Temporal Data and The Relational Model

Hugh Darwen
HD@TheThirdManifesto.com
www.TheThirdManifesto.com

University of Warwick
Course CS319

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.**

The chapter in Date first appeared in his 7th edition, as Chapter 22, but the chapter was quite heavily revised for the 8th edition.

There is an unfortunate typographical error on page 744. In the first bulleted paragraph ("The expanded form ..."), delete the last three words, "defined as follows:".

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!

March 2004 (c) Hugh Darwen

In particular, none of the leading SQL vendors (IBM, Oracle, Microsoft, Sybase ...) have implemented SQL extensions to solve the problems we describe.

There was significant interest for a time in the second half of the 1990s, when an incomplete working draft for an international standard for such extensions was produced by the SQL standards committee. However, the project was abandoned when support for XML documents in SQL databases became a higher priority to the industry than temporal extensions.

(Some people question the industry's priorities!)

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 of time.
- Propose additional operators on relations and relation variables ("relvars") having interval-valued attributes.
- Propose additional constraint definitions and new design constructs for management of temporal data.
- All of the above to be definable in terms of existing operators and constructs.
- And explore some interesting side issues.

March 2004 (c) Hugh Darwen

Imagine, for example, a database from which nothing is ever deleted and in which every record is somehow timestamped to show the time at which it arrived and, if its information is no longer current, the time at which it was superseded or deleted. The interval between those two times is an interval *throughout* which the record was "valid" (i.e., represented a held belief).

Such a database might be called a temporal database, but there is no precise definition of that term, nor do we really need one.

The records in a temporal database don't have to be exclusively about the past and present. They could be about the future, too (e.g., employees' planned vacations, project schedules etc.), though beliefs about the future are usually subject to more uncertainty than those about the past and present.

The Book's Structure (Parts I and II)

Part I Preliminaries

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

Part 2 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

March 2004 (c) Hugh Darwen

The course broadly follows the structure of the book, but we assume you have a grasp of relational concepts and we will not spend time teaching Tutorial D in detail.

The official definition of Tutorial D is in *Foundation for Future Database Systems: The Third Manifesto*, by C.J. Date and Hugh Darwen (Addision-Wesley, 2000, ISBN: 0-201-70928-7)

You are not expected to learn Tutorial D syntax in detail, but you should try to understand and be able to reproduce, roughly at least, the examples used in these slides.

The Book's Structure (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

March 2004

Regarding database design, constraints, queries and updates, you will be shown the complexity of the problems that need to be solved, and proposed solutions to those problems. You should familiarise yourself with the solutions and be able to describe in broad outline some of the problems addressed by those solutions. But you do not need to learn the complicated Tutorial D expressions for the longhand expansions of the proposed new shorthands!

The topics of Chapters 15 and 16 are **not** included in this course.

The Book's Structure (Appendixes)as

Appendixes

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

March 2004

None of these topics is included in the course.

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 "Foundation for Future Database Systems: The Third Manifesto" (2nd edition, Addison-Wesley, 2000). 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.

March 2004 (c) Hugh Darwen

Note the careful distinction between ***values*** (relation values in particular) and ***variables*** (relation variables in particular).

We normally abbreviate "relation variable" to ***relvar***. The SQL counterpart, roughly speaking, is the base table, though strictly speaking this corresponds to what we call ***real relvars*** in particular. (Our counterpart of the SQL updatable view is the ***virtual relvar***.)

In a separate handout (a single sheet) you will find an annotated table showing various notations for invoking operators of the relational algebra: Tutorial D, ISBL, Predicate Logic (where the operands are predicates rather than relations) and SQL.

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.

Quantisation is the key. Although most people intuitively think of time as continuous, we consider a time interval to be a set of discrete, equally spaced points. The "distance" between adjacent points is according to a chosen **scale**. In all our examples the scale is one day unless otherwise stated (explicitly or implicitly).

Quantisation has the huge advantage of making an interval correspond to a ***finite*** set of points. Computers are much better at dealing with finite sets than infinite ones and the Relational Model is explicitly based on finite relations only.

