databases, Types, and The Relational Model: *The Third Manifesto*" (3rd edition, Addison-Wesley, 2005). Also used in 8th edition of Date's "Introduction to Database Systems".

Introduction; Scalar type definitions; Relational definitions; Relational expressions; Relational assignments; Constraint definitions; Exercises.

7

## Chapter 3: Time and the Database

Introduction

Timestamped propositions

E.g. "Supplier S1 was under contract throughout the period from 1/9/1999 (and not immediately before that date) until 31/5/2002 (and not immediately after that date)."

"Valid time" vs. "transaction time"

Some fundamental questions:

Introduction of ***quantisation*** and its consequences.

8

# CHAPTER 4: What is The Problem?

9

## Example: Current State Only

"Suppliers and Shipments"

S

Predicate: "Supplier S# is under contract"

| S# |
|---|
| S1 |
| S2 |
| S3 |
| S4 |
| S5 |

SP

Predicate: "Supplier S# is able to supply part P#"

| S# | P# |
|---|---|
| S1 | P1 |
| S1 | P2 |
| S1 | P3 |
| S1 | P4 |
| S1 | P5 |
| S1 | P6 |
| S2 | P1 |
| S2 | P2 |
| S3 | P2 |
| S4 | P2 |
| S4 | P4 |
| S4 | P5 |

Consider queries: *Which suppliers can supply something? Which suppliers cannot supply anything?*

10

## "Semitemporalising"

S_SINCE

Predicate: "Supplier S# has been under contract since day SINCE"

| S# | SINCE |
|---|---|
| S1 | *d04* |
| S2 | *d07* |
| S3 | *d03* |
| S4 | *d04* |
| S5 | *d02* |

SP_SINCE

Predicate: "Supplier S# has been able to supply part P# since day SINCE"

| S# | P# | SINCE |
|---|---|---|
| S1 | P1 | *d04* |
| S1 | P2 | *d05* |
| S1 | P3 | *d09* |
| S1 | P4 | *d05* |
| S1 | P5 | *d04* |
| S1 | P6 | *d06* |
| S2 | P1 | *d08* |
| S2 | P2 | *d09* |
| S3 | P2 | *d08* |
| S4 | P2 | *d06* |
| S4 | P4 | *d04* |
| S4 | P5 | *d05* |

Consider queries: *Since when has supplier S# been able to supply anything?* (Not too difficult) *Since when has supplier S# been unable to supply anything?* (Impossible)

11

## "Fully temporalising" (try 1)

S_FROM_TO

Predicate: "Supplier S# was under contract from day FROM to day TO."

| S# | FROM | TO |
|---|---|---|
| S1 | *d04* | *d10* |
| S2 | *d02* | *d04* |
| S2 | *d07* | *d10* |
| S3 | *d03* | *d10* |
| S4 | *d04* | *d10* |
| S5 | *d02* | *d10* |

SP_FROM_TO

Predicate: "Supplier S# was able to supply part P# from day FROM to day TO."

| S# | P# | FROM | TO |
|---|---|---|---|
| S1 | P1 | *d04* | *d10* |
| S1 | P2 | *d05* | *d10* |
| S1 | P3 | *d09* | *d10* |
| S1 | P4 | *d05* | *d10* |
| S1 | P5 | *d04* | *d10* |
| S1 | P6 | *d06* | *d10* |
| S2 | P1 | *d08* | *d10* |
| S2 | P1 | *d02* | *d04* |
| S2 | P2 | *d08* | *d10* |
| S2 | P2 | *d03* | *d03* |
| S3 | P2 | *d09* | *d10* |
| S4 | P2 | *d06* | *d09* |
| S4 | P4 | *d04* | *d08* |
| S4 | P5 | *d05* | *d10* |

Consider queries: *During which times was supplier S# able to supply anything?* (Very difficult) *During which times was supplier S# unable to supply anything?* (Very difficult)

12

## Required Constraints

S_FROM_TO

| S# | FROM | TO |
|---|---|---|
| S1 | *d04* | *d10* |
| S2 | *d02* | *d04* |
| S2 | *d07* | *d10* |
| S3 | *d03* | *d10* |
| S4 | *d04* | *d10* |
| S5 | *d02* | *d10* |

Same supplier can't be under contract during distinct but overlapping or abutting intervals.

SP_FROM_TO

| S# | P# | FROM | TO |
|---|---|---|---|
| S1 | P1 | *d04* | *d10* |
| S1 | P2 | *d05* | *d10* |
| S1 | P3 | *d09* | *d10* |
| S1 | P4 | *d05* | *d10* |
| S1 | P5 | *d04* | *d10* |
| S1 | P6 | *d06* | *d10* |
| S2 | P1 | *d08* | *d10* |
| S2 | P1 | *d02* | *d04* |
| S2 | P2 | *d08* | *d10* |
| S2 | P2 | *d03* | *d03* |
| S3 | P2 | *d09* | *d10* |
| S4 | P2 | *d06* | *d09* |
| S4 | P4 | *d04* | *d08* |
| S4 | P5 | *d05* | *d10* |

Same supplier can't be able to supply same part during distinct but overlapping or abutting intervals

**These are *very* difficult!**

13

# CHAPTER 5: Intervals

14

## "Fully temporalising" (try 2)

S_DURING

| S# | DURING |
|---|---|
| S1 | [*d04*:*d10*] |
| S2 | [*d02*:*d04*] |
| S2 | [*d07*:*d10*] |
| S3 | [*d03*:*d10*] |
| S4 | [*d04*:*d10*] |
| S5 | [*d02*:*d10*] |

Introduction of ***interval types*** and their ***point types***.

