

Time and Relational Theory: Temporal Databases in the Relational Model and SQL

a technical seminar for database professionals

based on the book of the same name

*by C. J. Date, Hugh Darwen, and Nikos A. Lorentzos
(Morgan Kaufmann, 2014: <http://www.store.elsevier.com>)*

by

C. J. Date

TIME AND RELATIONAL THEORY

Temporal Databases
in the Relational Model
and SQL

C.J. Date

Hugh Darwen

Nikos A. Lorentzos

MK
Morgan Kaufmann



PREAMBLE :

Temporal DBMS: A DBMS that provides special facilities for storing, querying, and updating historical and/or future data

- Secondary storage media now cheap enough to support large volumes of temporal data ... So, data warehouse proliferation
- Hence temporal data problems, and need for solutions
- SQL:2011 includes some temporal features
- Vendors have begun to support those features

PREAMBLE (cont.) :

- So the time dimension is becoming increasingly important
- No one has done it right yet!
- We've been lobbying to get it done right ...

“If you do it the stupid way, you will have to do it again”

—Gregory Chudnovsky

- In particular: The relational model is ***the right and proper foundation*** for temporal support

Requires **NO** extension / revision / violations!

APPLICATION SCENARIOS :

*From “A Matter of Time: Temporal Data Management in DB2 10”,
by Cynthia M. Saracco, Matthias Nicola, and Lenisha Gandhi
(2012) /* with permission */ :*

1. An internal audit requires a financial institution to report on changes made to a client's records during the past five years.

APPLICATION SCENARIOS :

*From “A Matter of Time: Temporal Data Management in DB2 10”,
by Cynthia M. Saracco, Matthias Nicola, and Lenisha Gandhi
(2012) /* with permission */ :*

1. An internal audit requires a financial institution to report on changes made to a client's records during the past five years.
2. A pending lawsuit prompts a hospital to reassess its knowledge of a patient's medical condition **just before a new treatment was ordered.**

APPLICATION SCENARIOS :

*From “A Matter of Time: Temporal Data Management in DB2 10”,
by Cynthia M. Saracco, Matthias Nicola, and Lenisha Gandhi
(2012) /* with permission */ :*

1. An internal audit requires a financial institution to report on changes made to a client's records during the past five years.
2. A pending lawsuit prompts a hospital to reassess its knowledge of a patient's medical condition just before a new treatment was ordered.
3. A client challenges an insurance agency's resolution of a claim involving a car accident. The agency needs to determine the policy's terms in effect when the accident occurred.

-
4. An online travel agency wants to detect inconsistencies in itineraries. For example, if someone books a hotel in Rome for eight days and reserves a car in New York for **three of those days**, the agency would like to flag the situation for review.

-
4. An online travel agency wants to detect inconsistencies in itineraries. For example, if someone books a hotel in Rome for eight days and reserves a car in New York for three of those days, the agency would like to flag the situation for review.
 5. A retailer needs to ensure that no more than one discount is offered for a given product during any period of time.

-
4. An online travel agency wants to detect inconsistencies in itineraries. For example, if someone books a hotel in Rome for eight days and reserves a car in New York for three of those days, the agency would like to flag the situation for review.
 5. A retailer needs to ensure that no more than one discount is offered for a given product during any period of time.
 6. A client inquiry reveals a data entry error involving the three month introductory interest rate on a credit card. The bank needs to **retroactively** correct the error (and compute a new balance, if necessary).

DEVELOPMENT BENEFITS :

Also from “A Matter of Time”—a coding comparison (edited here):

The DB2 temporal support reduced coding requirements by more than 90% over homegrown implementations. Implementing just the core logic in SQL stored procedures or Java required 16 and 45 times as many lines of code, respectively, as the equivalent SQL statements using the DB2 temporal features. Also, it took less than an hour to develop and test those DB2 statements. By contrast, the homegrown approaches required 4-5 weeks to code and test, and they provided only a subset of the temporal support built into DB2. Thus, providing truly equivalent support through a homegrown implementation would likely take months.

AGENDA :

I. A review of relational concepts

II. Laying the foundations

III. Building on the foundations

IV. SQL support

V. Appendixes

PART I :

A review of relational concepts:

1. Types and relations
2. Relational algebra
3. Relation variables

BREAK :

Next = types

THE RUNNING EXAMPLE : SUPPLIERS AND SHIPMENTS

/ nontemporal version—sample values */*

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

Note supplier S5 in particular!

Potential shipments:
Supplier SNO is *currently*
able to supply part PNO

SP	SNO	PNO
	S1	P1
	S1	P2
	S1	P3
	S1	P4
	S1	P5
	S1	P6
	S2	P1
	S2	P2
	S3	P2
	S4	P2
	S4	P4
	S4	P5

DEFINITIONS */* Tutorial D—see next page */*:

```
TYPE SNO    ... ;           /* assume types INTEGER */
TYPE NAME   ... ;           /* and CHAR are built in   */
TYPE PNO    ... ;           /* (also BOOLEAN)        */
```

VAR S RELATION

```
{ SNO    SNO ,
  SNAME  NAME ,
  STATUS INTEGER ,
  CITY   CHAR }
KEY { SNO } ;
```

VAR SP RELATION

```
{ SNO SNO ,
  PNO PNO }
KEY { SNO , PNO }
FOREIGN KEY { SNO }
REFERENCES S ;
```

Note: RELATION here should really be *BASE RELATION* ... Definitions like those above are simplified throughout this presentation

THE THIRD MANIFESTO :

C. J. Date and Hugh Darwen:

Databases, Types, and the Relational Model:

The Third Manifesto (3rd edition, Addison-Wesley, 2007)

- Proposal for future direction of data and DBMSs
- **D** = any language that conforms to *Manifesto* principles
/ generic name */*
- **Tutorial D** = a particular **D**, used in *Manifesto* book and elsewhere as a basis for examples
/ though actually I'll be using a slight variant on conventional **Tutorial D** in this presentation */*

See www.thethirdmanifesto.com for further information, including free downloads for *Rel* (prototype implementation)

TYPES :

- Relations are defined over *types* (*aka* domains)
- A type is **a named set of values** (all possible values of some particular kind)—e.g.:
 - Set of all possible integers (INTEGER)
 - *Set of all possible character strings (CHAR)*
 - Set of all possible supplier numbers (SNO)
 - Set of all possible names (NAME)
- Types can be **user defined** or **system defined** (= built in)

-
- Every **value** is of some type (in fact, *exactly one* type*)

* *Except possibly if type inheritance is supported (see later)*

-
- Every **value** is of some type (in fact, *exactly one* type*)
 - Every **variable**, every **attribute** of every relation, every **operator** that returns a result, and every **parameter** of every operator is **declared** to be of some type

* *Except possibly if type inheritance is supported (see later)*

-
- Every **value** is of some type (in fact, *exactly one* type*)
 - Every **variable**, every **attribute** of every relation, every **operator** that returns a result, and every **parameter** of every operator is declared to be of some type
 - Every **expression** denotes some value and is of some type
= type of value in question = type of value returned by outermost operator

* *Except possibly if type inheritance is supported (see later)*

-
- Every **value** is of some type (in fact, *exactly one* type*)
 - Every **variable**, every **attribute** of every relation, every **operator** that returns a result, and every **parameter** of every operator is declared to be of some type
 - Every **expression** denotes some value and is of some type
= type of value in question = type of value returned by outermost operator
 - **Any** type can be used for declaring variables etc.

* *Except possibly if type inheritance is supported (see later)*

ASIDE :

Foregoing remarks re operators need some refinement to take care of *polymorphic* operators

Operator *Op* is *polymorphic* if it's defined in terms of some parameter *P* and arguments corresponding to *P* can be of different types on different invocations. For example:

- Equality “=” */* applies to every type */*
- Assignment “:=” */* applies to every type */*
- Aggregate operators (MAX, MIN, etc.)
- Relational operators (JOIN, etc.)
- *Interval operators* */* see later */*
- Etc., etc. ... */* end of aside! */*

TYPES (cont.) :

- Types can be **scalar*** or **nonscalar** /* *informal distinction* */
 - Scalar = not nonscalar (!)
 - Nonscalar = set of user visible components
 - *Relation types* are nonscalar
- But scalar types do have **possible representations** (“possreps”), which do have user visible components
... Don’t get confused!

* *Aka “encapsulated” (deprecated terminology)*

“POSSREPS” :

TYPE QTY /* *quantities* */

POSSREP QPR { Q INTEGER } ;

“POSSREPS” :

TYPE QTY /* *quantities* */

POSSREP QPR { Q INTEGER } ;

TYPE POINT /* *geometric points in 2D space* */

POSSREP CARTESIAN { X RATIONAL , Y RATIONAL }

POSSREP POLAR { R RATIONAL , THETA RATIONAL } ;

“POSSREPS” :

TYPE QTY /* quantities */

POSSREP QPR { Q INTEGER } ;

TYPE POINT /* geometric points in 2D space */

POSSREP CARTESIAN { X RATIONAL , Y RATIONAL }

POSSREP POLAR { R RATIONAL , THETA RATIONAL } ;

Physical representations aren't necessarily same as any declared *possible* representation

Default: Unnamed possrep inherits type name

Every possrep declaration causes “automatic” definition of following operators:

SELECTORS AND THE_ OPERATORS :

- **Selector operators** (same name as possrep): Specify or *select* a value of the type by supplying a value for each component of the possrep: e.g.,

CARTESIAN (5.0 , 2.5)

/ literal */*

CARTESIAN (X1 , Y1)

POLAR (2.7 , 1.0)

/ literal */*

SELECTORS AND THE OPERATORS :

- **Selector operators** (same name as possrep): Specify or *select* a value of the type by supplying a value for each component of the possrep: e.g.,

CARTESIAN (5.0 , 2.5)

```
/* literal */
```

CARTESIAN (X1 , Y1)

POLAR (2.7 , 1.0)

```
/* literal */
```

- **THE_operators** (one for each component of the possrep):
Access possrep components of values of the type: e.g.,

THE_X (P)

THE_R (P)

```
THE_Y ( exp )      /* exp = point valued expression */
```

MORE GENERALLY :

Important to understand that the type concept includes associated notion of the **operators** that can legally be applied to values and variables of the type in question

Such values and variables can be operated on *solely* by means of the operators defined for that type

E.g., system defined type INTEGER:

- System defines “:=”, “=”, “<”, etc., for assigning and comparing integers
- And “+”, “*”, etc., for arithmetic on integers
- But *not* “| |”, SUBSTR, etc.

E.g., user defined type SNO:

- Type definer defines “:=”, “=”, and maybe “<” etc., for assigning and comparing supplier numbers
- But *not* “+”, “*”, etc.

E.g., user defined type QTY:

- Etc., etc.

ASIDE :

User defined operators: Not relevant to this presentation until much later, but here are a couple of examples:

```
OPERATOR ABS ( N INTEGER ) RETURNS INTEGER ;  
    RETURN ( IF N ≥ 0 THEN +N ELSE -N END IF ) ;  
END OPERATOR ;
```


ASIDE :

User defined operators: Not relevant to this presentation until much later, but here are a couple of examples:

```
OPERATOR ABS ( N INTEGER ) RETURNS INTEGER ;  
    RETURN ( IF N ≥ 0 THEN +N ELSE -N END IF ) ;  
END OPERATOR ;
```

```
OPERATOR DIST ( P1 POINT , P2 POINT ) RETURNS RATIONAL ;  
    RETURN ( SQRT ( ( THE_X ( P1 ) - THE_X ( P2 ) ) ↑ 2  
                    + ( THE_Y ( P1 ) - THE_Y ( P2 ) ) ↑ 2 ) ) ;  
END OPERATOR ;
```

Aside: ABS and DIST are *read-only* operators (return a value)

Here by contrast is an *update* operator ... Given point (x,y), REFLECT replaces it by inverse point (-x,-y):

```
OPERATOR REFLECT ( P POINT ) UPDATES { P } ;  
    P := POINT ( - THE_X ( P ) , - THE_Y ( P ) ) ;  
        /* POINT selector invocation ... takes two          */  
        /* arguments (unlike SNO selector earlier)          */  
END OPERATOR ;
```

Doesn't return a value (no RETURN) ... Argument corresp. to P must be a variable specifically ... **End of aside.**

TYPES : *recap*

A **type** is a *named set of values* (i.e., all possible values of the type in question), along with an associated set of *operators* that can be applied to values and variables of the type in question

System or user defined

Types (and hence values and variables of the type in question) can be arbitrarily complex */* except for attributes of DB relations: no pointers, no self reference */*

Definition includes specification of set of legal values of the type (*type constraint*: see next page)

TYPE CONSTRAINTS :

Possrep is an *a priori* constraint—but can specify additional constraints: e.g.,

```
TYPE QTY  /* quantities */
```

```
POSSREP { Q INTEGER CONSTRAINT Q ≥ 0 AND Q ≤ 5000 } ;
```

So QTY(500) succeeds ... QTY(6000) fails

TYPE CONSTRAINTS :

Possrep is an *a priori* constraint—but can specify additional constraints: e.g.,

```
TYPE QTY  /* quantities */
```

```
POSSREP { Q INTEGER CONSTRAINT Q ≥ 0 AND Q ≤ 5000 } ;
```

So QTY(500) succeeds ... QTY(6000) fails

```
TYPE POINT /* geometric points in 2D space, restricted */
```

```
POSSREP CARTESIAN { X RATIONAL , Y RATIONAL  
                    CONSTRAINT SQRT ( X ** 2 + Y ** 2 ) ≤ 100.0 }
```

```
POSSREP POLAR { R RATIONAL , THETA RATIONAL  
               CONSTRAINT R ≤ 100.0 } ;
```

TYPES (cont.) :

Distinguish type vs. (physical) representation ...

But every type has at least one declared *possible* representation (“possrep”) ...

And one selector and one set of THE_ operators for each possrep

Values and variables can be operated upon *solely* by means of operators defined for the corresponding type ... Those operators **must** include “=” and “:=”

- Plus certain operators needed in connection with type inheritance, beyond the scope of this presentation

BREAK :

Next = relations

PART I cont. :

A review of relational concepts:

1. Types and **relations**
2. Relational algebra
3. Relation variables

“RELATION” DEFINED :

A *relation* consists of a *heading* and a *body* ...

The *heading* is a set of *attribute name : type name* pairs ...

The *body* is a set of tuples conforming to that heading

Can be pictured as a table: e.g.,

SNO : SNO	SNAME : NAME	STATUS : INTEGER	CITY : CHAR
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

MORE PRECISELY :

A **heading** H is a set of n **attributes** ($n \geq 0$), each of the form $\langle A_i, T_i \rangle$, where (a) A_i is an *attribute name* and T_i is the corresponding *type name* and (b) the attribute names A_i are all distinct.

Given heading H , a **body** B conforming to H is a set of m **tuples** ($m \geq 0$), each consisting of a set of n components (one for each A_i in H), where (a) each such component consists of attribute $\langle A_i, T_i \rangle$ and a value v_i of type T_i , and (b) v_i is the *attribute value* for attribute A_i within the tuple in question.

The combination $r = \langle H, B \rangle$ is a **relation**. The values m and n are the **cardinality** and the **degree**, respectively, of relation r .

POINTS ARISING :

- Type names usually omitted from tabular pictures

POINTS ARISING :

- Type names usually omitted from tabular pictures
- n -ary relations for arbitrary nonnegative integer n ... unary, binary, ternary, etc. ... **Don't forget *nullary*!**

POINTS ARISING :

- Type names usually omitted from tabular pictures
- n -ary relations for arbitrary nonnegative integer n ... unary, binary, ternary, etc. ... **Don't forget *nullary*!**
- No duplicate tuples

POINTS ARISING :

- Type names usually omitted from tabular pictures
- n -ary relations for arbitrary nonnegative integer n ... unary, binary, ternary, etc. ... **Don't forget *nullary*!**
- No duplicate tuples
- No tuple ordering top to bottom

POINTS ARISING :

- Type names usually omitted from tabular pictures
- n -ary relations for arbitrary nonnegative integer n ... unary, binary, ternary, etc. ... **Don't forget *nullary*!**
- No duplicate tuples
- No tuple ordering top to bottom
- No attribute ordering left to right

POINTS ARISING :

- Type names usually omitted from tabular pictures
- n -ary relations for arbitrary nonnegative integer n ... unary, binary, ternary, etc. ... **Don't forget nullary!**
- No duplicate tuples
- No tuple ordering top to bottom
- No attribute ordering left to right
- Every tuple contains exactly one value (of the pertinent type) for each attribute ... i.e., relations are always *normalized*, aka *first normal form*, 1NF

POINTS ARISING :

- Type names usually omitted from tabular pictures
- n -ary relations for arbitrary nonnegative integer n ... unary, binary, ternary, etc. ... **Don't forget *nullary*!**
- No duplicate tuples
- No tuple ordering top to bottom
- No attribute ordering left to right
- Every tuple contains exactly one value (of the pertinent type) for each attribute ... i.e., relations are always *normalized*, aka *first normal form*, 1NF
- **No nulls !!!** (nulls aren't values)

POINTS ARISING (cont.) :

- Every subset of a body is a body
- Every subset of a heading is a heading
- Every subset of a tuple is a tuple

ASIDE :

Let A and B be sets. Then:

“ A is a subset of B ” (or “ A is included in B ”), written $A \subseteq B$, means every element of A is also an element of B

“ A is a *proper* subset of B ” (or “ A is *properly* included in B ”), written $A \subset B$, means A is a subset of B but there exists at least one element of B that isn’t also an element of A

Every set is a subset of itself; no set is a proper subset of itself

“ B is a superset of A ” (or “ B includes A ”), written $B \supseteq A$, means the same as “ A is a subset of B ”

“ B is a proper superset of A ” (or “ B properly includes A ”), written $B \supset A$, means the same as “ A is a proper subset of B ”

Every set is a superset of itself; no set is a proper superset of itself

TUPLE EQUALITY /* important! */ :

- Two tuples **equal** iff (*= if and only if*)
 - Same attributes (i.e., same attribute name : type name pairs)
 - And attributes with same name have same attribute value

I.e., iff they're *the same tuple* !!!
- Two tuples are **duplicates** iff they're equal
- *MANY* features of the relational model rely on the above

NULLARY RELATIONS :

Empty heading is a valid heading ... So a relation can be of degree zero!