"Valid time" and **"transaction time"** are rather inappropriate and unintuitive terms in widespread use in the temporal database community. The valid time of a record refers to all the times at which the proposition it represents is held to be true. The transaction time of a record refers to all the times at which it was or is "in the database". **We do not pursue these concepts on this course.**

Chapter 4: What Is the problem?

S

S#
S1
S2
S3
S4
S5

SP

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

Suppliers and shipments:

S: "Supplier S# is under contract"

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

Nontemporal (current state only).

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

March 2004 (c) Hugh Darwen

This is what you might call a *non*temporal database. We use it as a starting point from which we will develop, in three stages, its fully temporal counterpart.

The queries we can make on this database have temporal counterparts too, and so do the constraints we would like to declare, and so do the update operations we would like to be able to perform---as we shall see as the course unfolds.

Chapter 4: What Is the problem?

"Semitemporalising"

S_SINCE

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

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

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

SP_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*

Queries: *Since when has supplier S# been able to supply something?* (Not too difficult)

Since when has supplier S# been unable to supply anything? (Impossible)

March 2004

"Semitemporalising" because we are doing only half the job, so to speak. Actually, rather less than half.

Although such "since" relvars are inadequate of themselves, we shall see (much later) that they do have part to play in a fully temporal database.

The notation *dnn* for a day number is used for convenience. In real life we would normally expect to see a date, such as 2004-03-01.

For each proposition (represented by a tuple), we have a "since" value indicating the day on which the proposition in question first became true. It is assumed still to be true at the present time. We have no record of similar propositions that used to be true in the past but are no longer true. Thus, this is still a "current state" database.

And of course the existing technology can easily handle such databases. Well, comparatively easily, anyway. But observe that SQL, for example, has no shorthand for expressing the constraint to the effect that the SINCE value in an SP tuple had better not be earlier than the SINCE value in the corresponding S tuple (for a supplier cannot be able to supply anything while not under contract).

Exercise: Write a **Tutorial D** or SQL expression for the constraint just described. For **Tutorial D**, you can use IS_EMPTY (*rel expr*) to express a constraint to the effect that the result of evaluating *rel expr* (an expression in **Tutorial D**'s relational algebra) must at all times be empty.

Chapter 4: What Is the problem?fs

"Fully temporalising" (try 1)

S_FROM_TO

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

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

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	*d03*	*d03*
S2	P2	*d09*	*d10*
S3	P2	*d08*	*d10*
S4	P2	*d06*	*d09*
S4	P4	*d04*	*d08*
S4	P5	*d05*	*d10*

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

Queries: ***During which times was supplier S# able to supply something?*** (Very difficult)

During which times was supplier S# unable to supply anything? (Very difficult)

March 2004 (c) Hugh Darwen

Now we have the times at which true propositions ceased to be true as well as the times at which they started to be true. And that means we have a historical record as well as a record of the current state of affairs (assuming that today is day 10, so every tuple whose TO value is *d10* represents a current state--an interim and inadequate solution to a difficult problem we will return to later).

By "very difficult", we mean so difficult that we won't even show how it might be done! But those queries are not impossible and you are welcome to have a try (in **Tutorial D** or SQL). In each case, the result should not show two or more tuples for the same supplier whose FROM-TO intervals overlap in time or are such that one immediately follows the other in time.

Notice "try 1". Although this representation can be achieved with existing technology, it is not really very suitable. When working with intervals, we sometimes want the end points to be considered as included, sometimes not. For example, how does the system know whether S2 was under contract on day 4, or whether day 4 was actually the first day on which S2 ceased to be under contract? Soon we will introduce "try 2" as a better solution, overcoming this problem.

Chapter 4: What Is the problem?

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	*d03*	*d03*
S2	P2	*d09*	*d10*
S3	P2	*d08*	*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!

March 2004 (c) Hugh Darwen

Again, you are welcome to try to express these constraints in Tutorial D or SQL.

Chapter 5: Intervals

"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*]

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	[*d03* :*d03*]
S2	P2	[*d09* :*d10*]
S3	P2	[*d08* :*d10*]
S4	P2	[*d06* :*d09*]
S4	P4	[*d04* :*d08*]
S4	P5	[*d05* :*d10*]

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

Here, the type of the DURING attributes is perhaps 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.