SP_DURING

| S# | P# | DURING |
|---|---|---|
| S1 | P1 | [*d04*:*d10*] |
| S1 | P2 | [*d05*:*d10*] |
| S1 | P3 | [*d09*:*d10*] |
| S1 | P4 | [*d05*:*d10*] |
| S1 | P5 | [*d04*:*d10*] |
| S1 | P6 | [*d06*:*d10*] |
| S2 | P1 | [*d08*:*d10*] |
| S2 | P1 | [*d02*:*d04*] |
| S2 | P2 | [*d08*:*d10*] |
| S2 | P2 | [*d03*:*d03*] |
| S3 | P2 | [*d09*:*d10*] |
| S4 | P2 | [*d06*:*d09*] |
| S4 | P4 | [*d04*:*d08*] |
| S4 | P5 | [*d05*:*d10*] |

Here, the type of the DURING attributes is perhaps named **INTERVAL_DATE** (its point type being **DATE**).

A point type requires a successor function - in this case **NEXT_DATE ( *d* )**. This is based on the *scale* of the point type.

15

# CHAPTER 6: Operators on Intervals

16

## Interval Selectors

In **Tutorial D**, we make the type name part of the operator name. E.g.:

INTERVAL_INTEGER ( [1:10] )

Note special syntax for denoting bounds. Square bracket denotes a closed bound, round one an open bound. Thus:

INTERVAL_INTEGER ( [1:10] ) =
INTERVAL_INTEGER ( (0:10] ) =
INTERVAL_INTEGER ( [1:11) ) =
INTERVAL_INTEGER ( (0:11) )

17

## Monadic Operators on Intervals

For a given interval, *i*:

PRE ( *i* ) gives open begin bound
BEGIN ( *i* ) gives closed begin bound
END ( *i* ) gives closed end bound
POST ( *i* ) gives open end bound

COUNT ( *i* ) gives length (number of points)

18

## Comparisons of Two Intervals

For given intervals, *i1* and *i2*:

*i1* = *i2*
*i1* MEETS *i2*
*i1* OVERLAPS *i2*
*i1* SUCCEEDS *i2*
*i1* PRECEDES *i2*
*i1* ⊆ *i2*

*i1* BEGINS *i2*
*i1* ENDS *i2*

Allen's operators
(James F. Allen, 1983)

Allen uses DURING for ⊆

Allen uses STARTS and ENDS

*i1* ⊇ *i2*
*i1* ⊂ *i2*
*i1* ⊃ *i2*

*i1* MERGES *i2*

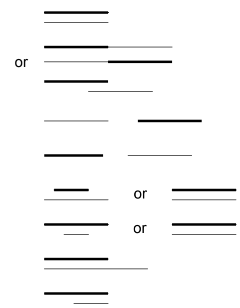
Added by Date, Darwen, Lorentzos

MERGES = MEETS OR OVERLAPS

19

## Some Pictorial Definitions

*i1* = *i2*
*i1* MEETS *i2* (or)
*i1* OVERLAPS *i2*
*i1* SUCCEEDS *i2*
*i1* PRECEDES *i2*
*i1* ⊆ *i2* (or)
*i1* ⊇ *i2* (or)
*i1* BEGINS *i2*
*i1* ENDS *i2*




20

## More Dyadic Operators

Membership test:

*p* ∈ *i1* or *p* IN *i1* (where *p* is a point)

Dyadic operators that return intervals:

*i1* UNION *i2*

*i1* INTERSECT *i2*

*i1* MINUS *i2*

Defined only for cases where the result is a single, nonempty* interval.

* empty intervals, such as INTERVAL_INTEGER ([1:1)), are not supported at all!

21

# CHAPTER 7: The COLLAPSE and EXPAND Operators

22

## Sets of Intervals

Let *SI1* and *SI2* be sets of intervals—e.g., {[1:2], [4:7], [6:9]}

We define an equivalence relationship:

*SI1* ≡ *SI2* iff every point in an interval in *SI1* is a point in some interval in *SI2*, and vice versa.

Under this equivalence relationship we then define two canonical forms: ***collapsed form*** and ***expanded form***.

In each of these forms, no point appears more than once.

23

## Collapsed Form

No two elements, *i1* and *i2* (*i1*≠*i2*) are such that *i1* MERGES *i2*.

So the collapsed form of {[1:2], [4:7], [6:9]} is {[1:2], [4:9]}.

