Temporal Data and The Relational Model

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


for Warwick University
CS253

The Book’s Aims

• Describe a foundation for inclusion of support for temporal data in a truly relational database management system (TRDBMS).
• Focusing 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.

Contents (Parts I and II)

Part I: Preliminaries
Chapter 1: A Review of Relational Concepts
Chapter 2: An Overview of Tutorial D

Part II: Laying the Foundations
Chapter 3: Time and the Database
Chapter 4: What is the Problem?
Chapter 5: Intervals
Chapter 6: Operators on Intervals
Chapter 7: The COLLAPSE and EXPAND Operators
Chapter 8: The PACK and UNPACK Operators
Chapter 9: Generalising the Relational Operators

Contents (Part III)

Part III: Building on the Foundations
Chapter 10: Database Design
Chapter 11: Integrity Constraints I: Candidate Keys and Related Constraints
Chapter 12: Integrity Constraints II: General Constraints
Chapter 13: Database Queries
Chapter 14: Database Updates
Chapter 15: Stated Times and Logged Times
Chapter 16: Point and Interval Types Revisited

Appendixes

Appendix A: Implementation Considerations
Appendix B: Generalizing the EXPAND and COLLAPSE Operators
Appendix C: References and Bibliography
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.

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.

Example: Current State Only

"Suppliers and Shipments"

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)
Required Constraints

**S_FROM_TO**

- **S1**: FROM: d04 TO: d10
- **S2**: d02 d04
- **S3**: d03 d10
- **S4**: d04 d10
- **S5**: d02 d10

**SP_FROM_TO**

- **S1**: FROM: d04 TO: d10
- **S2**: d02 d04
- **S3**: d03 d10
- **S4**: d04 d10
- **S5**: d02 d10

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

These are very difficult!

---

**“Fully temporalising” (try 2)**

**S_DURING**

- **S1**: [0:10]
- **S2**: [0:10]
- **S3**: [0:10]
- **S4**: [0:10]
- **S5**: [0:10]

**SP_DURING**

- **S1**: [0:10]
- **S2**: [0:10]
- **S3**: [0:10]
- **S4**: [0:10]
- **S5**: [0:10]

Introduction of interval types and their point types.

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.

---

**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])**

---

**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)
Comparisons of Two Intervals

For given intervals, \( i_1 \) and \( i_2 \):

- \( i_1 = i_2 \)
- \( i_1 \) MEETS \( i_2 \)
- \( i_1 \) OVERLAPS \( i_2 \)
- \( i_1 \) SUCCEEDS \( i_2 \)
- \( i_1 \) PRECEDES \( i_2 \)
- \( i_1 \subseteq i_2 \)
- \( i_1 \) BEGINS \( i_2 \)
- \( i_1 \) ENDS \( i_2 \)
- \( i_1 \subseteq i_2 \)
- \( i_1 \subseteq i_2 \)
- \( i_1 \) \( \sqsubseteq \) \( i_2 \)
- \( i_1 \) MERGES \( i_2 \)

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

Allen uses DURING for \( \subseteq \)

Allen uses STARTS and ENDS

Added by Date, Darwen, Lorentzos

\( \text{MERGES} = \text{MEETS OR OVERLAPS} \)

Some Pictorial Definitions

\( i_1 = i_2 \)
\( i_1 \) MEETS \( i_2 \)
\( i_1 \) OVERLAPS \( i_2 \)
\( i_1 \) SUCCEEDS \( i_2 \)
\( i_1 \) PRECEDES \( i_2 \)
\( i_1 \subseteq i_2 \)
\( i_1 \) BEGINS \( i_2 \)
\( i_1 \) ENDS \( i_2 \)

More Dyadic Operators

Membership test:

- \( p \in i_1 \) or \( p \text{ IN } i_1 \) (where \( p \) is a point)

Dyadic operators that return intervals:

- \( i_1 \) \text{ UNION } \( i_2 \)
- \( i_1 \) \text{ INTERSECT } \( i_2 \)
- \( i_1 \) \text{ MINUS } \( i_2 \)

* empty intervals, such as \( \text{INTERVAL\_INTEGER}([1:1]) \), are not supported at all!