March 2004 (c) Hugh Darwen

Now we put both end points together in a single column, so to speak. A square bracket before the begin point or after the end point indicates that that point is included in the interval. But we don't actually store the brackets (or the colons)! The next slide explains.

Of all the values whose type is DATE, there is one for which NEXT_DATE (*d*) is undefined. And that is the value representing the date of the "end of time".

Similarly, there is one value for which PRIOR_DATE (*d*) is undefined: the date of the "beginning of time".

For this course, as in most of the book, we concentrate on point types like this, in which there is a first value and a last value. Note, however, that such types cannot be used for intervals over, for example, days of the week or times of day. These require so-called ***cyclic*** point types, which have some rather interesting properties and are described in Chapter 16. **You are not required to study cyclic point types.**

Chapter 6: Operators on Intervals

Interval "selectors". E.g.:
INTERVAL_INTEGER ([1:10]) =
INTERVAL_INTEGER ((0:10]) =
INTERVAL_INTEGER ([1:11)) =
INTERVAL_INTEGER ((0:11))

BEGIN (*i*), END (*i*), PRE (*i*), POST (*i*) give the various bound points of *i*.

Membership test (of point in interval)

Interval comparisons: **Allen's operators**, to which we add *i1* MERGES *i2* (= *i1* MEETS *i2* OR *i1* OVERLAPS *i2).* =, of course, but no <.

Dyadic operators returning intervals: UNION, INTERSECT, MINUS.

COUNT (*i*) gives number of points in *i*.

March 2004 (c) Hugh Darwen

A **selector *S*** for type ***T*** is an operator that, when invoked, returns a value of type ***T***. None of the arguments to the invocation can be of type ***T***. For every value ***V*** of type ***T*** there is some invocation of ***S*** that returns ***V***.

A **literal** is a special kind of selector invocation. You can think of the literal 12 as being an invocation of a operator that operates on a given sequence of decimal digits and yields the integer indicated by that sequence. The invocations of INTERVAL_INTEGER shown in this slide are literals because in each case both arguments to the invocation are themselves literals.

Notice how there are four different ways of selecting the interval that runs from 1 to 10 inclusive. **Exercise:** How many ways are there of selecting the interval that runs from the beginning of time to the end of time?

Allen's comparison operators are described on the next slide.

The UNION, INTERSECT and MINUS operators are not defined for all pairs of intervals. They are defined only for pairs of intervals such that the union, intersection or difference of their sets of contained points constitutes a single interval.

So the operands *i1* and *i2* of UNION must be such that *i1* MERGES *i2* is true. **Exercise:** What constraint must the operands of INTERSECT satisfy? And those of MINUS?

Allen's Interval Comparison Operators

i1 MEETS i2

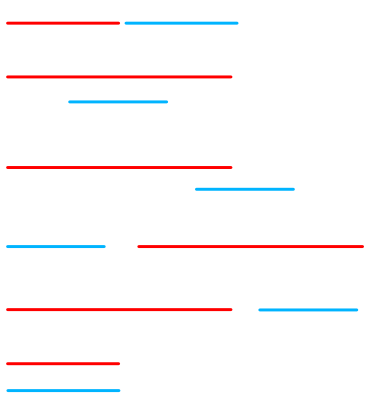
i1 CONTAINS i2

i1 OVERLAPS i2

i1 SUCCEEDS i2

i1 PRECEDES i2

i1 = i2



to which we add
i1 MERGES i2
short for
i1 MEETS i2 OR i1 OVERLAPS i2

March 2004

Chapter 7: The COLLAPSE and EXPAND Operators

COLLAPSE (SI) and **EXPAND (SI)**, where *SI* is a set of intervals, each yielding a set of intervals.

Equivalence relationship: *SI1* and *SI2* are equivalent iff every point in an interval in *SI1* is a point in some interval in *SI2*, and *vice versa*.

Canonical forms: **collapsed form** and **expanded form.** In each case, no point appears in more than one member interval.

SI is in **collapsed form** iff, for all (*p1, p2*) in *SI*, if *p1* and *p2* are contiguous points, then *p1* and *p2* are in the same interval (member of *SI).*

SI is in **expanded form** iff every interval in SI is a **unit interval** (contains just 1 point).