24

## Expanded Form

Every element is a ***unit interval***
(i.e., consists of a single point)

So the expanded form of {[1:2], [4:7], [6:9]}
is {[1:1], [2:2], [4:4], [5:5], [6:6], [7:7], [8:8], [9:9]}.

25

## COLLAPSE and EXPAND

Let *SI* be a set of intervals.

Then:
COLLAPSE(*SI*) denotes the collapsed form of *SI*.
EXPAND(*SI*) denotes the expanded form of *SI*.

These operators are handy for definitional purposes (as we shall see) but are not required to exist in the database language.

26

# CHAPTER 8: The PACK and UNPACK Operators

27

## Packed Form and Unpacked Form

Canonical forms for relations with one or more interval-valued attributes.

Based on collapsed and expanded forms.

Both forms avoid redundancy ("saying the same thing" more than once).

28

## Packed Form

SD_PART

| S# | DURING |
|---|---|
| S2 | [*d02*:*d04*] |
| S2 | [*d03*:*d05*] |
| S4 | [*d02*:*d05*] |
| S4 | [*d04*:*d06*] |
| S4 | [*d09*:*d10*] |

PACK SD_PART ON (DURING) →

Packed form of SD_PART "on DURING":

| S# | DURING |
|---|---|
| S2 | [*d02*:*d05*] |
| S4 | [*d02*:*d06*] |
| S4 | [*d09*:*d10*] |

29

## Unpacked Form

SD_PART

| S# | DURING |
|---|---|
| S2 | [*d02*:*d04*] |
| S2 | [*d03*:*d05*] |
| S4 | [*d02*:*d05*] |
| S4 | [*d04*:*d06*] |
| S4 | [*d09*:*d10*] |

UNPACK SD_PART ON (DURING) →

Unpacked form of SD_PART "on DURING":

| S# | DURING |
|---|---|
| S2 | [*d02*:*d02*] |
| S2 | [*d03*:*d03*] |
| S2 | [*d04*:*d04*] |
| S2 | [*d05*:*d05*] |
| S4 | [*d02*:*d02*] |
| S4 | [*d03*:*d03*] |
| S4 | [*d04*:*d04*] |
| S4 | [*d05*:*d05*] |
| S4 | [*d06*:*d06*] |
| S4 | [*d09*:*d09*] |
| S4 | [*d10*:*d10*] |

30

## Properties of PACK and UNPACK

Packing and unpacking on no attributes:
- Important degenerate cases
- Each yields its input relation

Unpacking on several attributes:
- UNPACK *r* ON (*a1, a2*) ≡
  UNPACK (UNPACK *r* ON *a1*) ON *a2* ≡
  UNPACK (UNPACK *r* ON *a2*) ON *a1*

Packing on several attributes:
- PACK *r* ON (*a1, a2*) ≡
  PACK (PACK (UNPACK *r* ON (a1,a2)) ON *a1*) ON *a2*
  ***not:*** PACK (PACK(UNPACK *r* ON (*a1*,*a2*)) ON *a2)* ON *a1*
  ***and not:*** PACK (PACK *r* ON *a1*) ON *a2*
- Although redundancy is eliminated, result can be of greater cardinality than *r*.

31

# CHAPTER 9: Generalizing the Relational Operators

32

## Tutorial D's Relational Operators

UNION
MATCHING
NOT MATCHING
restriction (WHERE)
projection ({…})
JOIN
EXTEND
SUMMARIZE
etc.

New syntax for invoking each operator:

USING ( *ACL* ) ◄ *rel op inv* ►

where *ACL* is an attribute-name commalist and *rel op inv* an invocation of a relational operator.

Common principle:
1. Unpack the operand(s) on *ACL*
2. Evaluate rel op inv on unpacked forms.
3. Pack result of 2. on *ACL*

33

## USING Example 1

USING ( DURING ) ◄ SP_DURING { S#, DURING } ►

gives (S#, DURING) pairs such that supplier S# was able to supply some part throughout the interval DURING.

We call this "U_project".

U_project is an example of what we call a "U_ operator".

Other examples are U_JOIN, U_UNION, U_restrict, etc.

34

## Example 2: U_NOT MATCHING

USING ( DURING )
◄ S_DURING NOT MATCHING SP_DURING ►

gives (S#, DURING) pairs such that supplier S# was under contract but unable to supply any part throughout the interval DURING.

*Note:* We have now solved the two query problems mentioned in Chapter 4, "What's the Problem?"

35

## Example 3: U_SUMMARIZE

USING ( DURING )
◄ SUMMARIZE SP_DURING
PER ( S_DURING { S#, DURING } )
ADD COUNT AS NO_OF_PARTS ►

gives (S#, NO_OF_PARTS, DURING) triples such that supplier S# was able to supply NO_OF_PARTS parts throughout the interval DURING.