NULLARY RELATIONS :

Empty heading is a valid heading ... So a relation can be of degree zero!

(Such relations are a little hard to draw)

NULLARY RELATIONS :

Empty heading is a valid heading ... So a relation can be of degree zero!

(Such relations are a little hard to draw)

Can a relation with no attributes have any tuples?

NULLARY RELATIONS :

Empty heading is a valid heading ... So a relation can be of degree zero!

(Such relations are a little hard to draw)

Can a relation with no attributes have any tuples?

Yes, it can have AT MOST ONE TUPLE (the 0-tuple)

One tuple: TABLE_DEE */* DEE for short */*

No tuples: TABLE_DUM */* DUM for short */*

NULLARY RELATIONS :

Empty heading is a valid heading ... So a relation can be of degree zero!

(Such relations are a little hard to draw)

Can a relation with no attributes have any tuples?

Yes, it can have AT MOST ONE TUPLE (the 0-tuple)

One tuple: TABLE_DEE */* DEE for short */*

No tuples: TABLE_DUM */* DUM for short */*

Fundamentally important! (perhaps surprisingly)

POINTS ARISING (cont.) :

If relation r has heading $\{A_1, A_2, \dots, A_n\}$, then relation r is of **type** **RELATION** $\{A_1, A_2, \dots, A_n\}$

- **RELATION** is a **type generator** */* like ARRAY */*
- **RELATION** $\{A_1, A_2, \dots, A_n\}$ is a specific (generated) **relation type** */* like ARRAY [1..10] OF INTEGER */*

A relation is a (nonscalar) **value!**

Similarly for tuples, mutatis mutandis

POINTS ARISING (cont.) :

Relation types are *not* defined by means of TYPE statements as such ... So no possrep and no THE_ ops as such ... But there *is* an associated selector: e.g.,

RELATION

```
{ TUPLE { SNO SNO ( 'S1' ) , PNO PNO ( 'P1' ) , QTY 300 } ,  
  TUPLE { SNO SNO ( 'S1' ) , PNO PNO ( 'P2' ) , QTY 200 } ,  
  TUPLE { SNO SNO ( 'S1' ) , PNO PNO ( 'P3' ) , QTY 400 } ,  
  .....  
  TUPLE { SNO SNO ( 'S4' ) , PNO PNO ( 'P5' ) , QTY 400 } }
```

/ This example is a relation literal ... */*

/ keyword RELATION does double duty in **Tutorial D** */*

Similarly for tuples, mutatis mutandis

RELATION VALUED ATTRIBUTES :

This will be very important later!

1. Attributes of relations can be of any type whatsoever
2. Relation types are types
3. In particular, therefore, *relation valued attributes* (RVAs) are legal—e.g.:

SNO	PNO_REL					
S1	<table><tr><th>PNO</th></tr><tr><td>P1</td></tr><tr><td>P2</td></tr><tr><td>..</td></tr><tr><td>P6</td></tr></table>	PNO	P1	P2	..	P6
PNO						
P1						
P2						
..						
P6						

S2	<table><tr><th>PNO</th></tr><tr><td>P1</td></tr><tr><td>P2</td></tr></table>	PNO	P1	P2
PNO				
P1				
P2				
..	...			
S5	<table><tr><th>PNO</th></tr><tr><td></td></tr></table>	PNO		
PNO				

Type = **RELATION { SNO SNO , PNO_REL RELATION { PNO PNO } }**

RELATIONS AND THEIR MEANING :

I.e., “intended interpretation” ...

Very important way of thinking about relations !!!

Heading represents a ***predicate*** (truth valued function):
e.g., for suppliers,

Supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY

Parameters SNO, SNAME, STATUS, CITY stand for values of the applicable types

Tuples represent **true propositions** (“instantiations” of the predicate that evaluate to TRUE), obtained by substituting arguments (= values of the applicable types) for the parameters: e.g.,

Supplier S1 is under contract, is named Smith, has status 20, and is located in city London

Etc., etc.

If otherwise valid tuple does *not* appear, we assume the corresponding proposition is false

The Closed World Assumption (CWA)

PREDICATES (cont.) :

Every relation has an associated predicate—including derived ones!

E.g., S { SNO , SNAME , STATUS } /* projection */:

There exists some city CITY such that supplier SNO is under contract, is named SNAME, has status STATUS, and is located in city CITY

Incidentally, this predicate has three parameters, not four—CITY is not a parameter but a “bound variable”

Similarly for join, restriction, etc.

PART I cont. :

A review of relational concepts:

1. Types and relations
2. Relational algebra
3. Relation variables

BREAK :

Next = relational algebra (I)

RENAME :

E.g., `SP RENAME { SNO AS SNUM }`

Result identical to SP except for renaming

SNUM	PNO
S1	P1
S1	P2
..	..

Note: DB as such remains unchanged!
... contrast ALTER TABLE in SQL

“Multiple rename”: e.g.,

`SP RENAME { SNO AS SNUM , PNO AS PNUM }`

RESTRICT :

E.g., `SP WHERE PNO = PNO ('P1')`

WHERE condition is a boolean expression in which (a) every attribute reference identifies some attribute of the pertinent relation and (b) there are no references to any other relation

Real languages (including both SQL and **Tutorial D**) allow WHERE conditions of arbitrary complexity, but technically such conditions are just shorthand

PROJECT :

E.g., $S \{ SNO , SNAME , CITY \}$

$S \{ ALL BUT STATUS \}$

Tutorial D supports this ALL BUT option wherever it makes sense (not just for projection as such)

We assume for simplicity that projection has very high precedence

UNION etc. :

Let relations $r1$ and $r2$ be of the same type T . Then:

$r1 \text{ UNION } r2$ returns the relation of type T with body the set of all tuples t such that t appears in $r1$ or $r2$ or both

$r1 \text{ INTERSECT } r2$ returns the relation of type T with body the set of all tuples t such that t appears in both $r1$ and $r2$

$r1 \text{ MINUS } r2$ returns the relation of type T with body the set of all tuples t such that t appears in $r1$ and not in $r2$

E.g., $S \{ SNO \} \text{ MINUS } SP \{ SNO \}$

Attribute renaming might be needed ahead of time

UNION and INTERSECT (not MINUS) are *commutative*, *associative*, and *idempotent*

Can therefore define *n*-adic versions of these ops:

UNION $\{r_1, r_2, \dots, r_n\}$ returns the relation of type T with body the set of all tuples t such that t appears in at least one of r_1, r_2, \dots, r_n

INTERSECT $\{r_1, r_2, \dots, r_n\}$ returns the relation of type T with body the set of all tuples t such that t appears in each of r_1, r_2, \dots, r_n

(But what if $n = 1$? Or 0?)

D_UNION / I_MINUS :

Let relations $r1$ and $r2$ be of the same type. Then:

If $r1$ and $r2$ have any tuples in common, then $r1$
D_UNION $r2$ is undefined; otherwise it reduces to $r1$
UNION $r2$

/ n-adic version can be defined too */*

If some tuple appears in $r2$ and not in $r1$, then $r1$
I_MINUS $r2$ is undefined; otherwise it reduces to $r1$
MINUS $r2$

JOINABILITY :

Relations $r1$ and $r2$ are **joinable** iff attributes with the same name are of the same type (formally, iff they're the very same attribute)

E.g., **suppliers and shipments**

Equivalently, $r1$ and $r2$ are joinable iff the set theory union of their headings is a valid heading

Joinability (a) doesn't apply just to join, (b) extends to any number of relations $r1, r2, \dots, rn$ */* must be pairwise joinable */*

JOIN :

Let relations $r1$ and $r2$ be joinable. Then:

$r1 \text{ JOIN } r2$ returns the relation of with heading the set theory union of the headings of $r1$ and $r2$, and body the set of all tuples t such that t is the set theory union of a tuple in $r1$ and a tuple in $r2$

E.g., $S \text{ JOIN } SP$

Attribute renaming might be needed ahead of time

INTERSECT and TIMES are special cases ... Supported explicitly mainly for psychological reasons

JOIN is *commutative*, *associative*, and *idempotent* ... Can therefore define an *n -adic version*:

JOIN $\{r_1, r_2, \dots, r_n\}$ is defined as follows:

If $n = 0$, the result is TABLE_DEE

if $n = 1$, the result is r_1

Otherwise, choose any two distinct relations r_i and r_j from the set r_1, r_2, \dots, r_n and replace them by the result of r_i JOIN r_j , and repeat this process until the set consists of just one relation r , which is the final result

MATCHING :

Many (most?) queries etc. involving join at all really require *semijoin* ... E.g.:

Get supplier information for suppliers who supply at least one parts:

S MATCHING SP

Shorthand for (S JOIN SP) { SNO , SNAME , STATUS , CITY }

$r1 \text{ MATCHING } r2 \equiv (r1 \text{ JOIN } r2) \{ A1, ..., An \}$
where { A1, ..., An } = heading of r1

/ so r1 and r2 must be joinable */*

NOT MATCHING :

Many (most?) queries etc. involving difference at all really require *semidifference* ... E.g.:

Get supplier information for suppliers who supply no parts at all:

S NOT MATCHING SP

Shorthand for S MINUS (S MATCHING SP)

$r1 \text{ NOT MATCHING } r2 \equiv r1 \text{ MINUS } (r1 \text{ MATCHING } r2)$

/ so r1 and r2 must be joinable */*

BY THE WAY :

If relations $r1$ and $r2$ are of the same type (not just joinable), then

$r1$ NOT MATCHING $r2$

degenerates to

$r1$ MINUS $r2$

E.g., consider $S \{ SNO \}$ NOT MATCHING $SP \{ SNO \}$

/ analogous remark NOT true of semijoin */*

BREAK :

Next = relational algebra (II)

EXTEND :

For each supplier, get full supplier information, together with a TRIPLE value that's equal to three times the supplier's status

EXTEND S : { TRIPLE := 3 * STATUS }

SNO	SNAME	STATUS	CITY	TRIPLE
S1	Smith	20	London	60
S2	Jones	10	Paris	30
S3	Blake	20	Paris	60
S4	Clark	20	London	60
S5	Adams	30	Athens	90

EXTEND r : { *attribute assignment commalist* }

EXTEND (“what if”) :

What if supplier status values were tripled?

EXTEND S : { STATUS := 3 * STATUS }

SNO	SNAME	TRIPLE	CITY
S1	Smith	60	London
S2	Jones	30	Paris
S3	Blake	60	Paris
S4	Clark	60	London
S5	Adams	90	Athens

EXTEND r : { *attribute assignment commalist* }

IMAGE RELATIONS :

Image relation = “image” in some relation of some tuple
(usually a tuple in some other relation)

E.g., image in SP of tuple in S for S4:

PNO
P2
P4
P5

(SP WHERE SNO = SNO ('S4')) { PNO }

Very useful and widely applicable concept! So we define a **shorthand** ... E.g.:

EXTEND S { SNO } : { PNO_REL := !!SP }

SNO	PNO_REL
S1	PNO
	P1
	P2
	..
	P6

↑
image in SP of "current tuple" in S{SNO}

S2	PNO
	P1
	P2
..	...
S5	PNO

GROUP AND UNGROUP :

r1

SNO	PNO
S2	P1
S2	P1
S3	P1
S4	P1
S4	P1
S4	P1

r2

SNO	PNO_REL
S2	PNO
	P1 P2
S3	PNO
	P2
S4	PNO
	P2 P4 P5

r1 GROUP { PNO } AS PNO_REL : *gives r2*

r2 UNGROUP PNO_REL : *gives r1*

r1 GROUP { PNO } AS PNO_REL : *gives r2*

r2 UNGROUP PNO_REL : *gives r1*

Exercise: What does this do?—

EXTEND *r1* { SNO } : { PNO_REL := !!*r1* }

r1 GROUP { PNO } AS PNO_REL : *gives* *r2*

r2 UNGROUP PNO_REL : *gives* *r1*

Exercise: What does this do?—

EXTEND *r1* { SNO } : { PNO_REL := !!*r1* }

GROUP and UNGROUP can be defined in terms of EXTEND

SUMMARIZATION :

For each supplier, get SNO and number of parts supplied:

```
EXTEND S { SNO } : { PCT := COUNT ( !!SP ) }
```

/ note (a) image relation reference, (b) aggregate operator invocation */*

SNO	PCT
S1	6
S2	2
S3	1
S4	3
S5	0

← *count of an empty set*

SUMMARIZATION (cont.) :

For each supplier city, get city and average status:

```
EXTEND S { CITY } : { AVT := AVG ( !!S , STATUS ) }
```

/ note AVG syntax ... other agg ops include SUM MAX MIN AND OR XOR */*

CITY	AVT
London	20
Paris	20
Athens	30

ASIDE :

Aggregate operator invocations can also appear in a WHERE condition (of course) ... in which case image relations will probably be involved once again:

Get suppliers with fewer than three shipments:

```
S WHERE COUNT ( !!SP ) < 3
```

```
/* result = S2, S3, S5 */
```

RELATIONAL COMPARISONS :

Must be able to test relations for equality, of course ...
e.g.:

$S \{ SNO \} = SP \{ SNO \} \quad /* \textit{FALSE} */$

Other important comparison ops:

$\neq \quad \subseteq \quad \subset \quad \supseteq \quad \supset$

 relational inclusion

Useful shorthands:

$IS_EMPTY (r)$

$IS_NOT_EMPTY (r)$

“WITH” SPECIFICATIONS :

Get pairs of supplier numbers such that the suppliers are colocated (i.e., in same city):

```
(( ( S RENAME { SNO AS XNO } ) { XNO , CITY } JOIN  
  ( S RENAME { SNO AS YNO } ) { YNO , CITY } )  
  WHERE XNO < YNO ) { XNO , YNO }
```

“WITH” SPECIFICATIONS :

Get pairs of supplier numbers such that the suppliers are colocated (i.e., in same city):

```
(( ( S RENAME { SNO AS XNO } ) { XNO , CITY } JOIN  
  ( S RENAME { SNO AS YNO } ) { YNO , CITY } )  
  WHERE XNO < YNO ) { XNO , YNO }
```

Or:

```
WITH ( t1 := ( S RENAME { SNO AS XNO } ) { XNO , CITY } ,  
      t2 := ( S RENAME { SNO AS YNO } ) { YNO , CITY } ,  
      t3 := t1 JOIN t2 ,  
      t4 := t3 WHERE XNO < YNO ) : /* assume “<” defined! */  
t4 { XNO , YNO }
```

RELATIONAL ALGEBRA :

- RENAME
- Restrict – WHERE
- Project – { }
- UNION, INTERSECT, MINUS; D_UNION, I_MINUS
- Joinability; JOIN
- MATCHING and NOT MATCHING
- EXTEND (and “what if”)
- Image relations
- GROUP and UNGROUP
- Summarization and aggregate ops
- Relational comparisons ... and **closure** (?)
- (WITH specifications)

PART I cont. :

A review of relational concepts:

1. Types and relations
2. Relational algebra
3. Relation variables

BREAK :

Next = relation variables

RELATION VARIABLES :

- Relation values vs. relation variables
- Relational assignment
- Keys and foreign keys
- General DB constraints
- Views
- The relational model defined

RELATION VALUES vs. RELATION VARIABLES :

Historically there has been much confusion between relations as such (i.e., relation **values**) and relation **variables**