COLLAPSE (SI) and **EXPAND (SI)** are not required to exist in the language.

March 2004 (c) Hugh Darwen

If several different forms are deemed to represent the same thing under some **equivalence relationship**, then certain of those forms might be the preferred ones in certain circumstances. A form that is the preferred one for some purpose is a **canonical form**.

A set of intervals in **collapsed form** is the smallest of all the sets of intervals that are equivalent to it under the given equivalence relationship (ultimately constituting the same set of points). A set of intervals in **expanded form** is the biggest.

The COLLAPSE and EXPAND operators are not very useful in themselves, but they help with the definition of the more important operators to come.

Chapter 8: The PACK and UNPACK Operators

PACK and **UNPACK** operate on relations, yielding relations. Based on COLLAPSE and EXPAND.

Consider:

SD_PART

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

Packed form of SD_PART "on DURING":

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

Obtained by PACK SD_PART ON (DURING)

So **packed form** is a canonical form for relations that have interval-valued attributes. Its main purpose is to **avoid redundancy**.

A relation that is not in packed form suffers from problems similar to those of SQL tables that contain duplicate rows.

Chapter 8: The PACK and UNPACK Operators

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*]

Obtained by UNPACK SD_PART ON (DURING)

March 2004 (c) Hugh Darwen

In **unpacked form** every interval is a **unit interva**l, therefore containing just a single point.

The intervals of an unpacked form could easily be replaced by their single point values, but for definitional purposes it is more convenient to keep the intervals. We expect unpacked forms to exist mostly only conceptually; we do not expect actually to "see" them very often.

If all relations were in unpacked form, queries, constraints, and updates would be as easy to express as they are without the shorthands that we propose. But relations in unpacked form are difficult to interpret and might be huge--especially when the scale of the point type is very small (say, 1 microsecond).

As we shall see, the proposed shorthands allow us to think we are operating on unpacked forms even though we actually see packed forms.

Chapter 8: The PACK and UNPACK Operators

Packing/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* ≠
  PACK (PACK(UNPACK *R* ON (*A1,A2*)) ON *A2*)
  ON *A1*

  not: PACK (PACK *R* ON *A1*) ON *A2*

- Although redundancy is eliminated, result can be of greater cardinality than *R*.

March 2004 (c) Hugh Darwen

You can ignore this slide if you wish. On this course you are not expected to study relations with more than one interval-valued attribute.

Note that packing on two attributes is not simply packing on one and then packing the result on the other. In general, you have to unpack on both first. Even then, the result of packing on both depends on the order in which you do the packing. That is why **Tutorial D** uses the lunula form of parenthesis for enclosing the attribute name list, as opposed to the curly brace form, {...}, that we normally prefer for lists in which the order of elements is immaterial. In the case of UNPACK, the order ***is*** immaterial, but we thought it would be confusing if UNPACK and PACK used different notations.

Chapter 9: Generalizing the Relational Operators

Union, intersect and difference
Restrict, project and join (natural)
Extend and summarize

Syntax:

USING (*ACL*) ◂ *rel op inv* ▸

- *ACL* is attribute-name commalist
- we call these "U_" operators

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:

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"

March 2004 (c) Hugh Darwen

One of your handouts gives **Tutorial D**'s relational algebra operators alongside their counterparts in ISBL, predicate logic and SQL. But it is not important for you to remember all these operators in detail. It is much more important for you to understand the single principle underlying each of the new "U_" operators (U for USING).

Each "old" operator has a U_ counterpart. What's more, each U_ operator degenerates to its "old" counterpart when it is invoked with no USING attributes.

Later we shall see how this same USING construct applies to constraints and updating operations too.

Notice that the U_project example shown on this slide solves the first of our two "very difficult" queries ...

Chapter 9: Generalizing the Relational Operators

More examples

U_MINUS:

USING (DURING)
◂ S_DURING { S#, DURING } MINUS
SP_DURING { S#, DURING } ▸

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

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.

... and notice that the U_MINUS example shown here solves the second of those two "very difficult" queries.

U_SUMMARIZE turns out to be especially interesting, but its complications are really beyond the scope of this course and you are not expected to study it in detail. The next two slides are included for possible interest only.