Temporal counterpart of:

SUMMARIZE SP PER ( S { S# } )
ADD COUNT AS NO_OF_PARTS

36

## U_SUMMARIZE is Interesting (1)

```
USING ( DURING )
◄SUMMARIZE SP_DURING
  PER ( S_DURING { DURING } )
  ADD COUNT AS NO_OF_PARTS ►
```

- note lack of S# from PER relation
- gives (NO_OF_PARTS, DURING) pairs such that NO_OF_PARTS parts were available *from some supplier* throughout the interval DURING.

37

## U_SUMMARIZE is Interesting (2)

```
USING ( DURING )
◄SUMMARIZE SP_DURING
  PER ( S_DURING { S# } )
  ADD COUNT AS NO_OF_CASES ►
```

- note lack of DURING from PER relation
- gives (S#, NO_OF_CASES) pairs such that there are NO_OF_CASES distinct cases of S# being able to supply some part on some date.

38

# CHAPTER 10: Database Design

39

## Contents

Chapter 10: Database Design



At last, we focus on specifically temporal issues!

40

## Current Relvars Only

SSSC

| <u>S#</u> | SNAME | STATUS | CITY |
|---|---|---|---|
| S1 | Smith | 20 | London |
| S2 | Jones | 10 | Paris |
| S3 | Blake | 30 | Paris |
| S4 | Clark | 20 | London |
| S5 | Adams | 30 | Athens |

SP

| <u>S#</u> | <u>P#</u> |
|---|---|
| S1 | P1 |
| S1 | P2 |
| S1 | P3 |
| S1 | P4 |
| S1 | P5 |
| S1 | P6 |
| S2 | P1 |
| S2 | P2 |
| S3 | P2 |
| S4 | P2 |
| S4 | P4 |
| S4 | P5 |

Note: keys indicated by underlining attribute names

41

## Semitemporalizing SSSC (try 1)

SSSC

| <u>S#</u> | SNAME | STATUS | CITY | SINCE |
|---|---|---|---|---|
| S1 | Smith | 20 | London | *d04* |
| S2 | Jones | 10 | Paris | *d05* |
| S3 | Blake | 30 | Paris | *d02* |
| S4 | Clark | 20 | London | *d09* |
| S5 | Adams | 30 | Athens | *d09* |

Problem: SINCE gives date of last update for that supplier.
So we cannot tell:
since when a given supplier's STATUS has held, or
since when a given supplier's CITY has held, or
since when a given supplier's NAME has held, or even
since when a given supplier has been under contract.

42

## Semitemporalizing SSSC (try 2)

```
VAR S_SINCE
  BASE RELATION
  { S# S#,            S#_SINCE      DATE,
    SNAME CHAR,   SNAME_SINCE   DATE,
    STATUS INT,     STATUS_SINCE  DATE,
    CITY  CHAR,     CITY_SINCE      DATE }
  KEY { S# } ;
```

Predicate:
Supplier S# has been under contract since S#_SINCE, has been named NAME since NAME_SINCE, has had status STATUS since STATUS_SINCE and has been located in city CITY since CITY_SINCE.

But we clearly cannot develop a fully temporalized counterpart on similar lines!

43

## Fully Temporalizing SSSC

```
VAR S_DURING
  BASE RELATION
  { S# S#,
    DURING INTERVAL_DATE }
  KEY { S#, DURING } ;
```

Predicate: Supplier S# was under contract throughout DURING and neither immediately before nor immediately after DURING.

```
VAR S_NAME_DURING
  BASE RELATION
  { S# S#,
    SNAME CHAR,
    DURING INTERVAL_DATE }
  KEY { S#, DURING } ;
```

Predicate: Supplier S# was named SNAME throughout DURING and neither immediately before nor immediately after DURING.

And so on. We call this process *vertical decomposition*.

44

## Sixth Normal Form (6NF)

Recall: A relvar *R* is in 5NF iff every nontrivial join dependency that is satisfied by *R* is implied by a candidate key of *R*.

A relvar *R* is in 6NF iff *R* satisfies no nontrivial join dependencies at all (in which case *R* is sometimes said to be *irreducible*).

SSSC and SSSC_SINCE are in 5NF but not 6NF (which is not needed).

S_DURING, SNAME_DURING and so on are in 6NF, thus allowing each of the supplier properties NAME, CITY and STATUS, ***which vary independently of each other over time***, to have its own recorded history (by supplier).

45

## “Circumlocution” and 6NF

| S# | NAME | STATUS | DURING |
|---|---|---|---|
| S1 | Smith | 20 | [*d01*:*d06*] |
| S1 | Smith | 30 | [*d07*:*d09*] |

Note S1 named Smith throughout [d01:d09], split across tuples.
We call this undesirable phenomenon *circumlocution*.
Decompose to 6NF, using U_projection:

| S# | NAME | DURING |
|---|---|---|
| S1 | Smith | [*d01*:*d09*] |

| S# | STATUS | DURING |
|---|---|---|
| S1 | 20 | [*d01*:*d06*] |
| S1 | 30 | [*d07*:*d09*] |

46

## “The Moving Point NOW”

We reject any notion of a special marker, NOW, as an interval bound. (It is a variable, not a value. Its use would be as much a departure from the Relational Model as NULL is!)
(We reject the use of NULL too, obviously.)