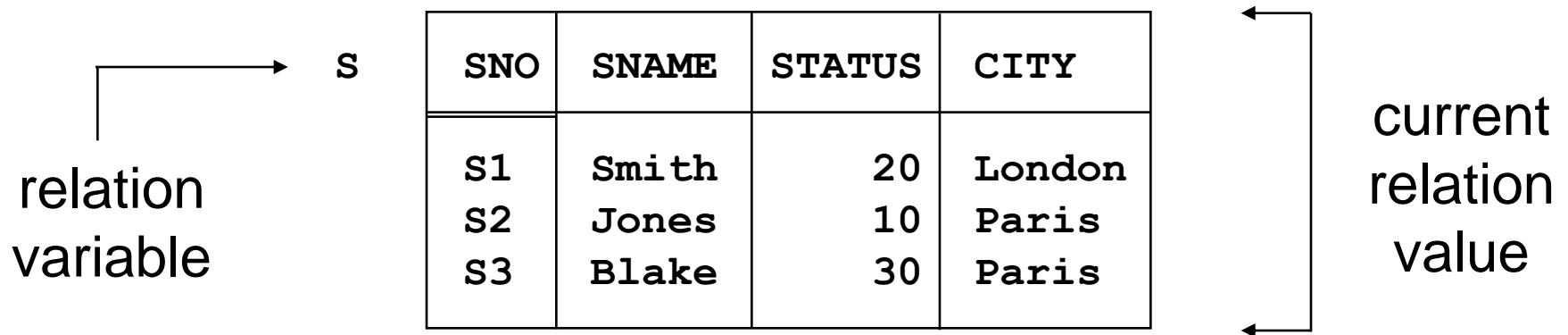
Consider: `DECLARE N INTEGER ...` — arbitrary programming language

N is an integer *variable* whose *values* are integers per se

Likewise: `CREATE TABLE T ...` — SQL

T is a relation *variable* whose *values* are relations per se */* ignoring SQL quirks */*

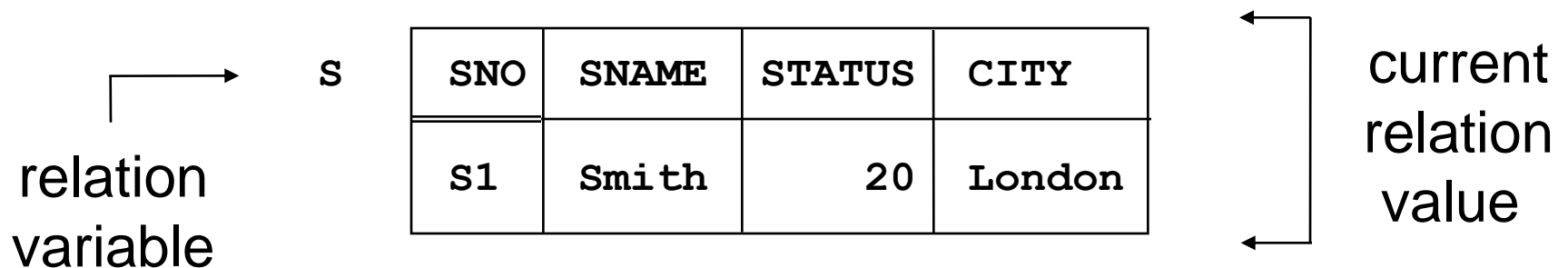
For example:



DELETE S WHERE CITY = 'Paris' ;

Shorthand for:

S := S MINUS (S WHERE CITY = 'Paris') ; */* explicit assignment */*



HENCE :

- **INSERT / DELETE / UPDATE** are all shorthand for some relational assignment, and—by definition—they all assign some relation *value* to some relation *variable*
- A relation variable, or **relvar** for short, is a variable whose permitted values are relation values, or **relations** for short
- ***Relvar predicate*** for relvar R = predicate common to all of the relations r that are possible values for R
- **Closed World Assumption**: Tuple t appears in relvar R at time T iff t satisfies predicate for R at time T

RELATIONAL ASSIGNMENT :

- $R := rx$ /* generic form */
- “INSERT R rx ” is shorthand for:
 $R := R \text{ UNION } rx$
- But what about “inserting a tuple that already exists” ??? ...

“D_INSERT R rx ” is shorthand for:

$R := R \text{ D_UNION } rx$

-
- “DELETE $R \text{ } rx$ ” is shorthand for:

$R := R \text{ MINUS } rx$

- But what about “deleting a tuple that doesn’t exist” ??? ...

“I_DELETE $R \text{ } rx$ ” is shorthand for:

$R := R \text{ I_MINUS } rx$

- “DELETE [R] R WHERE bx ” is shorthand for:

$R := R \text{ WHERE NOT } (bx)$

-
- “UPDATE R WHERE $bx : \{ A := ax \}$ ” is shorthand for:

$R := (R \text{ WHERE NOT } (bx))$
UNION
 $(\text{EXTEND } (R \text{ WHERE } bx) : \{ A := ax \} ;$

- I.e., effectively:

$R := (R \text{ MINUS } old) \text{ UNION } new ;$

The Assignment Principle:

After assignment of v to V , $v = V$ must give TRUE

Very simple ... but far reaching consequences!

EVERY RELVAR HAS AT LEAST ONE KEY (why?) :

Let K be a subset of the heading of relvar R . Then K is a **key** (aka **candidate key**) for R iff:

1. *Uniqueness:*
No possible value of R has two distinct tuples with the same value for K
2. *Irreducibility:*
No proper subset of K has the uniqueness property

E.g., {SNO} and {SNO,PNO} for relvars S and SP, respectively /* note the braces */ ... SP is “all key”

POINTS ARISING :

We don't insist on *primary* keys as such, but do usually follow PK discipline ourselves (marked by double underlining)

Key values are *tuples*! Key uniqueness relies on *tuple equality*!

Number of attributes is *degree* of key

Keys apply to *relvars*, not relations (why?)

Note: System can enforce uniqueness but can't enforce irreducibility */* why do we want irreducibility, anyway? */*

A subset SK of the heading of R that has the uniqueness property but not necessarily the irreducibility property is a **superkey** for R

/ loosely, a superkey is a superset of a key */*

Uniqueness of SK implies that the **functional dependency** $SK \rightarrow A$ holds in R for all subsets A of the heading of R */* see next page */*

I.e., there are always “arrows out of superkeys”

If X and Y are subsets of the heading of R , the **functional dependency** (FD) $X \rightarrow Y$ holds in R iff, in every relation r that's a possible value of R , whenever two tuples have the same value for X , they also have the same value for Y

E.g., suppose that if two suppliers are in the same city at the same time, they must have the same status at that time.

Then the following FD holds in S :

$\{ \text{CITY} \} \rightarrow \{ \text{STATUS} \}$ */* note the braces */*

SOME RELVARS HAVE FOREIGN KEYS :

- Let $R1$ and $R2$ be relvars, not necessarily distinct, and let K be a key for $R1$
- Let FK be a subset of the heading of $R2$ such that there exists a possibly empty sequence of attribute renamings on $R1$ that maps K into K' (say), where K' and FK contain exactly the same attributes
- Let $R2$ and $R1$ be subject to the constraint that, at all times, every tuple $t2$ in $R2$ has an FK value that's the K' value for some (necessarily unique) tuple $t1$ in $R1$ at the time in question
- Then FK is a **foreign key** (with the same *degree* as K); the associated constraint is a *referential constraint*; and $R2$ and $R1$ are the *referencing relvar* and the corresponding *referenced relvar*, respectively, for that constraint

E.g., {SNO} in relvar SP

Referential integrity rule: DB must never contain any unmatched FK values

Note reliance on tuple equality again ...

In practice, systems often allow a “cascade delete rule” to be specified in connection with foreign keys ... but for simplicity I’ll overlook any such possibility, most of the time

INTEGRITY CONSTRAINTS :

An integrity constraint is, loosely, a boolean expression that *must* evaluate to TRUE

Any attempt to update the DB in such a way as to cause some constraint to evaluate to FALSE ***must fail ...***

... and, in the relational model, fail **IMMEDIATELY !!!**
/ this is **The Golden Rule** */*

We've already talked about **key** constraints and **foreign key** constraints, but constraints of arbitrary complexity are possible, and do indeed occur in practice ("business rules")
... and they're **IMPORTANT!**

EXAMPLES :

1. Supplier status values must be in the range 1 to 100 inclusive:

CONSTRAINT CX1

IS_EMPTY (S WHERE STATUS < 1 OR STATUS > 100) ;

EXAMPLES :

1. Supplier status values must be in the range 1 to 100 inclusive:

```
CONSTRAINT CX1  
    IS_EMPTY ( S WHERE STATUS < 1 OR STATUS > 100 ) ;
```

Or:

```
CONSTRAINT CX1 AND ( S , STATUS ≥ 1 AND STATUS ≤ 100 ) ;
```

Note: AND here could reasonably be pronounced “for all”

2. Suppliers in London must have status 20:

CONSTRAINT CX2

IS_EMPTY (S WHERE CITY = 'London' AND STATUS \neq 20) ;

Or:

CONSTRAINT CX2 AND (S , CITY \neq 'London' OR STATUS = 20) ;

3. Supplier numbers must be unique:

```
CONSTRAINT CX3 COUNT ( S ) = COUNT ( S { SNO } ) ;
```

In practice would use KEY shorthand

-
4. Suppliers with status less than 20 aren't allowed to supply part P6:

```
CONSTRAINT CX4 IS_EMPTY  
  ( ( S JOIN SP ) WHERE STATUS < 20 AND PNO = 'P6' ) ;
```

/ **multirelvar constraint** ... CX1-CX3 were single-relvar constraints, or just **relvar constraints** for short */*

5. Every supplier number in SP must also appear in S:

CONSTRAINT CX5 SP { SNO } \subseteq S { SNO } ;

- Foreign key constraint from SP to S
- In practice would use FOREIGN KEY shorthand

/ note the explicit relational comparison in this example */*

6. Every supplier in London must supply part P1:

```
CONSTRAINT CX6 IS_EMPTY  
  ( ( S WHERE CITY = 'London' ) NOT MATCHING  
    ( SP WHERE PNO = PNO ( 'P1' ) ) ) ;
```

- If we insert a new London supplier, must insert a new shipment too ... but:
- INSERT on S first violates CX6; INSERT on SP first violates CX5 ... So:

The **multiple assignment** operator lets us carry out several assignments as a single operation, without any integrity checking being done until all assignments have been executed:

```
INSERT S RELATION { TUPLE { SNO SNO ( 'S9' ) , ... , CITY 'London' } } ,  
INSERT SP RELATION { TUPLE { SNO SNO ( 'S9' ) , PNO PNO ( 'P1' ) } } ;
```

Note comma separator ... One statement, not two!

Shorthand for:

```
S := ... , P := ... ;
```


SEMANTICS /* *slightly simplified* */ :

1. Evaluate source expressions
2. Execute individual assignments “simultaneously” *
3. Do integrity checking

No individual assignment depends on any other ... No way for the user to see an inconsistent state of the database between the two INSERTs, because notion of “between the two INSERTs” has no meaning ... Now no need for deferred checking at all! /* *not true in SQL, though* */

* *But assignments to the same target are done sequentially*

VIEWS :

Consider query “Get suppliers in London”:

```
S WHERE CITY = 'London'
```

Suppose this query is used repeatedly ... Can simplify matters by defining a **view** (“canned query”):

```
VAR LS VIRTUAL ( S WHERE CITY = 'London' ) KEY { SNO } ;
```

Query becomes:

```
LS
```

Views work because of closure!

The Principle of Interchangeability:

There must be no arbitrary and unnecessary distinctions between base and virtual relvars

Virtual relvars should “look and feel” just like base ones to the user

THE RELATIONAL MODEL :

1. An open ended collection of **types**, including in particular type BOOLEAN
2. A **relation type generator** and an intended interpretation for relations of types generated thereby
3. Facilities for defining **relation variables** of such generated relation types
4. A **relational assignment** operation for assigning relation values to such relation variables
5. A relationally complete, but otherwise open ended, collection of generic **relational operators** for deriving relation values from other relation values

PART I review :

A review of relational concepts:

1. Types and relations
2. Relational algebra
3. Relation variables

AGENDA :

I. A review of relational concepts

II. Laying the foundations

III. Building on the foundations

IV. SQL support

V. Appendixes

BREAK :

Next = time and the database

PART II :

Laying the foundations:

4. Time and the database
5. What's the problem?
6. Intervals
7. Interval operators
8. EXPAND and COLLAPSE
9. PACK and UNPACK I
10. PACK and UNPACK II
11. Generalizing the relational operators

TIME AND THE DATABASE :

- Timestamped propositions
- “Valid time” vs. “transaction time”
- Some fundamental questions

TEMPORAL DB :

- *Nontemporal DB* (i.e., DB as conventionally understood—sometimes most unfortunately called a “snapshot DB”):
 - Current data only—e.g., status of supplier S1 is currently (“now”) 20
- *Temporal DB*:
 - Historical data instead of or as well as current data—e.g., status of supplier S1 is currently 20 *AND* has been 20 ever since July 1st *AND* was 15 from April 5th to June 30th (etc., etc.)
 - If no DELETE or UPDATE then historical data *only*

A note on the research (there's been some controversy):

Two approaches: Treat temporal data as special and depart from relational principles?* ...

OR ...

Abide firmly by those principles?

Fortunately, the good guys won!

Added later: Well, sort of, anyway

* Especially *The Information Principle* (= DB contains relvars only)

CAVEAT :

Various questions regarding the nature of time—e.g.,

- Does time have a beginning or an end?
- Is time a continuum or is it divided into discrete quanta?
- How can we best characterize the concept *now* (*aka* “the *moving point now*”)?

—aren’t really DB questions as such! So I won’t try to answer them definitively, but just make what I hope are reasonable assumptions as we proceed ...

NOTE, HOWEVER, THAT (*important!*) :

- The (good) temporal DB research has led to certain INTERESTING GENERALIZATIONS ...
- I'll be touching on some of those generalizations from time to time */* pun intended */*
- However, I follow convention in sometimes referring to (e.g.) “temporal” ops, “temporal” relations, etc., even though the concepts are often not exclusive to temporal data as such

I also take “history” to include the future, where appropriate

TENTATIVE DEFINITIONS :

- If data = encoded representation of *propositions*, then temporal data = encoded representation of *timestamped* propositions ...
- **Temporal relation:** relation in which each tuple contains at least one timestamp (i.e., heading has at least one attribute of some timestamp type)
- **Temporal relvar:** relvar whose heading is that of a temporal relation
- **Temporal DB:** DB in which relvars are all temporal (?)

BUT ... !!!

“Temporal DB” concept as just defined isn’t very useful!
(Why not?)

So ... from this point forward, I take a temporal DB to be a DB that *contains* but *is not limited to* temporal relvars

Note: With one exception, all of the new ops and other constructs to be discussed are just *shorthand!*

Exception = **INTERVAL type generator** (see later)

CONSIDER THIS 2-TUPLE :

SNO	FROM
S1	July 1st, 2012

Possible interpretations include:

p1: Supplier S1 was placed under contract **on** July 1st, 2012

p2: Supplier S1 has been under contract **since** July 1st, 2012

p3: Supplier S1 was under contract **during** the interval from July 1st, 2012, to the present day

TIMESTAMPED PROPOSITIONS :

Any of $p1$, $p2$, $p3$ could be intended interpretation, depending on predicate for containing relation

Prepositions **on**, **since**, and **during** characterize the three interpretations

On = “**at** some instant” /* *not very interesting (?)* */

Since = “**ever** since”

During = “**throughout** the interval in question”

TIMESTAMPED PROPOSITIONS (cont.) :

But don't $p1$, $p2$, $p3$ really all say the same thing?

Well ... they need to be tightened up!

$p1$: Supplier S1 was most recently appointed on July 1st, 2012 (but the contract might subsequently have been terminated)

$p2$: Supplier S1 wasn't under contract on June 30th, 2012, but has been so ever since July 1st, 2012

$p3$: Supplier S1 wasn't under contract on June 30th, 2012, but has been so during the interval from July 1st, 2012, to the present day

$p2$ and $p3$ are equivalent to each other but not to $p1$

$p2$ and $p3$ are logically equivalent but significantly ***different in form*** ... Reverting to simpler versions for brevity:

$p2$: Supplier Sx has been under contract since date d

$p3$: Supplier Sx was under contract during the interval from date b to date e

Form of $p3$ can be used for historical records!
—which do typically involve intervals

Concept of ***DURING*** is important (all pervasive)

“VALID TIME” vs. “TRANSACTION TIME” :

- Can historical data be updated ???
- Well ... “historical data” in the DB represents, not history as such, but rather *our beliefs about* that history ... and beliefs **can** change
 - **Valid time** for p (updatable) : set of times at which (by our current knowledge) p is, was, or will be true
 - **Transaction time** for q (not updatable) : set of times at which q is *represented in DB* as being true

FOR EXAMPLE :

Let p be “Supplier S1 was under contract”

Suppose we currently believe this state of affairs held from July 1st, 2012, until May 1st, 2013, so we insert:

SNO	FROM	TO
S1	July 1st, 2012	May 1st, 2013

This tuple does **not** correspond to proposition p !—but to a *timestamped extension* of p

FROM/TO represents “valid time” for p —time when (according to our current beliefs) p represented a “true fact”

Later we discover date of appointment was **June** 1st, so we “update the tuple”:

SNO	FROM	TO
S1	June 1st, 2012	May 1st, 2013

Changes “valid time”, not p !