Chapter 9: Generalizing the Relational Operators

U_SUMMARIZE is interesting (1)

U_SUMMARIZE:

USING (DURING)
◂ SUMMARIZE SP_DURING
PER S_DURING { DURING }
ADD COUNT AS NO_OF_PARTS ▸

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

Chapter 9: Generalizing the Relational Operators

U_SUMMARIZE is interesting (2)

U_SUMMARIZE:

USING (DURING)
◂ SUMMARIZE SP_DURING
PER S_DURING { S# }
ADD COUNT AS NO_OF_PARTS ▸

- note lack of DURING in PER relation, so unpacking effectively applies to first operand only
- for each S#, counts number of distinct cases of S# being able to supply some part on some date (!)

March 2004

This is the only exception to the general rule that both operand relations are conceptually unpacked on the USING attribute(s). Here, the USING attribute doesn't even exist in the PER relation, but that's okay because the operation is still well defined (and possibly useful).

U_JOIN could be subect to similar treatment, not insisting on both operands having all of the specified USING attributes.

Chapter 10: Database Design

Structure of chapter:

- 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!

Chapter 10: Database Designa

Current relvars only:

SSSC

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Note: attribute names of key members are underlined.

SP

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

Note that SSSC is in 5NF (fifth normal form) and yet is decomposible. For example, we could split it into three binary relvars with attributes {S#, SNAME}, (S#, STATUS} and {S#, CITY}. As it happens, each of those three would be in 6NF, whereas SSSC, by virtue of being decomposible, is not in 6NF.

Exercise: Why do we normally not decompose relvars such as SSSC, with our existing technology? What constraints would need to be declared if we did decompose it as suggested?

We shall see that 6NF, while possibly a bad idea here, becomes a positively good idea when we "temporalize" the database.

Chapter 10: Database Designfs

Semitemporalizing SSSC (try 1):

SSSC_SINCE

S#	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. We cannot tell since when a certain STATUS has held or a certain CITY has held or a certain NAME has held, or even since when a certain supplier has been under contract.

SSSC_SINCE is also in 5NF but not in 6NF, and 5NF is still sufficient.

To overcome the problem mentioned on the slide, we really need a separate "since" attribute for each attribute of SSSC, so to speak, as shown on the next slide.

Chapter 10: Database Design

Semitemporalizing SSSC (try 2):

```
VAR  S_SINCE 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 in city CITY since CITY_SINCE.

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

Again we are in 5NF but not in 6NF, and again 5NF is sufficient.

Notice how even the key (S#) has a corresponding "since" attribute, S#_SINCE. It indicates the date on which supplier S# was placed under contract. Since that date it is possible that S#'s name, status and city have all changed, so we do need this date to be separately recorded, if we need it at all. (If the key is composite, we do not need a separate "since" attribute for each component of the key.)

Exercise: What constraints should be declared for S_SINCE?

Chapter 10: Database Design

Fully temporalizing SSSC:

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

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

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

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

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

For each "since" attribute of SSSC_SINCE, we have a corresponding "during" relvar. The others, not shown on the slide, would be S_STATUS_DURING and S_CITY_DURING.

We call this vertical decomposition because it is "column-wise", according to the normal tabular representation of relations.

Chapter 10: Database Design

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 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).

According to the normal definition of 5NF, a **join dependency** is said to hold in relvar *R* if there exist *n* distinct projections of *R*, p_1, p_2,... p_n, such that $n > 1$ and at all times *R* is equal to p_1 JOIN p_2 JOIN ... p_n. (Cases where $n = 2$ are by far the most common.)

A join dependency is "implied by a candidate key of *R*" if each of the projections includes each attribute of that candidate key.

For the purposes of 6NF, the definition is altered slightly such that the projections become U_projections and the JOINs become U_JOINs.

Chapter 10: Database Design

"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!)

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

March 2004 (c) Hugh Darwen

A special marker, NOW, has been advocated by some authorities. The problems with this approach are described in detail on pages 177-180 of the book.

For example, consider the interval [NOW:*d14*]. What happens if that interval is recorded somewhere in the database and the clock reaches day 15? For another example, what is the effect, on day 14, of assigning the interval [*d01,* NOW] to variable I1? Does I1 have the value [*d01:d14*] or the "value" [*d01*:NOW]? Does I1 compare equal to [d01, NOW] on the day the assignment is performed? And on the next day?

The term **horizontal decomposition** appeals to the fact that tuples appear as rows in our normal tabular representation of relations and tuples representing the current state of affairs are kept in a separate relvar from those representing the past. The term is a little loose, because the current state tuples are not of exactly the same type as the historical ones.

Chapter 10: Database Design

Horizontal Decomposition

Keep S_SINCE for current state.

Use S_DURING, S_NAME_DURING, S_STATUS_DURING and S_CITY_DURING for history only.

Having accepted the inevitability of vertical and horizontal decomposition, we need to consider the consequences for constraints ...

In other words, if we take as our starting point a proposed relvar with attributes S#, SNAME, STATUS, CITY, DURING, we first horizontally decompose to give S_SINCE and a "during" relvar for the historical information. Then we vertically decompose the "during" relvar, using U_project.

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.

March 2004 (c) Hugh Darwen

Having explained vertical decomposition, we can now dispense with S_NAME_DURING and S_CITY_DURING because their treatment will be the same as that of S_STATUS_DURING.

Redundancy: saying the same thing more than once.

Circumlocution: saying something in a roundabout way.

Contradiction: saying something that cannot be true, such as "S1's status is 20 and S1's status is 30".

Chapter 11: Integrity Constraints I

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.

Chapter 11: Integrity Constraints I

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.

Chapter 11: Integrity Constraints I

Solving The Redundancy and Circumlocution Problems