If current state is to be recorded, along with history, in S_DURING, S_NAME_DURING, S_STATUS_DURING and S_CITY_DURING, then we have a choice of evils:

- guess when, in the future, current state will change
- assume current state will hold until the end of time

Better instead to use *horizontal decomposition*

47

## Horizontal Decomposition

A very loose term! Components do not have exactly the same structure:

1. The current state component (S_SINCE)
2. The past history component, with DURING in place of S_SINCE’s SINCE.

The past history component is then vertically decomposed as already shown, giving S_DURING, S_NAME_DURING, S_STATUS_DURING, and S_CITY_DURING.

Having accepted the occasional (perhaps frequent) inevitability of vertical and horizontal decomposition, we need to consider the consequences for constraints ...

48

# CHAPTER 11: Integrity Constraints I



## Candidate Keys and Related Constraints

Example database:

S_SINCE { S#, S#_SINCE, STATUS, STATUS_SINCE }
SP_SINCE { S#, P#, SINCE }
S_DURING { S#, DURING }
S_STATUS_DURING { S#, STATUS, DURING }
SP_DURING { S#, P#, DURING }

We first examine three distinct problems:
- The redundancy problem
- The circumlocution problem
- The contradiction problem

A fourth problem, concerning "density", will come later.



## The Redundancy Problem

Consider:

S_STATUS_DURING { S#, STATUS, DURING }

The declared key, { S#, DURING } doesn't prevent this:

| S# | STATUS | DURING |
|---|---|---|
| S4 | 25 | [*d05* : *d06*] |
| S4 | 25 | [*d06* : *d07*] |

S4 shown twice as having status 25 on day 6.

Avoided in the packed form of S_STATUS_DURING.



## The Circumlocution Problem

Still considering:

S_STATUS_DURING { S#, STATUS, DURING }

The declared key, {S#, DURING } doesn't prevent this:

| S# | STATUS | DURING |
|---|---|---|
| S4 | 25 | [*d05* :*d05*] |
| S4 | 25 | [*d06* :*d07*] |

Longwinded way of saying that S4 has status 25 from day 5 to day 7.

Also avoided in the packed form of S_STATUS_DURING.



## Solving The Redundancy and Circumlocution Problems

VAR S_STATUS_DURING RELATION
{ S# S#,
STATUS INT, DURING INTERVAL_DATE }
KEY { S#, DURING }
**PACKED ON (** DURING **)** ;

**PACKED ON (** DURING **)** causes an update to be rejected if acceptance would result in
S_STATUS_DURING ≠ PACK S_STATUS_DURING ON ( DURING )

This kills two birds with one stone. We see no compelling reason for distinct shorthands to separate the two required constraints.



## The Contradiction Problem

Still considering:

S_STATUS_DURING { S#, STATUS, DURING }

The declared key, { S#, DURING } and PACKED ON ( DURING ) don't prevent this:

| S# | STATUS | DURING |
|---|---|---|
| S4 | 25 | [*d04* :*d06*] |
| S4 | 10 | [*d05* :*d07*] |

S4 has two statuses on days 5 and 6.

Easily avoidable in the unpacked form of S_STATUS_DURING!

## Solving The Contradiction Problem

VAR S_STATUS_DURING RELATION
{ S# S#,
STATUS CHAR, DURING INTERVAL_DATE }
KEY { S#, DURING }
PACKED ON ( DURING )
**WHEN UNPACKED ON (** DURING **)**
**THEN KEY {** S#, DURING **}** ;

**WHEN UNPACKED_ON (** DURING **) THEN KEY {** S#, DURING } causes an update to be rejected if acceptance would result in failure to satisfy a uniqueness constraint on { S#, DURING } in the result of UNPACK S_STATUS_DURING ON ( DURING ).

55

## WHEN / THEN without PACKED ON

Example (presidential terms):

TERM

| DURING | PRESIDENT |
|---|---|
| [1974 : 1976] | Ford |
| [1977 : 1980] | Carter |
| [1981 : 1984] | Reagan |
| [1985 : 1988] | Reagan |
| [1993 : 1996] | Clinton |
| [1997 : 2000] | Clinton |
| [2009 : 2012] | Obama |

PACKED ON ( DURING ) not desired because it would lose distinct consecutive terms by same president (e.g., Reagan and Clinton)
But we can't have two presidents at same time!
Perhaps not good design (better to include a TERM# attribute?) but we don't want to legislate against it.