Later we discover S1 was never under contract at all and therefore delete the tuple— p now known to be false and has no “valid time” at all

Suppose tuple was INSERT'd at time $t1$
UPDATE'd at time $t2$
DELETE'd at time $t3$

Interval from $t1$ to $t3$ = “transaction time”—not for p , but for proposition “ p was true throughout some interval”

Interval from $t1$ to $t2$ = “transaction time” for timestamped extension of p with timestamp July 1st, 2012 - May 1st, 2013

Interval from $t2$ to $t3$ = “transaction time” for timestamped extension of p with timestamp June 1st, 2012 - May 1st, 2013

We'll revisit these concepts later ...

NOTE :

In general, valid times and transaction times are both ***sets*** of intervals, not just intervals per se

E.g., if q = timestamped extension of p with timestamp
June 1st, 2012 - May 1st, 2013

then transaction time for $q = \{ i \}$
where i = interval from t_2 to t_3

We'll revisit these concepts later ...

OBVIOUS BUT FAR REACHING ASIDE :

interval with begin time b and end time e

can be thought of as

set of all times t such that $b \leq t \leq e$

(where “ $<$ ” means “earlier than”)

SOME FUNDAMENTAL QUESTIONS :

Doesn't "all times t such that $b \leq t \leq e$ " mean we're dealing with infinite sets?

Assumptions:

- Timeline = finite contiguous sequence of discrete indivisible *time quanta*
- Time quantum = smallest time unit representable in the system or *chronon* (?)
- Interval = section of timeline (thus, also a finite contiguous sequence of discrete time quanta)

Propositions *p1-p3* seem to assume time quanta are *days* ...
Doesn't the system support time units down to (e.g.)
microseconds, or actually something much smaller?

E.g., if S1 was appointed on July 1st, what do we do about
the interval from the beginning of July 1st up to the very
instant of appointment?

*Distinguish time quanta vs. time units relevant for some
particular purpose (e.g., years, months, days, msecs)
= **time points** aka “points” aka “granules”*

- **Granularity** = “size” of applicable time points
= “size” of gap between adjacent points (?)

If we regard the timeline (for some given purpose) as a finite sequence of time points, each time point has a unique successor and a unique predecessor—right?

Yes—except for the points corresponding to “the end of time” and “the beginning of time”, of course

$p3$: Supplier S_x was under contract during the interval from date b to date e

If the corresponding relation contains 3-tuple—

SNO	FROM	TO
S1	July 1st, 2012	April 25th, 2013

—doesn't the CWA imply it must also contain:

SNO	FROM	TO
S1	July 2nd, 2012	April 24th, 2013

etc.,
etc.?

GOOD POINT !!!

Clearly, $p3$ needs to be tightened up:

$p3$: Supplier S_x was under contract on every day from date b to date e , but not on the day immediately before b , nor on the day immediately after e ... i.e., throughout the interval from b to e , but not throughout any interval that properly includes that interval /* see later */

- *Since* = “**ever since and not immediately before**”
- *During* = “**throughout and not immediately before or immediately after** (the interval in question)”

SOME ASSUMPTIONS :

Henceforth I assume that:

- No supplier can end one contract on one day and start another on the very next day
- No supplier can be under two distinct contracts at the same time
- Contracts can be open ended—i.e., a supplier can be currently under contract and the end date for that contract can be currently unknown

PART II cont. :

Laying the foundations:

4. Time and the database
5. What's the problem?
6. Intervals
7. Interval operators
8. EXPAND and COLLAPSE
9. PACK and UNPACK I
10. PACK and UNPACK II
11. Generalizing the relational operators

BREAK :

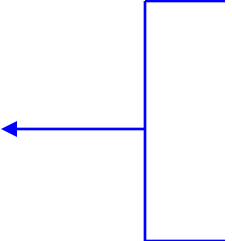
Next = what's the problem?

THE RUNNING EXAMPLE : SUPPLIERS AND SHIPMENTS

/ simplified version—sample values */*

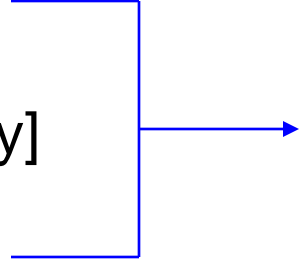
S	SNO
	S1
	S2
	S3
	S4
	S5

Supplier SNO is
[currently]
under contract



SP	SNO	PNO
	S1	P1
	S1	P2
	S1	P3
	S1	P4
	S1	P5
	S1	P6
	S2	P1
	S2	P2
	S3	P2
	S4	P2
	S4	P4
	S4	P5

Potential shipments:
Supplier SNO is [currently]
able to supply part PNO



SAMPLE CONSTRAINTS :

Consider *key and FK* constraints only (until further notice)

- {SNO} is sole key for S
- {SNO,PNO} is sole key for SP

relvar constraints

- {SNO} is foreign key in SP,
matching sole key of S

multirelvar
constraint

/ recall that we don't insist on primary keys as such, though we don't prohibit them either */*

SAMPLE QUERIES :

- **Query A:** Get supplier numbers of suppliers who are currently able to supply at least one part

SP { SNO }

/ projection */*

- **Query B:** Get supplier numbers of suppliers who are currently unable to supply any part at all

S { SNO } MINUS SP { SNO }

/ difference between two projections—first an identity */*

/ projection ... also, could have used NOT MATCHING */*

“SEMITEMPORALIZING” :

S_SINCE

SNO	SINCE
S1	d04
S2	d07
S3	d03
S4	d04
S5	d02

SP_SINCE

SNO	PNO	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

- “Granularity” = one day
- Assume day 1 immediately precedes day 2, etc., etc.

PREDICATES :

- S_SINCE:

Supplier SNO has been under contract ever since day SINCE, and not the day immediately before day SINCE

- SP_SINCE:

Supplier SNO has been able to supply part PNO ever since day SINCE, and not the day immediately before day SINCE

SAMPLE CONSTRAINTS :

Key and FK constraints are as for nontemporal version

But we also need to “augment” the FK constraint to say no supplier can supply any part before that supplier is under contract ...

```
CONSTRAINT XST1 IS_EMPTY
( ( ( SP_SINCE RENAME { SINCE AS SPS } ) JOIN
    ( S_SINCE RENAME { SINCE AS SS } ) )
  WHERE SPS < SS ) ;
```

```
/* if tuple sp in SP_SINCE references tuple s in S_SINCE, */
/* SINCE value in sp mustn't be less than that in s */
```

Would be nice to have some convenient shorthand ...

SAMPLE QUERIES :

Query A: Get supplier numbers of suppliers who are currently able to supply at least one part, together with the date since when they have been able to do so

```
EXTEND SP_SINCE { SNO } : { SINCE := MIN ( !!SP_SINCE , SINCE ) }
```

SNO	SINCE
S1	d04
S2	d08
S3	d08
S4	d04

minimum SINCE value in SP_SINCE
for “this” SNO in SP_SINCE

Query B: Get supplier numbers of suppliers who are currently unable to supply any part at all, together with the date since when they have been unable to do so

Supplier S5 is currently unable to supply any parts at all
... But we don't know the date since when S5 has been unable to supply any parts (insufficient information in the DB)—
DB is still only “**semitemporalized**”

I.e., we need to keep *historical records*!

FULLY TEMPORALIZING /* 1st attempt */:

S_FROM_TO

SNO	DFROM	DTO
S1	d04	d10
S2	d02	d04
S2	d07	d10
S3	d03	d10
S4	d04	d10
S5	d02	d10

SP_FROM_TO

SNO	PNO	DFROM	DTO
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	d02	d04
S2	P1	d08	d10
S2	P2	d03	d03
S2	P2	d09	d10
S3	P2	d08	d10
S4	P2	d06	d09
S4	P4	d04	d08
S4	P5	d05	d10

POINTS ARISING :

Assume for definiteness that today is $d10$

- Have shown $d10$ as TO value for each tuple pertaining to current state of affairs ... *BUT*:
- How can all of those $d10$'s become $d11$'s on the stroke of midnight? **See later!**

More tuples than before ... This fully temporal DB contains everything from semitemporal DB,* plus historical records regarding previous interval of time (from $d02$ to $d04$) during which S2 was also under contract and able to supply certain parts

* *Except that TO value for two of S4's shipments is earlier than today—i.e., those shipments are now “historical” instead of “current”*

PREDICATES :

- S_FROM_TO:

Supplier SNO was under contract throughout the interval from day FROM to day TO, and not throughout any interval that properly includes that interval

- SP_FROM_TO:

Supplier SNO was able to supply part PNO throughout the interval from day FROM to day TO, and not throughout any interval that properly includes that interval

SAMPLE CONSTRAINTS :

- Must prohibit FROM-TO pairs in which $TO < FROM$:

CONSTRAINT S_FROM_TO_OK

IS_EMPTY (S_FROM_TO WHERE $TO < FROM$) ;

CONSTRAINT SP_FROM_TO_OK

IS_EMPTY (SP_FROM_TO WHERE $TO < FROM$) ;

- Keys:

- For S_FROM_TO : {SNO, FROM} + {SNO, TO}

- For SP_FROM_TO : {SNO, PNO, FROM} + {SNO, PNO, TO}

But these constraints aren't sufficient!

SAMPLE CONSTRAINTS (cont.) :

- If S_FROM_TO contains (e.g.)—

SNO	FROM	TO
S1	<i>d04</i>	<i>d10</i>

—then it mustn't also contain (e.g.):

SNO	FROM	TO
S1	<i>d02</i>	<i>d06</i>

Note the **redundancy** in these two tuples

SAMPLE CONSTRAINTS (cont.) :

- These two tuples need to be merged into one ...

SNO	FROM	TO
S1	<i>d02</i>	<i>d10</i>

- Note that *not* merging the tuples would be as bad as permitting duplicates!—and would mean S_FROM_TO violated its own predicate
- Key constraint is insufficient to prohibit **overlapping** tuples

SAMPLE CONSTRAINTS (cont.) :

- If S_FROM_TO contains (e.g.)—

SNO	FROM	TO
S1	<i>d04</i>	<i>d10</i>

—then it mustn't also contain (e.g.):

SNO	FROM	TO
S1	<i>d02</i>	<i>d03</i>

SAMPLE CONSTRAINTS (cont.) :

- These two tuples need to be merged into one ...

SNO	FROM	TO
s1	<i>d02</i>	<i>d10</i>

- No redundancy as such, but ***circumlocution*** (and violation of predicate)
- Key constraint is insufficient to prohibit ***abutting*** tuples

SAMPLE CONSTRAINTS (cont.) :

CONSTRAINT XFT1

IS_EMPTY

```
(( ( S_FROM_TO RENAME { FROM AS F1 , TO AS T1 } ) JOIN
  ( S_FROM_TO RENAME { FROM AS F2 , TO AS T2 } ) )
 WHERE ( T1 ≥ F2 AND T2 ≤ F1 ) ) OR
        ( F2 = T1+1 OR F1 = T2+1 ) ) ;
```

Complicated !!!

“T1+1” ??? “T2+1” ???

Do we begin to see the problem ???

SAMPLE CONSTRAINTS (cont.) :

{SNO, FROM} is *not* a FK from SP_FROM_TO to S_FROM_TO

But if supplier *s* appears in SP_FROM_TO, then supplier *s* must appear in S_FROM_TO as well:

CONSTRAINT XFT2

$SP_FROM_TO \{ SNO \} \subseteq S_FROM_TO \{ SNO \};$

Example of an *inclusion dependency* (FK constraints are a special case)

SAMPLE CONSTRAINTS (cont.) :

But Constraint XFT2 isn't enough! ... If SP_FROM_TO shows supplier s as able to supply some part during some interval of time, then S_FROM_TO must show supplier s as under contract during that same interval of time

CONSTRAINT XFT3

```
COUNT ( SP_FROM_TO { SNO , FROM , TO } ) =  
COUNT ( ( ( SP_FROM_TO RENAME { FROM AS SPF , TO AS SPT } )  
                                                { SNO , SPF , SPT }  
JOIN  
( S_FROM_TO RENAME { FROM AS SF , TO AS ST } ) )  
      WHERE SF ≤ SPF AND ST ≥ SPT ) ;
```

Draw your own conclusions ...

SAMPLE QUERIES :

Query A: Get SNO-FROM-TO triples for suppliers who have been able to supply at least one part during at least one interval of time, where FROM and TO together designate a maximal interval during which supplier SNO was in fact able to supply at least one part

Query B: Get SNO-FROM-TO triples for suppliers who have been unable to supply any parts at all during at least one interval of time, where FROM and TO together designate a maximal interval during which supplier SNO was in fact unable to supply any part at all

You've got to be joking!

TO SUM UP :

“Temporal” constraints and queries—not to mention updates!—*can* be expressed, but they quickly get very complicated indeed

We need some carefully thought out and well designed shorthands ...

... which typically *don't* exist in today's commercial DBMSs, of course*

So let's investigate!

* *Most of what follows remains, sadly, unimplemented at this time*

PART II cont. :

Laying the foundations:

4. Time and the database
5. What's the problem?
6. Intervals
7. Interval operators
8. EXPAND and COLLAPSE
9. PACK and UNPACK I
10. PACK and UNPACK II
11. Generalizing the relational operators

BREAK :

Next = intervals

INTERVALS :

Crucial insight: Need to deal with *intervals as such* (i.e., as values in their own right), instead of as pairs of FROM-TO values

But what's an interval?

Consider proposition: “Supplier S1 was able to supply part P1
from day 4 to day 10”

Does “from day 4 to day 10” include day 4? day 10?

Given some interval i stretching from one point to another, we sometimes want to regard the specified points as part of i and sometimes not

$[d04:d10]$ — closed:closed = $d04\ d05\ d06\ d07\ d08\ d09\ d10$
 $[d04:d11)$ — closed:open = $d04\ d05\ d06\ d07\ d08\ d09\ d10$
 $(d03:d10]$ — open:closed = $d04\ d05\ d06\ d07\ d08\ d09\ d10$
 $(d03:d11)$ — open:open = $d04\ d05\ d06\ d07\ d08\ d09\ d10$

Closed:open is convenient and most often used in practice:

Thus, e.g., split $[d04:d11)$ immediately before, say, day 7 ...
result is $[d04:d07)$ and $[d07:d11)$

But **closed:closed** is most intuitive and I'll favor it throughout this presentation

FULLY TEMPORALIZING /* 2nd attempt */:

S_DURING

SNO	DURING
S1	[d04:d10]
S2	[d02:d04]
S2	[d07:d10]
S3	[d03:d10]
S4	[d04:d10]
S5	[d02:d10]

SP_DURING

SNO	PNO	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	[d02:d04]
S2	P1	[d08:d10]
S2	P2	[d03:d03]
S2	P2	[d09:d10]
S3	P2	[d08:d10]
S4	P2	[d06:d09]
S4	P4	[d04:d08]
S4	P5	[d05:d10]

PREDICATES :

- S_DURING:

Supplier SNO was under contract throughout the interval from the begin point of DURING to the end point of DURING inclusive, and not throughout any interval that properly includes that interval

- SP_DURING:

Supplier SNO was able to supply part PNO throughout the interval from the begin point of DURING to the end point of DURING inclusive, and not throughout any interval that properly includes that interval

IMMEDIATE BENEFITS :

- Sole keys:

- S_DURING : {SNO,DURING}

- SP_DURING : {SNO,PNO,DURING}

choice
no longer
arbitrary

- Don't need to worry about whether intervals (in previous version of DB) are open or closed with respect to FROM and TO

- $[d04:d10]$, $[d04:d11)$, $(d03:d10]$, $(d03:d11)$ are distinct “possreps” for the very same interval—don't need to know which, if any, is actual physical representation

Constraints to prohibit FROM-TO pairs in which $TO < FROM$ are now unnecessary (“ $FROM \leq TO$ ” is implicit)

Other constraints might be simplified (see later)

By the way ... note that intervals as discussed here are ***scalar values !!!***

/ just like type POINT (see earlier) */*

INTERVALS AREN'T NECESSARILY TEMPORAL :

- Tax brackets are represented by taxable income ranges (intervals whose contained points are money values)

INTERVALS AREN'T NECESSARILY TEMPORAL :

- Tax brackets are represented by taxable income ranges (intervals whose contained points are money values)
- Machines operate within certain temperature and voltage ranges (intervals whose contained points are temperatures and voltages, respectively)

INTERVALS AREN'T NECESSARILY TEMPORAL :

- Tax brackets are represented by taxable income ranges (intervals whose contained points are money values)
- Machines operate within certain temperature and voltage ranges (intervals whose contained points are temperatures and voltages, respectively)
- Animals vary in the range of frequencies of light and sound waves to which their eyes and ears are receptive

INTERVALS AREN'T NECESSARILY TEMPORAL :

- Tax brackets are represented by taxable income ranges (intervals whose contained points are money values)
- Machines operate within certain temperature and voltage ranges (intervals whose contained points are temperatures and voltages, respectively)
- Animals vary in the range of frequencies of light and sound waves to which their eyes and ears are receptive
- Various natural phenomena occur in ranges in depth of soil or sea or height above sea level

Etc., etc.