```
VAR  S_STATUS_DURING RELATION
{ S# S#,
   STATUS CHAR, 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.

Chapter 11: Integrity Constraints I

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.

Avoided in the **unpacked form** of S_STATUS_DURING!

Chapter 11: Integrity Constraints I

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).

We call this shorthand a WHEN-THEN constraint.

Chapter 11: Integrity Constraints I

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

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.

March 2004 (c) Hugh Darwen

The question arises as to whether a WHEN-THEN constraint is ever needed without a PACKED ON constraint. This example is our best attempt to find such a case. It is not a very compelling example. (Can you think of a better one?)

Omitting the PACKED ON constraint would permit the tuple ([1994:1995], Clinton) to appear in addition to those shown in the example, though the WHEN-THEN constraint prohibits ([1994:1995], Lincoln) from being inserted.

Notice in passing that there appear to be a couple of gaps in the historical record here, for the intervals 1989-1992 and 2001 to the present day. Did the person responsible for the example accidentally miss something out or is there mischief afoot? In case you think another kind of constraint is needed to prevent such "accidents", you are absolutely right! We call this kind of constraint a "**denseness**" constraint. We shall come across the need for these in our suppliers-and-shipments database very shortly.

It seems that mostly both PACKED ON and WHEN-THEN are required, so a further shorthand is justified.

Chapter 11: Integrity Constraints I

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"

Can you think of a compelling example of a "during" relvar where neither a PACKED ON constraint nor a WHEN-THEN constraint is needed? We can't, as this feeble example--our best attempt to find one--illustrates.

Chapter 11: Integrity Constraints I

WHEN / THEN and PACKED ON both required

```
VAR  S_STATUS_DURING RELATION
{ S#          S#,
  STATUS  INTEGER,
  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.

March 2004 (c) Hugh Darwen

Here is the promised shorthand to cover the normal case, where both PACKED ON (DURING) and WHEN UNPACKED ON (DURING) THEN KEY ... are both required. Yet another application of the USING construct. (And there are more to come, when we deal with updating.)

Chapter 12: Integrity Constraints II

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*.

March 2004

That the requirements fall neatly into three groups of three is partly an accident of our chosen example. However, the method of grouping follows a general pattern that can be applied in any database design.

Chapter 12: Integrity Constraints II

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*

These are the three that relate to a supplier being under contract.

If we were recording suppliers' names and cities as well as their statuses, then Requirement R3 would be accompanied by two more similar requirements relating to name and city.

This sets the theme, which recurs with some subtle variations, as we are about to see ...

Chapter 12: Integrity Constraints II

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*

Requirement R4 is obviously of the same kind as Requirement R1, but here it addresses contradiction as well as redundancy. We don't want more than one tuple indicating that S1 has status 20 on day 1, for example, nor do we want one tuple showing S1 as having status 20 on day 1 and another showing S1 as having status 30 on day 1.

Requirement R6 is not only similar in kind to Requirement R3: it is the inverse of R3.

Chapter 12: Integrity Constraints II

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*

The variation here is that Requirement R9, unlike R3 and R6, is not accompanied by its inverse: a supplier who is unable to supply anything on a certain day is permitted to be under contract on that day.

Chapter 12: Integrity Constraints II

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 )
```