56

## Neither WHEN / THEN nor PACKED ON

Example (measures of inflation):

INFLATION

| DURING | PERCENTAGE |
|---|---|
| [*m01*:*m03*] | 18 |
| [*m04*:*m06*] | 20 |
| [*m07*:*m09*] | 20 |
| [*m07*:*m07*] | 25 |
| .......... | .. |
| [*m01*:*m12*] | 20 |

But the predicate for this is not:

"Inflation was at PERCENTAGE throughout the interval DURING"

but rather, perhaps:

"Inflation was measured to be PERCENTAGE over the interval DURING"

57

## WHEN / THEN and PACKED ON both required

VAR S_STATUS_DURING RELATION
{ S# S#,
STATUS CHAR, DURING INTERVAL_DATE }
USING ( DURING ) ◄ KEY { S#, DURING } ► ;

USING ( *ACL* ) ◄ KEY { *K* } ►, where *K* includes *ACL*, is shorthand for:
WHEN UNPACKED ON ( *ACL* )
THEN KEY { *K* }
PACKED ON (*ACL* )
KEY { *K* }

(KEY { *K* } is implied by WHEN/THEN + PACKED ON anyway)

We call this constraint a "U_key" constraint.

58

# CHAPTER 12: Integrity Constraints II

59

## General Constraints

Example database is still:

S_SINCE { S#, S#_SINCE, STATUS, STATUS_SINCE }
SP_SINCE { S#, P#, SINCE }
S_DURING { S#, DURING }
S_STATUS_DURING { S#, STATUS, DURING }
SP_DURING { S#, P#, DURING }

with added U_keys. But more constraints are needed.

We examine nine distinct requirements, in three groups of three. In each group, one requirement relates to **redundancy** (and sometimes also to **contradiction**), one to **circumlocution** and one to **denseness**.

60

## Requirement Group 1

Requirement R1:
If the database shows supplier S*x* as being under contract on day *d*, then it must contain exactly one tuple that shows that fact.
Note: avoiding *redundancy*

Requirement R2:
If the database shows supplier S*x* as being under contract on days *d* and *d*+1, then it must contain exactly one tuple that shows that fact.
Note: avoiding *circumlocution*

Requirement R3:
If the database shows supplier S*x* as being under contract on day *d*, then it must also show supplier S*x* as having some status on day *d*.
Note: to do with *denseness*

61

## Requirement Group 2

Requirement R4:
If the database shows supplier S*x* as having some status on day *d*, then it must contain exactly one tuple that shows that fact.
Note: avoiding *redundancy* and *contradiction*

Requirement R5:
If the database shows supplier S*x* as having status *s* on days *d* and *d*+1, then it must contain exactly one tuple that shows that fact.
Note: avoiding *circumlocution*

Requirement R6:
If the database shows supplier S*x* as having some status on day *d*, then it must also show supplier S*x* as being under contract on day *d*.
Note: to do with *denseness*

62

## Requirement Group 3

Requirement R7:
If the database shows supplier S*x* as being able to supply part P*y* on day *d*, then it must contain exactly one tuple that shows that fact.
Note: avoiding *redundancy*

Requirement R8:
If the database shows supplier S*x* as being able to supply part P*y* on days *d* and *d*+1, then it must contain exactly one tuple that shows that fact.
Note: avoiding *circumlocution*

Requirement R9:
If the database shows supplier S*x* as being able to supply some part on day *d*, then it must also show supplier S*x* as being under contract on day *d*.
Note: to do with *denseness*

63

## Meeting the Nine Requirements (a): current relvars only

```
S_SINCE { S#, S#_SINCE, STATUS, STATUS_SINCE }
          KEY { S# }

CONSTRAINT CR6 IS_EMPTY
     ( S_SINCE WHERE STATUS_SINCE < S#_SINCE )

SP_SINCE { S#, P#, SINCE }
      KEY { S#, P# }
      FOREIGN KEY { S# } REFERENCES S_SINCE

CONSTRAINT CR9 IS_EMPTY
     ( ( S_SINCE JOIN SP_SINCE )
        WHERE SINCE < S#_SINCE )
```

64

## Meeting the Nine Requirements (b): historical relvars only

```
S_DURING { S#,  DURING }
    USING ( DURING ) ◄ KEY { S#, DURING } ►
    USING ( DURING ) ◄ FOREIGN KEY { S#, DURING }
                    REFERENCES S_STATUS_DURING ►

S_STATUS_DURING { S#, STATUS, DURING }
    USING ( DURING ) ◄ KEY { S#, DURING } ►
    USING ( DURING ) ◄ FOREIGN KEY { S#, DURING }
                        REFERENCES S_DURING ►

SP_DURING { S#, P#, DURING }
    USING ( DURING ) ◄ KEY { S#, P#, DURING } ►
    USING ( DURING ) ◄ FOREIGN KEY { S#, DURING }
                        REFERENCES S_DURING ►
```

65

## Meeting the Nine Requirements (c): current and historical relvars

Very difficult, even with shorthands defined so far. E.g.,

Requirement R9:
If the database shows supplier S*x* as being able to supply any part P*y* on day *d*, then it must also show supplier S*x* as being under contract on day *d*.

```
CONSTRAINT BR9_A IS_EMPTY
( ( S_SINCE JOIN SP_SINCE ) WHERE S#_SINCE > SINCE )

CONSTRAINT BR9_B
WITH ( EXTEND S_SINCE
   ADD ( INTERVAL_DATE ( [S#_SINCE : LAST_DATE ( ) ] )
     AS DURING ) { S#, DURING } AS T1,
(T1 UNION S_DURING ) AS T2,
SP_DURING { S#, DURING } AS T3 :
USING ( DURING ) ◄ T3 ⊆ T2 ►
```