POINT AND INTERVAL TYPES :

Assume type DATE is built in and represents calendar dates (i.e., points on timeline accurate to one day)

Consider interval $[d04:d10]$, whose contained points are values of type DATE ... *Granularity* = one day

Exact type of this interval = **INTERVAL_DATE** ...

- INTERVAL is a *type generator*
- DATE is the *point type* for interval type INTERVAL_DATE

INTERVAL is a ***type generator***

- With associated generic *operators* (see later) and generic constraints

DATE is the ***point type*** for the interval type INTERVAL_DATE

- Determines specific set of interval values that make up this particular interval type
- Namely, the set of all possible intervals of the form $[di:dj]$, where di and dj are DATE values and $di \leq dj$
/ and “ \leq ” is calendar ordering */*

FURTHER EXAMPLES :

INTERVAL_INTEGER

- Values are intervals of the form $[i:j]$, where i and j are INTEGER values and $i \leq j$
- Granularity = one (unity)

INTERVAL_MONEY

- Values are intervals of the form $[dci:dcj]$, where dci and dcj are dollar and cent values and $dci \leq dcj$
- Granularity = one cent

POINT TYPES :

Type T can be used as a point type if all of the following are defined for it:

- A **total ordering** (“ \leq ” etc. available for any pair of values of type T)
- Niladic **FIRST** and **LAST** operators
- Monadic **NEXT** and **PRIOR** operators (can fail)
 - $\text{NEXT} = \text{successor function}$
 - Successor function *assumed unique* (but see later!)

I.e., if T is an *ordinal type* (see later) ... Thus, e.g., DATE, INTEGER, and MONEY are all valid point types

ASIDE :

To get a little more formal regarding the successor function:

- Every value p of type T has exactly one successor $\text{NEXT_}T(p)$, which is also a value of type T
- If $p1 \neq p2$ then $\text{NEXT_}T(p1) \neq \text{NEXT_}T(p2)$
- Exactly one value of type T is not equal to $\text{NEXT_}T(p)$ for any p

Informal notation: Successor of $d = d + 1$
 Predecessor of $d = d - 1$

Successor function is what enables us to determine, in sequence, what points are contained in any given interval

E.g., if $i = [d04:d10]$, contained points are exactly
 $d04, d04+1, d04+2, \dots, d10$
(in that order)

Successor function for type DATE = “next day”

INTERVALS :

Let T be a point type. Then an **interval** (value) i of type $\text{INTERVAL_}T$ is a scalar value for which two monadic ops, BEGIN and END , and one dyadic op, \in , are defined, such that:

- $\text{BEGIN}(i)$ and $\text{END}(i)$ each return a value of type T
- $\text{BEGIN}(i) \leq \text{END}(i)$
- If p is a value of type T , then $p \in i$ is true iff $\text{BEGIN}(i) \leq p$ and $p \leq \text{END}(i)$ are both true

Intervals always nonempty

BEGIN & END would be THE_BEGIN & THE_END in **Tutorial D**
(begin and end points constitute a possrep)

MORE SEARCHING EXAMPLE :

S_PARTS_DURING

SNO	PARTS	DURING
S1	[P1 : P3]	[d01 : d04]
S1	[P2 : P4]	[d07 : d08]
S1	[P5 : P6]	[d09 : d09]
S2	[P1 : P1]	[d08 : d09]
S2	[P1 : P2]	[d08 : d08]
S2	[P3 : P4]	[d07 : d08]
S3	[P2 : P4]	[d01 : d04]
S3	[P3 : P5]	[d01 : d04]
S3	[P2 : P4]	[d05 : d06]
S3	[P2 : P4]	[d06 : d09]
S4	[P3 : P4]	[d05 : d08]

/ Not meant to correspond
in any particular way to
sample SP_DURING value */*

Note problems: e.g., “S3
was able to supply P4 on
days 1-4” appears twice

**We’ll revisit this example
later**

PART II cont. :

Laying the foundations:

4. Time and the database
5. What's the problem?
6. Intervals
7. Interval operators
8. EXPAND and COLLAPSE
9. PACK and UNPACK I
10. PACK and UNPACK II
11. Generalizing the relational operators

INFORMAL NOTATION :

Point type T ; typical value p ; use $p+1$, $p+2$, $p-1$, $p-2$, etc., as shorthands with obvious meanings ... though a real language would provide $NEXT_T$ and $PRIOR_T$ ops, also $FIRST_T$ and $LAST_T$

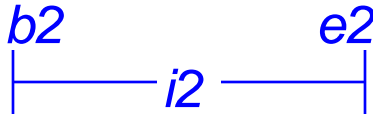
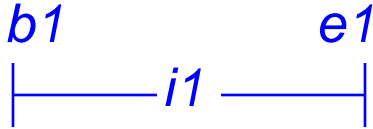
Interval type $INTERVAL_T$ —use $[p1:pn]$ to denote typical interval selector invocation ... though a real language would use more explicit syntax—e.g., $INTERVAL_T([p1:pn])$

Let i be the interval $[b:e]$. Then:

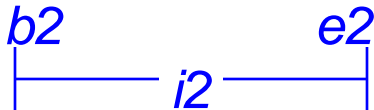
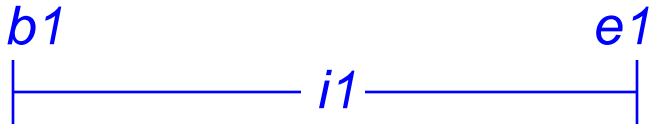
- $BEGIN(i)$ and $END(i)$ return b and e , resp.
- $p \in i \equiv b \leq p \text{ AND } p \leq e$
- $PRE(i)$ returns $b-1$... $POST(i)$ returns $e+1$ /* can fail */
- $i \ni p \equiv p \in i$
- $POINT\ FROM\ i$ returns p iff $i = [p:p]$ /* unit interval */

ALLEN'S OPERATORS :

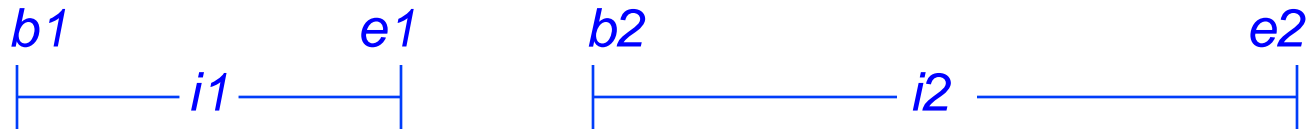
$$i1 = i2$$



$$i1 \supseteq i2 \text{ and } i2 \subseteq i1 \text{ (also } i1 \supset i2 \text{ and } i2 \subset i1)$$



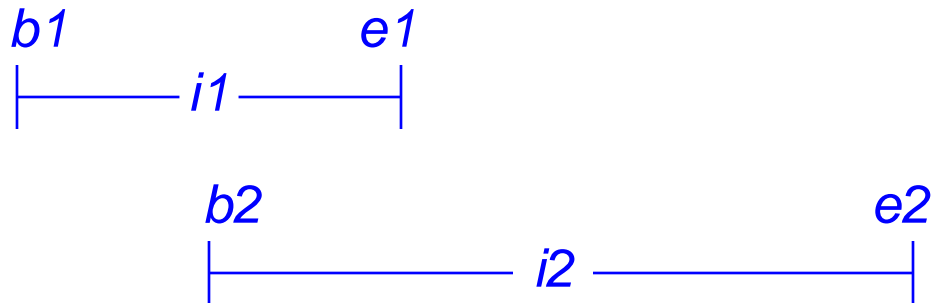
$i1$ BEFORE $i2$ and $i2$ AFTER $i1$



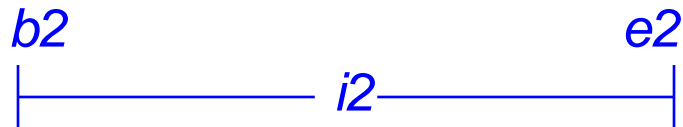
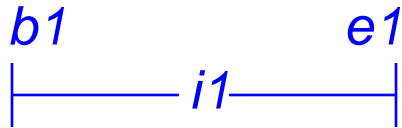
$i1$ MEETS $i2$ and $i2$ MEETS $i1$



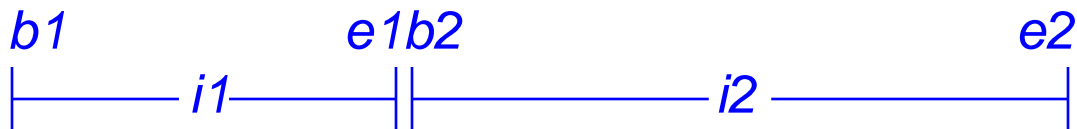
$i1$ OVERLAPS $i2$ and $i2$ OVERLAPS $i1$



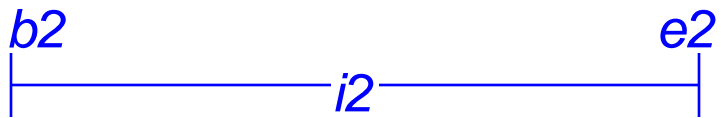
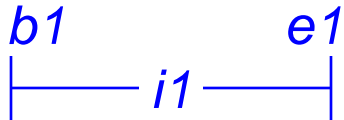
$i1$ MERGES $i2 \equiv i1$ OVERLAPS $i2$ OR $i1$ MEETS $i2$



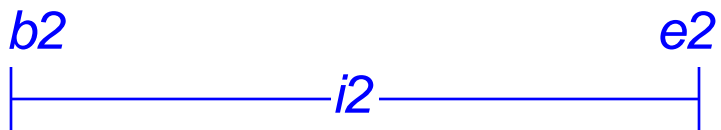
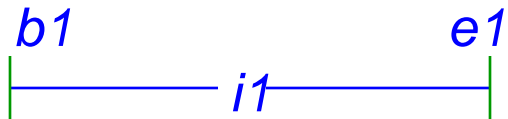
Or:



i1 BEGINS *i2*



i1 ENDS *i2*



MAXIMAL INTERVALS :

Recall sample query:

Query A: Get SNO-FROM-TO triples such that FROM and TO together designate a **maximal interval** during which supplier SNO was able to supply at least one part

What's a “maximal interval”?

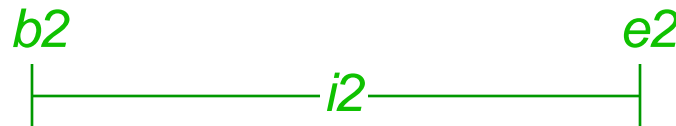
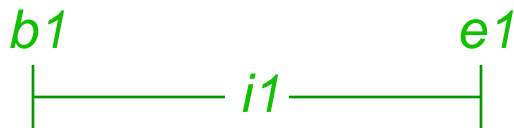
Let P be a predicate whose sole parameter is of some interval type. Then interval i is maximal with respect to P if and only if i satisfies P and no interval j such that $j \supset i$ satisfies P .

OTHER OPERATORS :

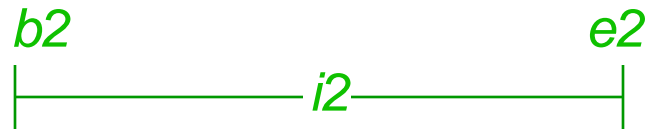
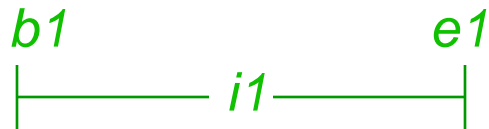
$\text{COUNT}(i)$ /* aka $\text{DURATION}(i)$, aka $\text{LENGTH}(i)$ */ returns no. of points in i (i.e., cardinality)

$\text{MAX}\{p1, p2\}$ and $\text{MIN}\{p1, p2\}$
—with obvious definitions

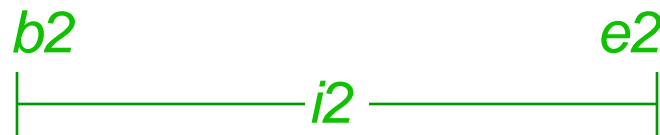
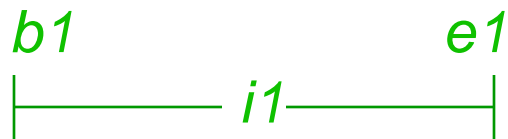
$i1 \text{ UNION } i2$ returns $[\text{MIN}\{b1, b2\} : \text{MAX}\{e1, e2\}]$ if $i1 \text{ MERGES } i2$ is true and is otherwise undefined /* result is an interval */



$i1$ INTERSECT $i2$ returns $[\text{MAX}\{b1, b2\} : \text{MIN}\{e1, e2\}]$ if
 $i1$ OVERLAPS $i2$ is true and is otherwise undefined
/ result is an interval */*



$i1$ MINUS $i2$ returns $[b1:\text{MIN}\{b2-1, e1\}]$ if $b1 < b2$ and $e1 \leq e2$,
 $[\text{MAX}\{e2+1, b1\}:e1]$ if $b1 \geq b2$ and $e1 > e2$,
 and is otherwise undefined (i.e., defined iff (a) $i1$ and $i2$ disjoint
 or (b) $i1 \ni b2$ or $i1 \ni e2$ but not both or (c) exactly one of $i2$ ENDS
 $i1$ or $i2$ BEGINS $i1$ is true */* result is an interval */*



$i1$ MINUS $i2$

SAMPLE QUERIES :

Get supplier numbers for suppliers who were able to supply part P2 on day 8

```
( SP_DURING WHERE PNO = PNO ( 'P2' )  
      AND d08 ∈ DURING ) { SNO }
```

SAMPLE QUERIES :

Get supplier numbers for suppliers who were able to supply part P2 on day 8

```
( SP_DURING WHERE PNO = PNO ( 'P2' )  
      AND d08 ∈ DURING ) { SNO }
```

Get SNO pairs for suppliers who were able to supply the same part at the same time

```
WITH ( t1 := SP_DURING RENAME { SNO AS XNO , DURING AS XD } ,  
      t2 := SP_DURING RENAME { SNO AS YNO , DURING AS YD } ,  
      t3 := t1 JOIN t2 ,  
      t4 := t3 WHERE XD OVERLAPS YD ,  
      t5 := t4 WHERE XNO < YNO ) :  
t5 { XNO, YNO }
```

Get SNO pairs for suppliers who were able to supply the same part at the same time, together with the parts and times in question

```
WITH ( t1 := SP_DURING RENAME { SNO AS XNO , DURING AS XD } ,  
      t2 := SP_DURING RENAME { SNO AS YNO , DURING AS YD } ,  
      t3 := t1 JOIN t2 ,  
      t4 := t3 WHERE XD OVERLAPS YD ,  
      t5 := t4 WHERE XNO < YNO ,  
      t6 := EXTEND t5 : { DURING := XD INTERSECT YD } ) :  
t6 { XNO, YNO, PNO, DURING }
```

PART II cont. :

Laying the foundations:

4. Time and the database
5. What's the problem?
6. Intervals
7. Interval operators
8. EXPAND and COLLAPSE
9. PACK and UNPACK I
10. PACK and UNPACK II
11. Generalizing the relational operators

BREAK :

Next = EXPAND and COLLAPSE

EXPAND AND COLLAPSE :

Work on **sets** of intervals, not individual intervals
(or pairs of intervals) per se

Each takes a set of intervals all of the same type as its
single operand and returns another such set as its result

$$\{ [\dots], [\dots], \dots, [\dots] \} \\ \implies \\ \{ [\dots], [\dots], \dots, [\dots] \}$$

Result in each case is a particular ***canonical form*** for
the original set

ASIDE :

Given (a) a set S of objects and

(b) a notion of equivalence among such objects,

subset C of S is said to be a ***set of canonical forms*** for S (under the stated definition of equivalence) iff every object s in S is equivalent to just one object c in C

- Object c is the ***canonical form*** for object s
- All “interesting” properties that apply to s also apply to c ; thus, we can study just the small set C , not the large set S , in order to obtain or prove a variety of “interesting” results