CHAPTER 7: The COLLAPSE and EXPAND Operators

Sets of Intervals

Let \( S_1 \) and \( S_2 \) be sets of intervals—e.g., \([1:2], [4:7], [6:9] \)

We define an equivalence relationship:

\[ S_1 = S_2 \text{ iff every point in an interval in } S_1 \text{ is a point in some interval in } S_2, \text{ and vice versa.} \]

Under this equivalence relationship we then define two canonical forms: \text{collapsed form} and \text{expanded form}.

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

Collapsed Form

No two elements, \( i_1 \) and \( i_2 \) \((i_1 \neq i_2)\) are such that \( i_1 \) MERGES \( i_2 \).

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

**COLLAPSE and EXPAND**

Let \( S \) be a set of intervals.

Then:
- \( \text{COLLAPSE}(S) \) denotes the collapsed form of \( S \).
- \( \text{EXPAND}(S) \) denotes the expanded form of \( S \).

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

**CHAPTER 8: The PACK and UNPACK Operators**

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).

**Packed Form**

<table>
<thead>
<tr>
<th>SD_PART</th>
<th>DURING</th>
</tr>
</thead>
<tbody>
<tr>
<td>S#</td>
<td>[d02:d04]</td>
</tr>
<tr>
<td>S2</td>
<td>[d02:d04]</td>
</tr>
<tr>
<td>S4</td>
<td>[d02:d04]</td>
</tr>
<tr>
<td>S4</td>
<td>[d04:d10]</td>
</tr>
</tbody>
</table>

PACK SD_PART ON (DURING)

Packed form of SD_PART "on DURING":

<table>
<thead>
<tr>
<th>SD_PART</th>
<th>DURING</th>
</tr>
</thead>
<tbody>
<tr>
<td>S4</td>
<td>[d02:d05]</td>
</tr>
<tr>
<td>S4</td>
<td>[d02:d05]</td>
</tr>
<tr>
<td>S4</td>
<td>[d04:d10]</td>
</tr>
</tbody>
</table>

**Unpacked Form**

<table>
<thead>
<tr>
<th>SD_PART</th>
<th>DURING</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
<td>[d02:d04]</td>
</tr>
<tr>
<td>S2</td>
<td>[d02:d04]</td>
</tr>
<tr>
<td>S4</td>
<td>[d02:d04]</td>
</tr>
<tr>
<td>S4</td>
<td>[d04:d10]</td>
</tr>
</tbody>
</table>

UNPACK SD_PART ON (DURING)
Properties of PACK and UNPACK

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

Unpacking on several attributes:
- \( \text{UNPACK } r \text{ ON } (a_1, a_2) = \text{UNPACK } (\text{UNPACK } r \text{ ON } a_1) \text{ ON } a_2 = \text{UNPACK } (\text{UNPACK } r \text{ ON } a_2) \text{ ON } a_1 \)

Packing on several attributes:
- \( \text{PACK } r \text{ ON } (a_1, a_2) = \text{PACK } (\text{PACK } (\text{UNPACK } r \text{ ON } (a_1,a_2)) \text{ ON } a_1 ) \text{ ON } a_2 = \text{PACK } (\text{PACK } r \text{ ON } a_1) \text{ ON } a_2 \)
- \( \text{not: PACK } r \text{ ON } a \text{ not: PACK } r \text{ ON } a_1 \text{ not: PACK } r \text{ ON } a_2 \)
- Although redundancy is eliminated, result can be of greater cardinality than \( r \).

CHAPTER 9: Generalizing the Relational Operators

Tutorial D's Relational Operators

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

New syntax for invoking each operator:
- USING (ACL) \( \text{rel op inv} \)
  - 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

Example 1: U_PROJECT

USING (DURING) \( \text{SP}_{\text{DURING}} \) { 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".

Other examples are U_JOIN, U_UNION, U_restrict, etc.