(Note U_ form of relational comparison operator)

66

## Special Treatment for Current and Historical Relvars

So, to cut a long story short:

```
VAR S_SINCE RELATION
  { S#            S#,
    S#_SINCE      DATE  SINCE_FOR { S# }
                        HISTORY_IN ( S_DURING ),
    STATUS        INTEGER,
    STATUS_SINCE  DATE  SINCE_FOR { STATUS }
                        HISTORY_IN
                          ( S_STATUS_DURING ) }
  KEY { S# } ;
VAR SP_SINCE RELATION
  { S#            S#, P#          P#,
    SINCE         DATE  SINCE_FOR { S#, P# }
                        HISTORY_IN ( SP_DURING ) }
  KEY { S#, P# }
  FOREIGN KEY { S# } REFERENCES S_SINCE ;
```

and we conjecture that the historical relvar definitions can be generated automatically.

67

# CHAPTER 13: Database Queries

68

## Database Queries

In Chapter 13, twelve generic queries of varying complexity are presented and then solved:

a. for current relvars only
b. for historical relvars only
c. for both current and historical relvars

The c. section raises requirement for virtual relvars (views) that "undo" horizontal decomposition, such as:

```
VAR S_DURING_NOW_AND_THEN VIRTUAL
  S_DURING UNION
 ( EXTEND S_SINCE
  ADD INTERVAL_DATE ( [ S#_SINCE : LAST_DATE ( ) ] )
  AS DURING )  { S#, DURING }
```

69

## Query Example

Example for c. (both current and historical relvars):

Get supplier numbers for suppliers who were able to supply both part P1 and part P2 at the same time

```
WITH ( EXTEND SP_SINCE
          ADD INTERVAL_DATE ( [ SINCE : LAST_DATE ( ) ] )
                      AS DURING ) { S#, P#, DURING } AS T1 ,

     ( SP_DURING UNION T1 ) AS T2 ,

     ( T2 WHERE P# = P# ('P1') ) { S#, DURING } AS T3 ,

     ( T2 WHERE P# = P# ('P2') ) { S#, DURING } AS T4 ,

     ( USING ( DURING ) ◄ T3 JOIN T4 ► ) AS T5 :

T5 { S# }
```

70

# CHAPTER 14: Database Updates

71

## The Example Database

S_DURING

| S# | DURING |
|---|---|
| S1 | [*d04*:*d10*] |
| S2 | [*d02*:*d04*] |
| S2 | [*d07*:*d10*] |
| S3 | [*d03*:*d10*] |
| S4 | [*d04*:*d10*] |
| S5 | [*d02*:*d10*] |

Predicate: "Supplier S# was under contract throughout DURING (and not immediately before or after DURING)."

SP_DURING

| S# | P# | DURING |
|---|---|---|
| S1 | P1 | [*d04*:*d10*] |
| S1 | P2 | [*d05*:*d10*] |
| S1 | P3 | [*d09*:*d10*] |
| S1 | P4 | [*d05*:*d10*] |
| S1 | P5 | [*d04*:*d10*] |
| S1 | P6 | [*d06*:*d10*] |
| S2 | P1 | [*d08*:*d10*] |
| S2 | P1 | [*d02*:*d04*] |
| S2 | P2 | [*d08*:*d10*] |
| S2 | P2 | [*d03*:*d03*] |
| S3 | P2 | [*d09*:*d10*] |
| S4 | P2 | [*d06*:*d09*] |
| S4 | P4 | [*d04*:*d08*] |
| S4 | P5 | [*d05*:*d10*] |

Predicate: "Supplier S# was able to supply part P# throughout DURING (and not immediately before or after DURING)."

Regular INSERT, UPDATE, DELETE become too difficult for many common purposes …

## What Are The Problems?

Thirteen generic update operations of varying complexity are presented in terms of addition, removal or replacement of propositions. E.g.:

Add the proposition "Supplier S2 was under contract from day 5 to day 6".

Remove the proposition "Supplier S1 was able to supply part P1 from day 5 to day 6".

Replace the proposition "Supplier S2 was able to supply part P1 from day 3 to day 4" by the proposition "Supplier S2 was able to supply part P1 from day 5 to day 7".

Inevitable conclusion is need for U_update operators ...

73

## U_ update operators

"U_INSERT":

USING ( *ACL* ) ◄ INSERT *R r* ► ;
is shorthand for
*R* := USING (*ACL* ) ◄ *R* UNION *r* ;►

"U_DELETE":

USING ( *ACL* ) ◄ DELETE *R* WHERE *p* ;►
is shorthand for
*R* := USING (*ACL* ) ◄ *R* WHERE NOT *p* ;►

and there's "U_UPDATE" too, of course (difficult to define formally)

But U_update operators aren't all that's needed ...