EXPANDED FORM :

The objects we wish to study are **sets of intervals**, where the intervals are all of the same type

Let $X1$ and $X2$ be two such sets. Define *equivalence*:

- $X1$ and $X2$ are **equivalent** iff
set of all points in intervals in $X1$ =
set of all points in intervals in $X2$

E.g.:

$$X1 = \{ [d01:d01], [d03:d05], [d04:d06] \}$$

$$X2 = \{ [d01:d01], [d03:d04], [d05:d05], [d05:d06] \}$$

EXPANDED FORM (cont.) :

Corresp set of points = $\{ d01, d03, d04, d05, d06 \}$

But we're more interested in corresp set of **unit intervals**:

$$X3 = \{ [d01:d01], [d03:d03], [d04:d04], \\ [d05:d05], [d06:d06] \}$$

$X3$ is equivalent to both $X1$ and $X2$ (it's the *expanded form* of both)

If X is a set of intervals all of the same type, then the **expanded form** of X is the set of all intervals of the form $[p:p]$ —i.e., unit intervals—such that p is a point in some interval in X

EXPANDED FORM (cont.) :

Given any such set X , a corresponding expanded form always exists; expanded form is equivalent to X and is unique

Expanded form of X is one possible canonical form for X

- Unique set equivalent to X such that every contained interval is of minimum possible duration (*viz.*, one)

$X1 \equiv X2$ iff they have the same expanded form

Intuitively, the expanded form of X allows us to focus on the information content of X at an atomic level, without worrying about the many different ways that information might be bundled together into clumps

COLLAPSED FORM :

$$X1 = \{ [d01:d01], [d03:d05], [d04:d06] \}$$

$$X2 = \{ [d01:d01], [d03:d04], [d05:d05], [d05:d06] \}$$

$$X3 = \{ [d01:d01], [d03:d03], [d04:d04], \\ [d05:d05], [d06:d06] \}$$

Expanded form here (X3) has greatest cardinality: **fluke!**

$$X4 = \{ [d01:d01], [d03:d03], [d03:d04], [d03:d05], \\ [d03:d06], [d04:d04], [d04:d05], [d04:d06] \}$$

X4 has same expanded form, but greater cardinality than X3

COLLAPSED FORM (cont.) :

$$X5 = \{ [d01:d01], [d03:d06] \}$$

$X5$ has same expanded form but minimum possible cardinality
(it's the *collapsed form* of $X1, X2, X3, X4$)

If X is a set of intervals all of the same type, then the **collapsed form** of X is the set Y of intervals of the same type such that:

- a. X and Y have the same expanded form, and
- b. No two distinct intervals $i1$ and $i2$ in Y are such that $i1 \text{ MERGES } i2$ is true

COLLAPSED FORM (cont.) :

Given any such set X , a corresponding collapsed form always exists; collapsed form is equivalent to X and is unique

Collapsed form of X is another possible canonical form for X

- Unique set equivalent to X that has the minimum possible cardinality

$X1 \equiv X2$ iff they have the same collapsed form

Intuitively, the collapsed form of X allows us to focus on the information content of X in a compressed (clumped) form, without worrying about the possibility that clumps might overlap or abut

LET X BE A SET OF INTERVALS ALL OF THE SAME TYPE :

- $\text{EXPAND} (X)$: returns expanded form of X
- $\text{COLLAPSE} (X)$: returns collapsed form of X

(What happens if X has cardinality one? Or zero?)

Ops are *not* inverses of each other! In fact:

- $\text{EXPAND} (\text{COLLAPSE} (X)) \equiv \text{EXPAND} (X)$
- $\text{COLLAPSE} (\text{EXPAND} (X)) \equiv \text{COLLAPSE} (X)$

NOW I NEED TO CLEAN UP MY ACT !

Relational model doesn't support general sets, it supports *relations*!

However, a set of values $v1, v2, \dots, vn$ all of the same type can easily be mapped to a unary relation:

RELATION { TUPLE { A $v1$ } , TUPLE { A $v2$ } ,
....., TUPLE { A vn } }

Returns:

A
$v1$
$v2$
..
vn

↑
relation
selector
invocation

So let's replace EXPAND and COLLAPSE as previously described by versions in which the argument is specified as a *unary relation* ...

r
DURING
[d06:d09]
[d04:d08]
[d05:d10]
[d01:d01]

EXPAND (r)
DURING
[d01:d01]
[d04:d04]
[d05:d05]
[d06:d06]
[d07:d07]
[d08:d08]
[d09:d09]
[d10:d10]

COLLAPSE (r)
DURING
[d01:d01]
[d04:d10]

... and extend the definition of equivalence accordingly

NULLARY RELATIONS :

Nullary relation has no attributes at all ...

Recall: There are exactly two such relations!

- TABLE_DEE has just one tuple (the “0-tuple”)
- TABLE_DUM has no tuples at all

Highly desirable to define versions of EXPAND and COLLAPSE for nullary relations /* Why? See later */

Definition: Result = input in every case

PART II cont. :

Laying the foundations:

4. Time and the database
5. What's the problem?
6. Intervals
7. Interval operators
8. EXPAND and COLLAPSE
9. PACK and UNPACK I : The single-attribute case
10. PACK and UNPACK II : The multiattribute case
11. Generalizing the relational operators

BREAK :

Next = PACK and UNPACK (I)

PRELIMINARY EXAMPLE :

r

SNO	DURING
S2	[d02:d04]
S2	[d03:d05]
S4	[d02:d05]
S4	[d04:d06]
S4	[d09:d10]

UNPACK *r*
ON (DURING)

SNO	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]

PACK *r*
ON (DURING)

SNO	DURING
S2	[d02:d05]
S4	[d02:d06]
S4	[d09:d10]

PACKING RELATIONS :

Recall sample query:

Query A: Get SNO-FROM-TO triples such that FROM and TO together designate a maximal interval during which supplier SNO was able to supply at least one part

Revised version:

Query A: Get SNO-DURING pairs such that DURING designates a maximal interval during which supplier SNO was able to supply at least one part

Result can be obtained from SP_DURING alone ...

SP_DURING /* *sample value* */ :

SNO	PNO	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	[d02:d04]
S2	P1	[d08:d10]
S2	P2	[d03:d03]
S2	P2	[d09:d10]
S3	P2	[d08:d10]
S4	P2	[d06:d09]
S4	P4	[d04:d08]
S4	P5	[d05:d10]

DESIRED RESULT :

SNO	DURING
S1	[d04:d10]
S2	[d02:d04]
S2	[d08:d10]
S3	[d08:d10]
S4	[d04:d10]

Note: Given DURING value for given supplier in this result does *not* necessarily exist as an explicit DURING value for that supplier in SP_DURING: see, e.g., supplier S4

Original version of Query A involved projection on SP

Revised version of Query A is going to involve “temporal projection” on SP_DURING */* slightly deprecated terminology */*

I’ll build up the formulation of the query one small step at a time ...

Project away part numbers:

```
WITH ( t1 := SP_DURING { SNO , DURING } ) :  
/* part numbers irrelevant */
```

t1 LOOKS LIKE THIS :

SNO	DURING
S1	[d04:d10]
S1	[d05:d10]
S1	[d09:d10]
S1	[d06:d10]
S2	[d02:d04]
S2	[d08:d10]
S2	[d03:d03]
S2	[d09:d10]
S3	[d08:d10]
S4	[d06:d09]
S4	[d04:d08]
S4	[d05:d10]

Note the *redundancy*—
E.g., "Supplier S1 was able to
supply something on day 6"
appears three times!

NEXT :

<i>t2</i>	SNO	X
	S1	<div>DURING</div> <div>[d04:d10]</div> <div>[d05:d10]</div> <div>[d09:d10]</div> <div>[d06:d10]</div>
	S2	<div>DURING</div> <div>[d02:d04]</div> <div>[d08:d10]</div> <div>[d03:d03]</div> <div>[d09:d10]</div>

WITH (*t2* :=
t1 GROUP { DURING } AS X) :
/ X is relation valued */*

S3	<div>DURING</div> <div>[d08:d10]</div>
S4	<div>DURING</div> <div>[d06:d09]</div> <div>[d04:d08]</div> <div>[d05:d10]</div>

WITH ($t3 := \text{EXTEND } t2 : \{ X := \text{COLLAPSE} (X) \}) :$

$t3$

SNO	X
S1	DURING
	[$d04 : d10$]
S2	DURING
	[$d02 : d04$] [$d08 : d10$]

S3	DURING
	[$d08 : d10$]
S4	DURING
	[$d04 : d10$]

t3 UNGROUP X

SNO	DURING
S1	[<i>d04</i> : <i>d10</i>]
S2	[<i>d02</i> : <i>d04</i>]
S2	[<i>d08</i> : <i>d10</i>]
S3	[<i>d08</i> : <i>d10</i>]
S4	[<i>d04</i> : <i>d10</i>]

**Packed form
of *t1* on DURING**

WITH (*t1* := SP_DURING { SNO , DURING } ,
 t2 := *t1* GROUP { DURING } AS X ,
 t3 := EXTEND *t2* : { X := COLLAPSE (X) }) :
t3 UNGROUP X

THE PACK OPERATOR (shorthand!) :

$$\text{PACK } r \text{ ON } (A) \equiv \text{WITH } (r1 := r \text{ GROUP } \{A\} \text{ AS } X, \\ r2 := \text{EXTEND } r1 : \{X := \text{COLLAPSE}(X)\}) : \\ r2 \text{ UNGROUP } X$$

Note: Packing r on A involves **partitioning** r on all of its attributes *apart from* A /* though one partition might yield N result tuples */

Query A:

$$\text{PACK SP_DURING } \{ \text{SNO} , \text{DURING} \} \text{ ON } (\text{DURING})$$

/ “temporal projection”—see later */*

UNPACKING RELATIONS :

Recall sample query:

Query B: Get SNO-FROM-TO triples such that FROM and TO together designate a maximal interval during which supplier SNO was unable to supply any parts at all

Revised version:

Query B: Get SNO-DURING pairs such that DURING designates a maximal interval during which supplier SNO was unable to supply any parts at all

Need to inspect both S_DURING and SP_DURING ...

Original version of Query B involved relational difference

Revised version is going to involve “temporal difference”
(involves unpack *and* pack) */* slightly deprecated terminology */*

Intuitively, we want SNO-DURING pairs that appear in or are implied by S_DURING and do *not* appear in and are *not* implied by SP_DURING

Suggests (correctly!)

1. two unpacks
2. take difference
3. pack result

THE UNPACK OPERATOR (shorthand!) :

$$\text{UNPACK } r \text{ ON } (A) \equiv \text{WITH } (r1 := r \text{ GROUP } \{A\} \text{ AS } X, \\ r2 := \text{EXTEND } r1 : \{X := \text{EXPAND}(X)\}) : \\ r2 \text{ UNGROUP } X$$

Left operand (u1) for difference op:

SNO-DURING pairs that appear in or are implied by S_DURING:

UNPACK S_DURING { SNO , DURING } ON (DURING)

/ shorthand for: */*

```

WITH ( r1 := S_DURING { SNO , DURING } ,
      r2 := r1 GROUP { DURING } AS X ,
      r3 := EXTEND r2 : { X := EXPAND ( X ) } ) :
r3 UNGROUP X

```

LEFT OPERAND (*u1*) :

SNO	DURING
S1	[<i>d04</i> : <i>d04</i>]
S1	[<i>d05</i> : <i>d05</i>]
S1	[<i>d06</i> : <i>d06</i>]
S1	[<i>d07</i> : <i>d07</i>]
S1	[<i>d08</i> : <i>d08</i>]
S1	[<i>d09</i> : <i>d09</i>]
S1	[<i>d10</i> : <i>d10</i>]
S2	[<i>d02</i> : <i>d02</i>]
S2	[<i>d03</i> : <i>d03</i>]
S2	[<i>d04</i> : <i>d04</i>]
S2	[<i>d07</i> : <i>d07</i>]
S2	[<i>d08</i> : <i>d08</i>]
S2	[<i>d09</i> : <i>d09</i>]
S2	[<i>d10</i> : <i>d10</i>]
S3	[<i>d03</i> : <i>d03</i>]
S3	[<i>d04</i> : <i>d04</i>]
S3	[<i>d05</i> : <i>d05</i>]
S3	[<i>d06</i> : <i>d06</i>]
S3	[<i>d07</i> : <i>d07</i>]

S4	[<i>d04</i> : <i>d04</i>]
S4	[<i>d05</i> : <i>d05</i>]
S4	[<i>d06</i> : <i>d06</i>]
S4	[<i>d07</i> : <i>d07</i>]
S4	[<i>d08</i> : <i>d08</i>]
S4	[<i>d09</i> : <i>d09</i>]
S4	[<i>d10</i> : <i>d10</i>]
S5	[<i>d02</i> : <i>d02</i>]
S5	[<i>d03</i> : <i>d03</i>]
S5	[<i>d04</i> : <i>d04</i>]
S5	[<i>d05</i> : <i>d05</i>]
S5	[<i>d06</i> : <i>d06</i>]
S5	[<i>d07</i> : <i>d07</i>]
S5	[<i>d08</i> : <i>d08</i>]
S5	[<i>d09</i> : <i>d09</i>]
S5	[<i>d10</i> : <i>d10</i>]

RIGHT OPERAND (*u2*) :

SNO-DURING pairs that appear in or are implied by SP_DURING:

UNPACK SP_DURING { SNO , DURING } ON (DURING)

SNO	DURING
S1	[d04:d04]
S1	[d05:d05]
S1	[d06:d06]
S1	[d07:d07]
S1	[d08:d08]
S1	[d09:d09]
S1	[d10:d10]
S2	[d02:d02]
S2	[d03:d03]
S2	[d04:d04]
S2	[d08:d08]
S2	[d09:d09]
S2	[d10:d10]

S3	[d08:d08]
S3	[d09:d09]
S3	[d10:d10]
S4	[d04:d04]
S4	[d05:d05]
S4	[d06:d06]
S4	[d07:d07]
S4	[d08:d08]
S4	[d09:d09]
S4	[d10:d10]

DIFFERENCE (*u3*) :

u1 MINUS *u2*

SNO	DURING
S2	[<i>d07</i> : <i>d07</i>]
S3	[<i>d03</i> : <i>d03</i>]
S3	[<i>d04</i> : <i>d04</i>]
S3	[<i>d05</i> : <i>d05</i>]
S3	[<i>d06</i> : <i>d06</i>]
S3	[<i>d07</i> : <i>d07</i>]
S5	[<i>d02</i> : <i>d02</i>]
S5	[<i>d03</i> : <i>d03</i>]

S5	[<i>d04</i> : <i>d04</i>]
S5	[<i>d05</i> : <i>d05</i>]
S5	[<i>d06</i> : <i>d06</i>]
S5	[<i>d07</i> : <i>d07</i>]
S5	[<i>d08</i> : <i>d08</i>]
S5	[<i>d09</i> : <i>d09</i>]
S5	[<i>d10</i> : <i>d10</i>]

FINALLY :

PACK *u3* ON (DURING)

SNO	DURING
S2	[<i>d07</i> : <i>d07</i>]
S3	[<i>d03</i> : <i>d07</i>]
S5	[<i>d02</i> : <i>d10</i>]

Query B:

PACK

((UNPACK S_DURING { SNO , DURING } ON (DURING))

MINUS

(UNPACK SP_DURING { SNO , DURING } ON (DURING)))

ON (DURING)

/ “temporal difference”—see later */*

PLEASE NOTE :

PACK and UNPACK are *not* inverses of each other! In fact:

$$\text{PACK} (\text{UNPACK } r \text{ ON } (A)) \text{ ON } (A) \equiv \text{PACK } r \text{ ON } (A)$$

$$\text{UNPACK} (\text{PACK } r \text{ ON } (A)) \text{ ON } (A) \equiv \text{UNPACK } r \text{ ON } (A)$$

Hence, can ignore first operation in a PACK/UNPACK or (more important) UNPACK/PACK sequence

Also:

$$\text{PACK} (\text{PACK } r \text{ ON } (A)) \text{ ON } (A) \equiv \text{PACK } r \text{ ON } (A)$$

$$\text{UNPACK} (\text{UNPACK } r \text{ ON } (A)) \text{ ON } (A) \equiv \text{UNPACK } r \text{ ON } (A)$$

MORE EXAMPLES :

NHW { NAME , HEIGHT , WEIGHT }

/ not temporal! */*

NHW	NAME	HEIGHT	WEIGHT
	<i>n1</i>	70	150
	<i>n2</i>	71	150
	<i>n3</i>	70	150
	<i>n4</i>	60	150
	<i>n5</i>	61	150
	<i>n6</i>	66	180
	<i>n7</i>	67	180
	<i>n8</i>	62	150

For each weight in NHW, get every range of heights such that for each height in that range there's at least one person in NHW of that height and weight