Example 2: U_NOT MATCHING

USING (DURING) \( \text{SP}_{\text{DURING}} \) \( \text{S}_{\text{DURING}} \) \( \text{NOT MATCHING} \) \( \text{SP}_{\text{DURING}} \) gives (S#, DURING) pairs such that supplier S# was under contract but unable to supply any part throughout the interval DURING.

Example 3: U_SUMMARIZE

USING (DURING) \( \text{SUMMARIZE SP}_{\text{DURING}} \) PER (S, DURING { S#, DURING }) \( \text{ADD COUNT AS NO_OF_PARTS} \)
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:
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.

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.

Contents

Chapter 10: Database Design
• Introduction
• Current relvars only
• Historical relvars only
• Sixth normal form (6NF)
• "The moving point now"
• Both current and historical relvars
• Concluding remarks
• Exercises

At last, we focus on specifically temporal issues!

Current Relvars Only

<table>
<thead>
<tr>
<th>SSSC</th>
<th>S#</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>S1</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S2</td>
<td>S2</td>
<td>Jones</td>
<td>10</td>
<td>Paris</td>
</tr>
<tr>
<td>S3</td>
<td>S3</td>
<td>Blake</td>
<td>30</td>
<td>Paris</td>
</tr>
<tr>
<td>S4</td>
<td>S4</td>
<td>Clark</td>
<td>20</td>
<td>London</td>
</tr>
<tr>
<td>S5</td>
<td>S5</td>
<td>Adams</td>
<td>30</td>
<td>Athens</td>
</tr>
</tbody>
</table>

Note: keys indicated by underlining attribute names

Semitemporalizing SSSC (try 1)

<table>
<thead>
<tr>
<th>SSSC</th>
<th>S#</th>
<th>SNAME</th>
<th>STATUS</th>
<th>CITY</th>
<th>SINCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>S1</td>
<td>Smith</td>
<td>20</td>
<td>London</td>
<td>d04</td>
</tr>
<tr>
<td>S2</td>
<td>S2</td>
<td>Jones</td>
<td>10</td>
<td>Paris</td>
<td>d05</td>
</tr>
<tr>
<td>S3</td>
<td>S3</td>
<td>Blake</td>
<td>30</td>
<td>Paris</td>
<td>d02</td>
</tr>
<tr>
<td>S4</td>
<td>S4</td>
<td>Clark</td>
<td>20</td>
<td>London</td>
<td>d09</td>
</tr>
<tr>
<td>S5</td>
<td>S5</td>
<td>Adams</td>
<td>30</td>
<td>Athens</td>
<td>d09</td>
</tr>
</tbody>
</table>

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.
Semitemporalizing SSSC (try 2)

VAR S_SINCE
BASE RELATION
( 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!

Fully Temporalizing SSSC

VAR S_DURING
BASE RELATION
( 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#, 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.

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).

\( S\_DURING, S\_NAME\_DURING \) 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).

“Circumlocution” and 6NF

Note S1 named Smith throughout \([d01:d09]\), split across tuples.
We call this undesirable phenomenon circumlocution.
Decompose to 6NF, using \( U\_projection \):

"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

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 ...

CS253: Topics in Databases
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:

<table>
<thead>
<tr>
<th>S#</th>
<th>STATUS</th>
<th>DURING</th>
</tr>
</thead>
<tbody>
<tr>
<td>S4</td>
<td>25</td>
<td>[d05 : d06]</td>
</tr>
<tr>
<td>S4</td>
<td>25</td>
<td>[d06 : d07]</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>S#</th>
<th>STATUS</th>
<th>DURING</th>
</tr>
</thead>
<tbody>
<tr>
<td>S4</td>
<td>25</td>
<td>[d05 : d06]</td>
</tr>
<tr>
<td>S4</td>
<td>25</td>
<td>[d06 : d07]</td>
</tr>
</tbody>
</table>

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#_SINCE, STATUS, STATUS_SINCE }
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:

<table>
<thead>
<tr>
<th>S#</th>
<th>STATUS</th>
<th>DURING</th>
</tr>
</thead>
<tbody>
<tr>
<td>S4</td>
<td>25</td>
<td>[d04 : d06]</td>
</tr>
<tr>
<td>S4</td>
<td>10</td>
<td>[d05 : d07]</td>
</tr>
</tbody>
</table>

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 ).