March 2004

This slide show Tutorial D constraint declarations that are needed to meet the nine requirements in the "semitemporal" counterpart of our database. Perhaps there is no compelling need for any new shorthands yet.

IS_EMPTY (*rel expr*) is Tutorial D's shorthand hand for *rel expr* { } = TABLE_DUM (the relation with no attributes and no tuples).

Chapter 12: Integrity Constraints II

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
```


And here are the Tutorial D constraint declarations, using shorthands we have already seen, to handle the case where all the relvars are "during" ones--in other words, where horizontal decomposition has not been needed.

Chapter 12: Integrity Constraints II

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 is_subset_of T2 ▸
```

(Note U_ form of relational comparison operator)

March 2004 (c) Hugh Darwen

You can study these constraints if you really want to satisfy yourself that they are correct and do the required job, but the whole point of this slide is to show what a compelling case there is for some much more powerful shorthand than any we have yet introduced.

Tutorial D features used here include WITH, which assigns names to expressions, allowing you to break down a complicated expression into several parts, and the "is subset of" relational comparison operator, for which Tutorial D uses the usual mathematical symbol, not available in the technology used to make these slides!

Chapter 12: Integrity Constraints II

Meeting the Nine Requirements (c): 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.

March 2004 (c) Hugh Darwen

And here are the proposed shorthands. SINCE_FOR associates an attribute of a point type (such as DATE, as here) with another attribute in an intuitive way, and HISTORY_IN associates a "during" relvar with that "since" attribute in an equally intuitive way.

All the constraints we have described under "the nine requirements" are implicitly declared by these uses of SINCE_FOR and HISTORY_IN.

Chapter 13: Database Queries

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 }
```

March 2004

Recall that a virtual relvar is Tutorial D's counterpart of SQL's "updatable view".

The one illustrated here provides a much more convenient target for the familiar database updating operations than the "since" and "during" relvars, and also factors out a subexpression that is likely to be required in very many queries.

Its form is common to all horizontal decompositions, which makes it possible to conceive of a shorthand for generating it, as we shall eventually see in the next chapter.

Chapter 14: Database Updates

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 able to supply part P4 on day 2".

Remove the proposition "Supplier S6 was able to supply part P3 from day 3 to day 5".

Replace the proposition "Supplier S2 was able to supply part P5 from day 3 to day 4" by the proposition "Supplier S2 was able to supply part P5 from day 2 to day 4".

Inevitable conclusion is need for U_update operators ...

Chapter 14: Database Updates

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 ...

March 2004 (c) Hugh Darwen

":=" is **Tutorial D**'s **assignment** operator.

Even with these U_ update operators, correctly applying required updates to a horizontally decomposed database can be excruciatingly difficult. We really need to be able to use that virtual relvar in which current state and history are combined, as a target of updates, so that the system can take care of special needs such as, for example, data deleted from the "since" relvar" being appropriately added to the corresponding "during" relvar.

This topic is not included in the course.

Chapter 14: Database Updates

S_DURING

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

Replace the propositon "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*]

Chapter 14: Database Updates

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# } ;
```

This topic is not included in the course.

Chapter 15: Stated Times and Logged Times

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.]

This topic is not included in the course.

Chapter 15: Stated Times and 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.

These topics are not included in the course.

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.

These topics are not included in the course.

Temporal Data and The Relational Model

Hugh Darwen
HD@TheThirdManifesto.com
www.TheThirdManifesto.com

The End