74

## The PORTION Clause

S_DURING

| S# | DURING |
|---|---|
| S1 | [ *d03* : *d10* ] |
| S2 | [ *d02* : *d05* ] |

Replace the proposition "Supplier S1 was under contract from day 4 to day 8" by "Supplier S2 was under contract from day 6 to day 7".
(A trifle unreasonable but must be doable!)

We introduce PORTION:

UPDATE S_DURING WHERE S# = S# ( 'S1' )
PORTION { DURING = INTERVAL_DATE ( [ *d04* : *d08* ] ) }
( S# := S# ( 'S2' ) ,
DURING := INTERVAL_DATE ( [ *d06* : *d07* ] ) ) ;

yielding:

| S# | DURING |
|---|---|
| S1 | [ *d03* : *d03* ] |
| S1 | [ *d09* : *d10* ] |
| S2 | [ *d02* : *d07* ] |

75

## Updating the Combination View

Finally, we need to be able to apply update operators to the virtual relvar that combines current state with history.

So we propose to add a COMBINED_IN specification to relvar declaration syntax, for that express purpose. E.g.:

```
VAR S_SINCE RELATION
 { S#            S#,
   S#_SINCE      DATE   SINCE_FOR { S# }
                        HISTORY_IN ( S_DURING )
      COMBINED_IN ( S_DURING_NOW_AND_THEN ),
   STATUS         INTEGER,
   STATUS_SINCE  DATE   SINCE_FOR { STATUS }
                        HISTORY_IN
                          ( S_STATUS_DURING )
      COMBINED_IN
          ( S_STATUS_ DURING_NOW_AND_THEN )
 KEY { S# } ;
```

76

# CHAPTER 15: Stated Times and Logged Times

77

## Proposed Terminology

Stated times = "valid times"
Logged times = "transaction times"

Justification for proposed terms:
The stated times of proposition *p* are times when, according to our current belief, *p* was, is or will be true.
The logged times of proposition *q* are times (in the past and present only) when the database recorded *q* as being true.

[If *q* includes a stated time, then some might call "*q* during logged time [*t1*:*t2*]" a "bitemporal" proposition and hence talk about "bitemporal relations". We don't.]

78

## Special Treatment for Logged Times

We propose a LOGGED_TIMES_IN specification to be available in relvar declarations. E.g.:

```
VAR S_DURING RELATION
  { S#          S#,
    DURING      INTERVAL_DATE }
USING ( DURING ) ◄ KEY { S#, DURING } ►
LOGGED_TIMES_IN ( S_DURING_LOG ) ;
```

Attributes of S_DURING_LOG are S#, DURING and a third one, for logged times.

79

## Chapter 16: Point Types Revisited

Detailed investigation of point types and the significance of scale (preferred term to "granularity"). Includes discussion of:

If point type *pt2* is a proper subtype of *pt1* (under specialisation by constraint), what are the consequences for types INTERVAL_*pt2* and INTERVAL_*pt1*?
(E.g.: EVEN_INTEGER and INTEGER)

What about nonuniform scales, as with pH values, Richter values and prime numbers?

What about cyclic point types, such as WEEKDAY and times of day? Consequences of $a < b$ being equivalent to $a \neq b$ for all (a,b), leading to modified definitions of various interval operators.

Is there any point in considering *continuous* point types? We conclude not, because you lose some operators and gain none.

80

## Appendixes

A. Implementation Considerations
- Various useful transformations.
- Avoiding unpacking.
- The SPLIT operator.
- Algorithms for implementing U_ operators.

B. Generalizing EXPAND and COLLAPSE
- On sets of relations, sets of sets, sets of bags, other kinds of sets.
- PACK, UNPACK and U_ operators therefore also defined for relations with attributes having such types.

C. References and Bibliography
- Over 100 references

81

## Beware of Wikipedia!

"A temporal database is a database management system with built-in time aspects, e.g. a temporal data model and a temporal version of structured query language.

"More specifically the temporal aspects usually include valid-time and transaction-time. These attributes go together to form bitemporal data.

- "Valid time denotes the time period during which a fact is true with respect to the real world.
- "Transaction time is the time period during which a fact is stored in the database.
- "Bitemporal data combines both Valid and Transaction Time."

82

## Beware of Wikipedia!

"Valid time is the time for which a fact is true in the real world. In the example above, the Person table gets two extra fields, Valid-From and Valid-To, specifying when a person's address was valid in the real world. On April 4th, 1975 Joe's father proudly registered his son's birth. An official will then insert a new entry to the database stating that John lives in Smallville from the April, 3rd. Notice that although the data was inserted on the 4th, the databases states that the information is valid since the 3rd. The official does not yet know if or when John will ever move to a better place so in the database the Valid-To is filled with infinity (∞). Resulting in this entry in the database:

"Person(John Doe, Smallville, 3-Apr-1975, ∞)"

← Uh?

83

The End

84