When / Then without PACKED ON

Example (presidential terms):

<table>
<thead>
<tr>
<th>TERM</th>
<th>DURING</th>
<th>PRESIDENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1974 : 1976</td>
<td>Ford</td>
<td></td>
</tr>
<tr>
<td>1977 : 1980</td>
<td>Carter</td>
<td></td>
</tr>
<tr>
<td>1981 : 1984</td>
<td>Reagan</td>
<td></td>
</tr>
<tr>
<td>1985 : 1988</td>
<td>Reagan</td>
<td></td>
</tr>
<tr>
<td>1993 : 1996</td>
<td>Clinton</td>
<td></td>
</tr>
<tr>
<td>1997 : 2000</td>
<td>Clinton</td>
<td></td>
</tr>
<tr>
<td>2009 : 2012</td>
<td>Obama</td>
<td></td>
</tr>
</tbody>
</table>

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.

Neither When / Then nor PACKED ON

Example (measures of inflation):

<table>
<thead>
<tr>
<th>INFLATION</th>
<th>DURING</th>
<th>PERCENTAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>m01 : m03</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>m04 : m06</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>m07 : m09</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>m07 : m07</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>m01 : m12</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

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"

When / Then and PACKED ON both required

VAR S_STATUS_DURING RELATION
{ S# S#, STATUS CHAR, DURING INTERVAL_DATE }
USING ( DURING )

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.

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.
Requirement Group 1

Requirement R1:
If the database shows supplier Sx 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 Sx 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 Sx as being under contract on day d, then it must also show supplier Sx as having some status on day d.
Note: to do with denseness

Requirement Group 2

Requirement R4:
If the database shows supplier Sx 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 Sx 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 Sx as having some status on day d, then it must also show supplier Sx as being under contract on day d.
Note: to do with denseness

Requirement Group 3

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

Requirement R8:
If the database shows supplier Sx as being able to supply part Py 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 Sx as being able to supply some part on day d, then it must also show supplier Sx as being under contract on day d.
Note: to do with denseness

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 )

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 ★

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

Meeting the Nine Requirements (a):
current relvars only

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

Requirement R9:
If the database shows supplier Sx as being able to supply any part Py on day d, then it must also show supplier Sx as being under contract on day d.
CONSTRAINT BR9_A IS_EMPTY
( ( S_SINCE JOIN SP_SINCE ) WHERE STATUS_SINCE >= DURING )
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.C T2 ★
(Note U_ form of relational comparison operador)
So, to cut a long story short:

**VAR S_SINCE RELATION**

<table>
<thead>
<tr>
<th>S#</th>
<th>S#_SINCE</th>
<th>DATE</th>
<th>SINCE_FOR</th>
<th>{ S# }</th>
<th>HISTORY_IN</th>
<th>{ S_DURING }</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS</td>
<td>INTEGER</td>
<td>STATUS</td>
<td>SINCE_FOR</td>
<td>STATUS</td>
<td>HISTORY_IN</td>
<td>{ STATUS_SINCE }</td>
</tr>
<tr>
<td>KEY</td>
<td>{ S# }</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**VAR SP_SINCE RELATION**

<table>
<thead>
<tr>
<th>S#</th>
<th>S#_, P#</th>
<th>P#</th>
<th>SINCE</th>
<th>DATE</th>
<th>SINCE_FOR</th>
<th>{ S#, P# }</th>
<th>HISTORY_IN</th>
<th>{ SP_DURING }</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY</td>
<td>{ S#, P# }</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOREIGN KEY</td>
<td>{ S# } REFERENCES S_SINCE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