DESIRED RESULT :

For each weight in NHW, get every range of heights such that for each height in that range there's at least one person in NHW of that height and weight

WEIGHT	HR
150	[60 : 62]
150	[70 : 71]
180	[66 : 67]

For each weight in NHW, get every range of heights such that for each height in that range there's at least one person in NHW of that height and weight:

PACK

```
(( EXTEND NHW { HEIGHT , WEIGHT } :  
  { HR := INTERVAL_HEIGHT ( [ HEIGHT : HEIGHT ] } )  
                                     { WEIGHT, HR } )
```

ON (HR)

Predicate: HR denotes a maximal interval of heights such that, for all heights h in HR, there exists at least one person p such that p has height h and weight WEIGHT

Back to suppliers and shipments: At any given time, if there are any shipments at all at that time, then there's some part number $pmax$ such that no supplier is able to supply any part at that time with a part number greater than $pmax$

For each part number that has ever been such a $pmax$ value, get that part number together with the interval(s) during which it actually was that $pmax$ value

DESIRED RESULT :

PMAX	DURING
P2	[<i>d08:d08</i>]
P3	[<i>d09:d10</i>]
P5	[<i>d04:d05</i>]
P6	[<i>d06:d07</i>]

DESIRED RESULT :

PMAX	DURING
P2	[<i>d08</i> : <i>d08</i>]
P3	[<i>d09</i> : <i>d10</i>]
P5	[<i>d04</i> : <i>d05</i>]
P6	[<i>d06</i> : <i>d07</i>]

```
WITH ( t1 := UNPACK SP_DURING ON ( DURING ) ,  
      t2 := EXTEND t1 { DURING } : { PMAX := MAX ( !!t1 , PNO ) } ) :  
PACK t2 ON ( DURING )
```

S_PARTS_DURING { SNO , PARTS , DURING }

SNO	PARTS	DURING
..
S2	[P3:P4]	[d07:d08]
S3	[P2:P4]	[d01:d04]
S3	[P3:P5]	[d01:d04]
S3	[P2:P4]	[d05:d06]
S3	[P2:P4]	[d06:d09]
S4	[P3:P4]	[d05:d08]

For each part that has ever been capable of being supplied by S3, get PNO and applicable time intervals

WITH (*t1* := S_PARTS_DURING WHERE SNO = SNO ('S3') ,
 t2 := *t1* { PARTS , DURING } ,
 t3 := UNPACK *t2* ON (PARTS) ,
 t4 := EXTEND *t3* : { PNO := POINT FROM PARTS } ,
 t5 := *t4* { PNO , DURING }) :
PACK *t5* ON (DURING)

PNO	DURING
P2	[<i>d01</i> : <i>d09</i>]
P3	[<i>d01</i> : <i>d09</i>]
P4	[<i>d01</i> : <i>d09</i>]
P5	[<i>d01</i> : <i>d04</i>]

PART II cont. :

Laying the foundations:

4. Time and the database
5. What's the problem?
6. Intervals
7. Interval operators
8. EXPAND and COLLAPSE
9. PACK and UNPACK I : The single-attribute case
10. PACK and UNPACK II : The multiattribute case
11. Generalizing the relational operators

BREAK :

Next = PACK and UNPACK (II)

PACK / UNPACK ON NO ATTRIBUTES :

Can generalize PACK and UNPACK r to operate in terms of *any subset* of set of interval attributes of r

Consider empty subset first (important!)

Define both PACK r ON () and UNPACK r ON () to return r

Justification: PACK r ON () \equiv

```
WITH (  $r1 := r$  GROUP { } AS  $X$ ,  
       $r2 :=$  EXTEND  $r1 : \{ X := \text{COLLAPSE} ( X ) \} ) :$   
 $r2$  UNGROUP  $X$ 
```

/ as usual, except for GROUP { } instead of GROUP { A } */*

Step 1 effectively:

$r1 = \text{EXTEND } r : \{ X := \text{TABLE_DEE} \}$

Step 2 effectively:

$r2 = r1$ /* collapsing TABLE_DEE gives TABLE_DEE */

Step 3 effectively:

Result = $r2$ without attribute X (in other words, just r)

Similarly for UNPACK r ON ()

UNPACK ON 2+ ATTRIBUTES :

Let r be as follows: \Rightarrow

Consider expression

UNPACK

(UNPACK r ON (A1))

ON (A2)

A1	A2
[P1 : P1]	[d08 : d09]
[P1 : P2]	[d08 : d08]
[P3 : P4]	[d07 : d08]

UNPACK r ON (A1) $\Rightarrow r'$

A1	A2
[P1 : P1]	[d08 : d09]
[P1 : P1]	[d08 : d08]
[P2 : P2]	[d08 : d08]
[P3 : P3]	[d07 : d08]
[P4 : P4]	[d07 : d08]

UNPACK r' ON (A2)

A1	A2
[P1 : P1]	[d08 : d08]
[P1 : P1]	[d09 : d09]
[P2 : P2]	[d08 : d08]
[P3 : P3]	[d07 : d07]
[P3 : P3]	[d08 : d08]
[P4 : P4]	[d07 : d07]
[P4 : P4]	[d08 : d08]

UNPACK r ON (A2) $\implies r'$ UNPACK r' ON (A1)

A1	A2
[P1:P1]	[d08:d08]
[P1:P1]	[d09:d09]
[P1:P2]	[d08:d08]
[P3:P4]	[d07:d07]
[P4:P4]	[d08:d08]

A1	A2
[P1:P1]	[d08:d08]
[P1:P1]	[d09:d09]
[P2:P2]	[d08:d08]
[P3:P3]	[d07:d07]
[P4:P4]	[d07:d07]
[P3:P3]	[d08:d08]
[P4:P4]	[d08:d08]

Final result is same as before!

SO :

UNPACK (UNPACK r ON ($A1$)) ON ($A2$)

≡

UNPACK (UNPACK r ON ($A2$)) ON ($A1$)

Generalization:

UNPACK r ON ($A1$, $A2$, ... , An)

/ unambiguous ... parens could be braces */*

≡

UNPACK (... (UNPACK (UNPACK r ON ($B1$)) ON ($B2$)) ...) ON (Bn)

where $B1$, $B2$, ..., Bn is some arbitrary permutation of $A1$, $A2$, ..., An

PACK ON 2+ ATTRIBUTES :

Let r be as follows:



Define PACK r ON (A1 , A2)

to involve preliminary

UNPACK r ON (A1 , A2) ; *

then ...

A1	A2
[P2 : P4]	[d01 : d04]
[P3 : P5]	[d01 : d04]
[P2 : P4]	[d05 : d06]
[P2 : P4]	[d06 : d09]

PACK r ON (A1 , A2)

A1	A2
[P2 : P5]	[d01 : d04]
[P2 : P4]	[d05 : d09]

PACK r ON (A2 , A1)

A1	A2
[P2 : P4]	[d01 : d09]
[P5 : P5]	[d01 : d04]

* This definition **IS** consistent with earlier definitions for $n = 0$ and $n = 1$

SO :

PACK (PACK r ON ($A1$)) ON ($A2$)

\neq

PACK (PACK r ON ($A2$)) ON ($A1$)

Generalization:

PACK r ON ($A1$, $A2$, ... , An)

/ must be parens, not braces, because sequence matters */*

\equiv

PACK (... (PACK (PACK r' ON ($A1$)) ON ($A2$)) ...) ON (An)

where r' is UNPACK r ON ($A1$, $A2$, ..., An)

GRAPHICAL REPRESENTATION :

Let r be as follows: \Rightarrow

$x1$
 $x2$
 $x3$
 $x4$

A1	A2
[P2 : P4]	[d01 : d04]
[P3 : P5]	[d01 : d04]
[P2 : P4]	[d05 : d06]
[P2 : P4]	[d06 : d09]

$left = \text{PACK } r \text{ ON } (A1, A2)$

A1	A2
[P2 : P5] [P2 : P4]	[d01 : d04] [d05 : d09]

$right = \text{PACK } r \text{ ON } (A2, A1)$

A1	A2
[P2 : P4] [P5 : P5]	[d01 : d09] [d01 : d04]

Draw a graph of part nos. (vertical) vs. days (horizontal) ... for original relation r and for each of $left$ and $right$

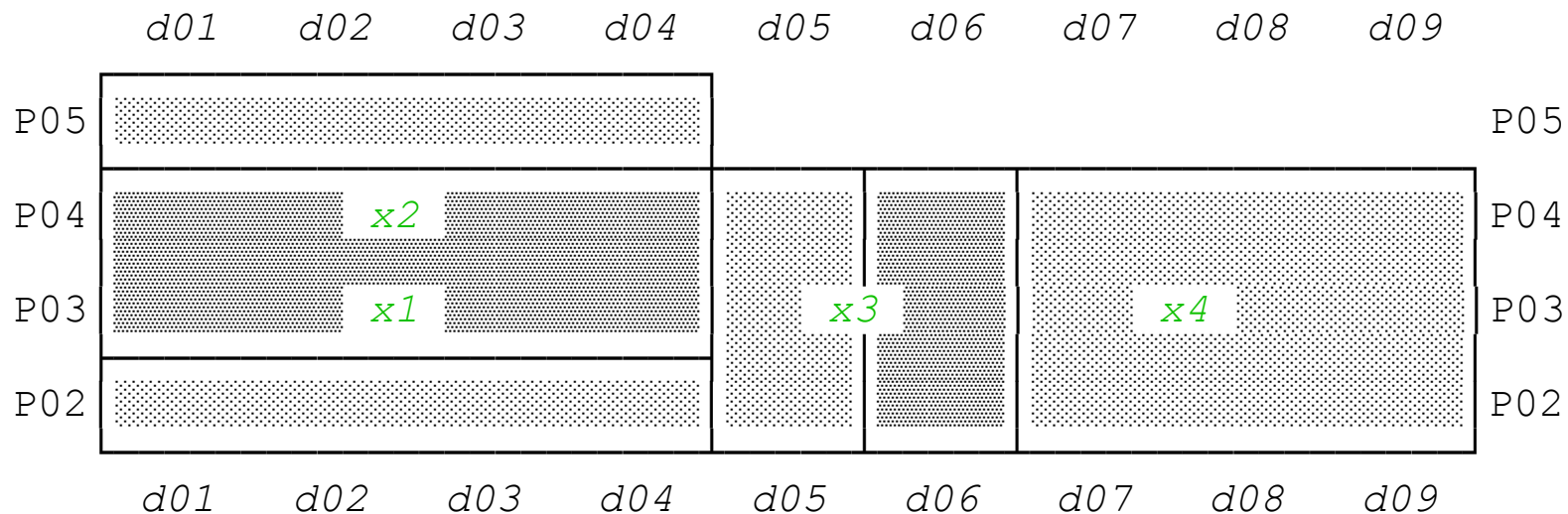
GRAPHICAL REPRESENTATION :

x1 [P2:P4] [d01:d04]

x2 [P3:P5] [d01:d04]

x3 [P2:P4] [d05:d06]

x4 [P2:P4] [d06:d09]

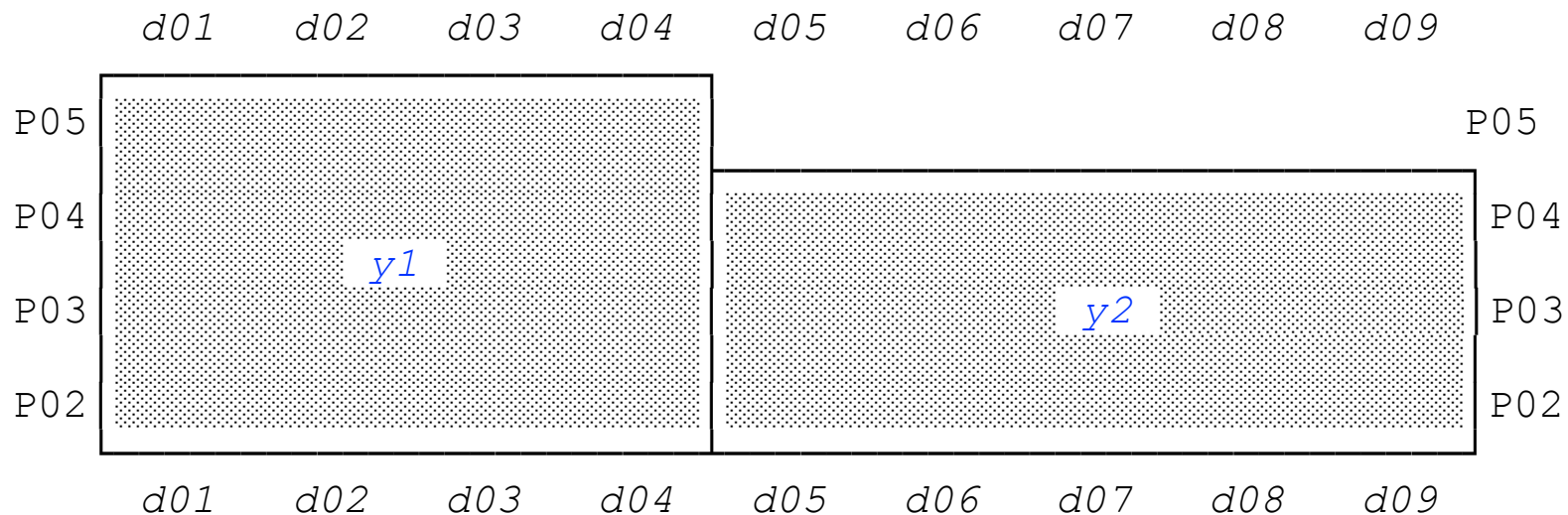


GRAPHICAL REPRESENTATION :

left

y1 [P2:P5] [d01:d04]

y2 [P2:P4] [d05:d09]

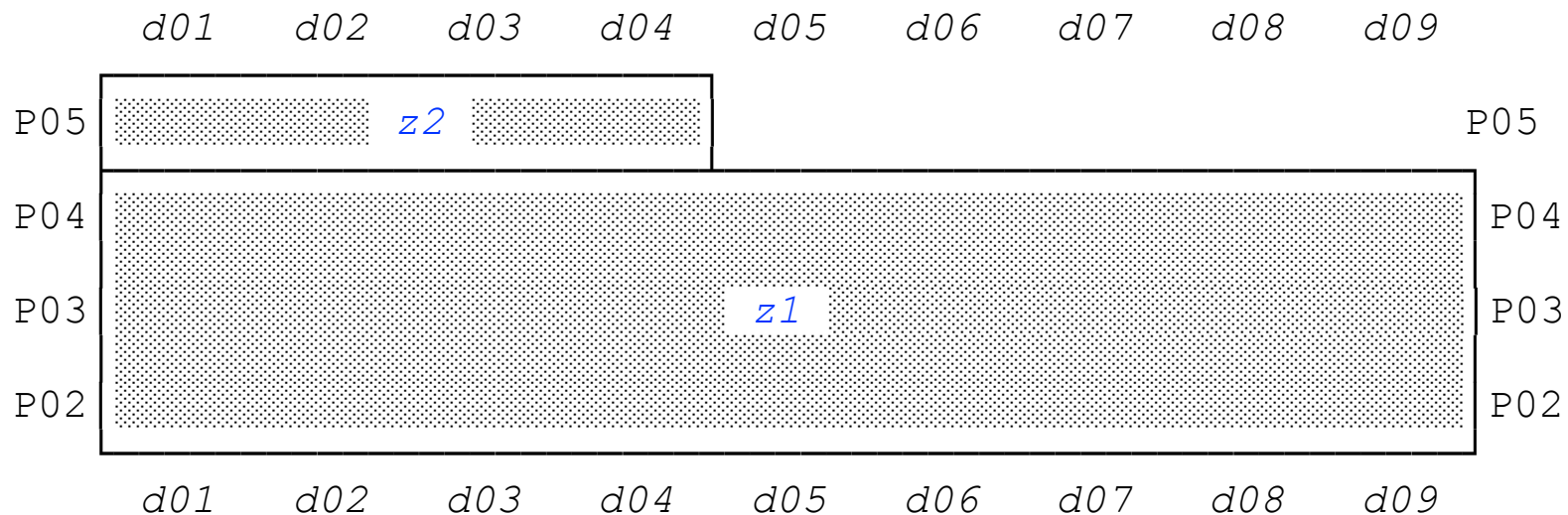


GRAPHICAL REPRESENTATION :

right

z1 [P2:P4] [d01:d09]

z2 [P5:P5] [d01:d04]



WHY UNPACK FIRST ???