### Special Treatment for Current and Historical Relvars

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**

(EXTEND S_SINCE
ADD INTERVAL_DATE ([ S#_SINCE : LAST_DATE ()] AS DURING ) { S#, DURING }

**VAR SP_SINCE RELATION**

<table>
<thead>
<tr>
<th>S#</th>
<th>S#, P#</th>
<th>P#</th>
<th>SINCE</th>
<th>DATE</th>
<th>SINCE_FOR</th>
<th>{ S#, P# }</th>
<th>HISTORY_IN</th>
<th>{ SP_DURING }</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY</td>
<td>{ S#, P# }</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOREIGN KEY</td>
<td>{ S# } REFERENCES S_SINCE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Database Queries

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

```sql
WITH ( EXTEND SP_SINCE
ADD INTERVAL_DATE ([ SINCE : LAST_DATE ()] AS DURING ) ( SP_DURING UNION T1 ) AS T2 ,
( T2 WHERE P# = P# ('P1') ) ( SP_DURING ) AS T3 ,
( T2 WHERE P# = P# ('P2') ) ( SP_DURING ) AS T4 ,
( USING ( DURING ) T3 JOIN T4 ) AS T5 :
T5 { S# }
```

### 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

```sql
WITH ( EXTEND SP_SINCE
ADD INTERVAL_DATE ([ SINCE : LAST_DATE ()] AS DURING ) ( SP_DURING UNION T1 ) AS T2 ,
( T2 WHERE P# = P# ('P1') ) ( SP_DURING ) AS T3 ,
( T2 WHERE P# = P# ('P2') ) ( SP_DURING ) AS T4 ,
( USING ( DURING ) T3 JOIN T4 ) AS T5 :
T5 { S# }
```

### The Example Database

<table>
<thead>
<tr>
<th>S_DURING</th>
<th>S#</th>
<th>DURING</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>004</td>
<td>d10</td>
</tr>
<tr>
<td>2</td>
<td>002</td>
<td>d04</td>
</tr>
<tr>
<td>2</td>
<td>007</td>
<td>d10</td>
</tr>
<tr>
<td>3</td>
<td>003</td>
<td>d10</td>
</tr>
<tr>
<td>4</td>
<td>004</td>
<td>d10</td>
</tr>
<tr>
<td>5</td>
<td>005</td>
<td>d10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SP_DURING</th>
<th>S#</th>
<th>P#</th>
<th>DURING</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>002</td>
<td>P1</td>
<td>d10</td>
</tr>
<tr>
<td>2</td>
<td>005</td>
<td>P2</td>
<td>d10</td>
</tr>
<tr>
<td>3</td>
<td>009</td>
<td>P1</td>
<td>d10</td>
</tr>
<tr>
<td>4</td>
<td>005</td>
<td>P3</td>
<td>d10</td>
</tr>
<tr>
<td>5</td>
<td>004</td>
<td>P2</td>
<td>d10</td>
</tr>
<tr>
<td>6</td>
<td>006</td>
<td>P1</td>
<td>d10</td>
</tr>
</tbody>
</table>

Predicate:

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

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 ...

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 ...

The PORTION Clause

<table>
<thead>
<tr>
<th>S#</th>
<th>DURING</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>[d03 : d10]</td>
</tr>
<tr>
<td>S2</td>
<td>[d02 : d05]</td>
</tr>
</tbody>
</table>

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]) )
(DURING := INTERVAL_DATE ([d06 : d07]));
```

yielding:

<table>
<thead>
<tr>
<th>S#</th>
<th>DURING</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>[d03 : d03]</td>
</tr>
<tr>
<td>S1</td>
<td>[d09 : d10]</td>
</tr>
<tr>
<td>S2</td>
<td>[d02 : d07]</td>
</tr>
</tbody>
</table>

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# };
```

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.]
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#, DURING INTERVAL_DATE }
USING ( DURING )
LOGGED_TIMES_IN ( S_DURING_LOG ) ;

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

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 ≠ 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.

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

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."

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 database states that the information is valid since the 3rd.

Uh?