Again let r be as follows: \Rightarrow

A1	A2
[P2:P4]	[d01:d04]
[P3:P5]	[d01:d04]
[P2:P4]	[d05:d06]
[P2:P4]	[d06:d09]

$left = \text{PACK } r \text{ ON } (A1, A2)$

A1	A2
[P2:P5] [P2:P4]	[d01:d04] [d05:d09]

$\text{PACK} (\text{PACK } r \text{ ON } (A1))$
 $\text{ON } (A2)$ gives same result *left*
/ fluke! */*

$right = \text{PACK } r \text{ ON } (A2, A1)$

A1	A2
[P2:P4] [P5:P5]	[d01:d09] [d01:d04]

$\text{PACK} (\text{PACK } r \text{ ON } (A2))$
 $\text{ON } (A1)$ does **not** give same
 result *right* (or *left*) ... gives
 [P3:P5] **not** [P5:P5]

IN OTHER WORDS :

PACK r ON ($A2$, $A1$)

➡ **no redundancy**

PACK (PACK r ON ($A2$)) ON ($A1$)

➡ **redundancy**

That's why we define PACK to do an UNPACK first !!!

More generally:

PACK r ON ($A1$, ..., A_n) /* $n > 1$ */ \neq
PACK (... PACK (PACK r ON ($A1$)) ...) ON (A_n)

EQUIVALENCE OF RELATIONS :

PACK r ON ($A1$, $A2$)

A1	A2
[P2 : P5] [P2 : P4]	[d01 : d04] [d05 : d09]

PACK r ON ($A2$, $A1$)

A1	A2
[P2 : P4] [P5 : P5]	[d01 : d09] [d01 : d04]

Convey same information! (obviously, since they have the same unpacked form) ... I.e., **equivalent** (with respect to $A1$ and $A2$)

- $r1$ and $r2$ are **equivalent** (with respect to $A1, ..., An$) iff
UNPACK $r1$ ON ($A1, ..., An$) = UNPACK $r2$ ON ($A1, ..., An$)

But different “points of view”—*left* = ranges of parts for given time intervals, *right* = time intervals for given ranges of parts

EQUIVALENCE (cont.) :

To convert *left* into *right*:

PACK *left* ON (A2 , A1)

Next ...

left has at most one tuple for any given day, *right* has at most one tuple for any given part—*fluke*!

■ Counterexample ➡

<i>r</i>	A1	A2
	[P3:P8]	[d01:d04]
	[P5:P9]	[d03:d08]
	[P1:P7]	[d07:d10]

PACK *r* ON (A1 , A2)

A1	A2
[P3:P8]	[d01:d02]
[P3:P9]	[d03:d04]
[P5:P9]	[d05:d06]
[P1:P9]	[d07:d08]
[P1:P7]	[d09:d10]

Two distinct tuples
for part P3
(ditto part P4)



PACK *r* ON (A2 , A1)

A1	A2
[P3:P4]	[d01:d04]
[P1:P4]	[d07:d10]
[P5:P7]	[d01:d10]
[P8:P8]	[d01:d08]
[P9:P9]	[d03:d08]

REDUNDANCY REVISITED :

- In foregoing example, two packed forms are distinct again
- Both redundancy free /* see below */
- *But not as “compact” as original relation!*
- Let r have interval attributes A_1, \dots, A_n . Let u be result of UNPACK r ON (A_1, \dots, A_n) . Then r is **redundancy free** iff every tuple in u derives from exactly one tuple in r .
- Any “fully packed” form of any relation is guaranteed to be redundancy free

REDUNDANCY (cont.) :

In foregoing example, two packed forms are of same cardinality—*fluke!*

<i>r</i>	A1	A2
	[P1 : P2]	[d01 : d02]
	[P1 : P4]	[d03 : d04]
	[P3 : P4]	[d05 : d06]

PACK *r* ON (A1 , A2)

A1	A2
[P1 : P2]	[d01 : d02]
[P1 : P4]	[d03 : d04]
[P3 : P4]	[d05 : d06]

PACK *r* ON (A2 , A1)

A1	A2
[P1 : P2]	[d01 : d04]
[P3 : P4]	[d03 : d06]

REDUNDANCY (cont.) :

Given such examples, we see that:

Minimum cardinality, fully packed, form of *r* ***not unique*** (in general)—i.e., there's no canonical form that's “more canonical” than all the rest

Exercise: Give all “fully packed” forms of

A1	A2	A3
[S1 : S2] [S2 : S3]	[P2 : P3] [P3 : P4]	[d03 : d04] [d04 : d05]

ONE LAST POINT :

Let relation r have just one attribute A , of some interval type

Then $\text{PACK } r \text{ ON } (A)$ reduces to $\text{COLLAPSE } (r)$ and
 $\text{UNPACK } r \text{ ON } (A)$ reduces to $\text{EXPAND } (r)$

In other words, */* the relational versions of */* EXPAND and COLLAPSE are really just special cases of UNPACK and PACK , respectively

(They were introduced initially purely as a pedagogical device)

PART II cont. :

Laying the foundations:

4. Time and the database
5. What's the problem?
6. Intervals
7. Interval operators
8. EXPAND and COLLAPSE
9. PACK and UNPACK I : The single-attribute case
10. PACK and UNPACK II : The multiattribute case
11. Generalizing the relational operators

BREAK :

Next = generalized operators

RECALL QUERY B :

Get SNO-DURING pairs such that DURING designates a maximal interval during which supplier SNO was unable to supply any parts at all

PACK

```
( ( UNPACK S_DURING { SNO , DURING } ON ( DURING ) )  
  MINUS  
  ( UNPACK SP_DURING { SNO , DURING } ON ( DURING ) ) )  
ON ( DURING )
```

This kind of expression is needed very frequently ...

So let's define a shorthand!

“U_MINUS” :

USING (ACL) : r1 MINUS r2

/ not checked for */*

/ syntactic soundness */*

r1 and *r2* must be of same type

ACL is a commalist of attribute names in which every attribute mentioned is of some interval type and appears in both relations

Shorthand for:

```
PACK
  ( ( UNPACK r1 ON ( ACL ) )
    MINUS
    ( UNPACK r2 ON ( ACL ) ) )
ON ( ACL )
```

/ “temporal difference”—slightly deprecated terminology */*

QUERY B *bis* :

```
USING ( DURING ) : S_DURING { SNO , DURING }  
MINUS  
SP_DURING { SNO , DURING }
```

QUERY B *bis* :

USING (DURING) : S_DURING { SNO , DURING }
MINUS
SP_DURING { SNO , DURING }

Result can have cardinality > that of left operand! E.g.:

r1

A
[d02:d04]

r2

A
[d03:d03]

USING (A) :
r1 MINUS *r2*

A
[d02:d02] [d04:d04]

“U_UNION” :

USING (ACL) : *r1* UNION *r2* ... Shorthand for:

PACK
((UNPACK *r1* ON (ACL))
UNION
(UNPACK *r2* ON (ACL)))
ON (ACL)

Result can have cardinality < that of either operand! E.g.:

r1

A
[<i>d02</i> : <i>d04</i>]
[<i>d04</i> : <i>d04</i>]

r2

A
[<i>d03</i> : <i>d03</i>]
[<i>d04</i> : <i>d04</i>]

USING (A) :
r1 UNION *r2*

A
[<i>d02</i> : <i>d04</i>]

A NOTE ON OPTIMIZATION :

USING (A) : *r1* UNION *r2*

Shorthand can be simplified to:

PACK (*r1* UNION *r2*) ON (A)

(i.e., UNPACKs unnec if *ACL* involves just one attribute)

Analogous remarks apply to certain other “U_” ops

/ see the textbook for details */*

“U_INTERSECT” :

USING (ACL) : *r1* INTERSECT *r2* ... Shorthand for:

PACK
((UNPACK *r1* ON (ACL))
INTERSECT
(UNPACK *r2* ON (ACL)))
ON (ACL)

Result can have cardinality > that of either operand! E.g.:

r1

A
[<i>d01</i> : <i>d01</i>]
[<i>d03</i> : <i>d07</i>]

r2

A
[<i>d01</i> : <i>d04</i>]
[<i>d06</i> : <i>d09</i>]

USING (A) :
r1 INTERSECT *r2*

A
[<i>d01</i> : <i>d01</i>]
[<i>d03</i> : <i>d04</i>]
[<i>d06</i> : <i>d07</i>]

“U_JOIN” :

USING (ACL) : r1 JOIN r2 ... Shorthand for:

```
PACK
  ( ( UNPACK r1 ON ( ACL ) )
    JOIN
      ( UNPACK r2 ON ( ACL ) ) )
ON ( ACL )
```

Every attribute in *ACL* must be common to *r1* and *r2* (thus, join is on at least all attributes in *ACL*)

/ U_INTERSECT is a special case of U_JOIN */*

For example:

S_CITY_DURING { SNO , CITY , DURING }
KEY { SNO , DURING }

Predicate: *DURING denotes a maximal interval of days throughout which supplier SNO was located in city CITY*

Get SNO-CITY-PNO-DURING tuples such that SNO was both (a) located in CITY, and (b) able to supply PNO, throughout interval DURING

USING (DURING) : S_CITY_DURING JOIN SP_DURING

“U_MATCHING” :

USING (ACL) : *r1* MATCHING *r2* ... Shorthand for:

```
PACK
  ( ( UNPACK r1 ON ( ACL ) )
    MATCHING
    ( UNPACK r2 ON ( ACL ) ) )
ON ( ACL )
```

Regular MATCHING is a regular projection of a regular join ...
U_MATCHING is a U_projection (q.v.) of a U_join

Expansion:

```
USING ( ACL ) : ( ( USING ( ACL ) : r1 JOIN r2 ) { A1 , ... , An } )
```

/ A1,...,An = all of the attributes of r1 */*

“NOT U_MATCHING” :

USING (ACL) : r1 NOT MATCHING r2 ... Shorthand for:

```
PACK
  ( ( UNPACK r1 ON ( ACL ) )
    NOT MATCHING
    ( UNPACK r2 ON ( ACL ) ) )
ON ( ACL )
```

Expansion:

```
USING ( ACL ) : ( r1 MINUS ( USING ( ACL ) : r1 MATCHING r2 ) )
```

/ a U_difference in which the second operand is a U_semijoin */*

n-ADIC OPERATORS :

USING (ACL) : JOIN { *r1* , *r2* , ... , *rn* } ... Shorthand for:

```
PACK
  ( JOIN { UNPACK r1 ON ( ACL ) ,
           UNPACK r2 ON ( ACL ) ,
           ..... ,
           UNPACK rn ON ( ACL ) } )
ON ( ACL )
```

Similarly for UNION and INTERSECT (and D_UNION)

“U_RENAME” :

USING (ACL) : r RENAME { A AS B } ... Shorthand for:

```
PACK  
  ( ( UNPACK r ON ( ACL ) ) RENAME { A AS B } )  
ON ( ACL )
```

Can simplify to:

```
PACK ( r RENAME { A AS B } ) ON ( ACL )
```

/ included mainly for completeness */*

“U_restrict” :

USING (ACL) : r WHERE bx ... Shorthand for:

PACK ((UNPACK r ON (ACL)) WHERE bx) ON (ACL)

E.g.:

USING (DURING) : S_DURING WHERE DURING =
INTERVAL_DATE ([d04:d04])

Note: UNPACK applies to *r*, **not** to *r WHERE bx*

Result can have cardinality > that of its operand!
(Exercise for the reader)

“U_project” :

USING (ACL) : r { BCL } ... Shorthand for:

PACK ((UNPACK r ON (ACL)) { BCL }) ON (ACL)

Every attribute in *ACL* must be in *BCL* too

Recall *Query A*: Get SNO-DURING pairs such that DURING designates a maximal interval during which supplier SNO was able to supply at least one part

USING (DURING) : SP_DURING { SNO , DURING }

/ “temporal projection”—slightly deprecated terminology */*

“U_EXTEND” :

USING (ACL) : EXTEND $r : \{ A := exp \}$... Shorthand for:

PACK
(EXTEND (UNPACK r ON (ACL)) : { $A := exp$ })
ON (ACL)

Result can have cardinality either $>$ or $<$ that of its operand! E.g.:

SNO	DURING
S2	[$d01 : d04$]
S2	[$d02 : d03$]

USING (DURING) : EXTEND S_DURING :
{ $X := \text{POINT FROM DURING}$ }

“U_EXTEND” :

USING (ACL) : EXTEND $r : \{ A := exp \}$... Shorthand for:

PACK

$$\text{ON} (ACL) \left(\text{EXTEND} (\text{UNPACK } r \text{ ON} (ACL)) : \{ A := exp \} \right)$$

Result can have cardinality either $>$ or $<$ that of its operand! E.g.:

SNO	DURING
S2	[<i>d01</i> : <i>d04</i>]
S2	[<i>d02</i> : <i>d03</i>]

```

USING ( DURING ) : EXTEND S_DURING :
                        { X := POINT FROM DURING }

```

USING (DURING) : EXTEND S_DURING : { X := 1 }

“U_GROUP”, “U_UNGROUP” :

USING (*ACL*) : *r* GROUP { *BCL* } AS *C* \equiv

PACK
((UNPACK *r* ON (*ACL*)) GROUP { *BCL* } AS *C*)
ON (*ACL*)

ACL and *BCL* must be disjoint

USING (*ACL*) : *r* UNGROUP *B* \equiv

PACK
((UNPACK *r* ON (*ACL*)) UNGROUP *B*)
ON (*ACL*)

B is an RVA and thus can't appear in *ACL*

U_ COMPARISONS :

USING (ACL) : *r1 op r2*

Shorthand for:

(UNPACK *r1* ON (ACL)) *op* (UNPACK *r2* ON (ACL))

E.g.:

r1

A
[<i>d01</i> : <i>d03</i>]
[<i>d02</i> : <i>d05</i>]
[<i>d04</i> : <i>d04</i>]

r2

A
[<i>d01</i> : <i>d02</i>]
[<i>d03</i> : <i>d05</i>]

r1 = *r2* is false, but USING (A) : *r1* = *r2* is true

UNDERLYING INTUITION :

An alternative way of thinking about “temporal relations” and the “U_” operators ... Consider following temporal relation *r*:

SNO	DURING
S1	[d04:d06]
S2	[d02:d04]
S2	[d06:d07]
S3	[d05:d07]
S4	[d03:d05]

Predicate (simplified): Supplier SNO was under contract throughout DURING

UNDERLYING INTUITION (cont.) :

Can think of r as specifying state of affairs at each of six points in time $d02, d03, \dots, d07$

I.e., can think of r as shorthand for *sequence* of six separate relations (sequence chronological, of course):

<i>r02</i>	<i>r03</i>	<i>r04</i>	<i>r05</i>	<i>r06</i>	<i>r07</i>
SNO	SNO	SNO	SNO	SNO	SNO
S2	S2 S4	S1 S2 S4	S1 S3 S4	S1 S2 S3	S2 S3

UNDERLYING INTUITION (cont.) :

E.g., relation $r03$ shows suppliers S2 and S4 (only) as under contract on day $d03$

Obtained from r by:

1. Restricting r to tuples where $d03 \in \text{DURING}$
2. Projecting result on all attributes but DURING

Let REL_TO_SEQ be the operator that produces the sequence of relations $r02, r03, \dots, r07$ from r

Let SEQ_TO_REL be the inverse operator that will take the sequence of relations $r02, r03, \dots, r07$ and give r again

UNDERLYING INTUITION (cont.) :

Consider (e.g.) U_JOIN

Basic idea:

1. For each individual point in time, apply regular JOIN to pair of relations that correspond to that time point
2. Put the results of all those individual JOINS back together again

U_JOIN = regular join on a time point by time point basis!

Similarly for all of the other U_ ops

REGULAR RELATIONAL OPERATORS :

Consider U_MINUS once again: USING (*ACL*) : *r1* MINUS *r2* ...
Shorthand for:

```
PACK  
  ( ( UNPACK r1 ON ( ACL ) )  
    MINUS  
    ( UNPACK r2 ON ( ACL ) ) )  
ON ( ACL )
```

Suppose *ACL* is empty: USING () : *r1* MINUS *r2* ... Shorthand for:

```
PACK  
  ( ( UNPACK r1 ON ( ) )  
    MINUS  
    ( UNPACK r2 ON ( ) ) )  
ON ( )
```

BUT ... !!!

UNPACK *r* ON () = PACK *r* ON () = *r*

Expansion reduces to just *r1* MINUS *r2* !!!

Regular MINUS is just a special case of U_MINUS !!!

So define MINUS syntax:

[USING (*ACL*) :] *r1* MINUS *r2*

/ allow USING specification (including colon separator) to */*
/ be omitted iff ACL is empty */*

No longer any need to talk about “U_MINUS” as such at all —all MINUS invocations effectively become U_MINUS invocations, and we can generalize the meaning of MINUS accordingly

Similarly for all other relational operators: In all cases, regular operator is just that special case of the corresponding “U_” operator in which the USING spec mentions no attributes at all (and hence can be omitted)

In other words, don’t need to talk about “U_” operators, as such, at all (except perhaps for emphasis)

Regular operators permit (but don’t require) an additional operand when they’re applied to relations with interval attributes

PART II review :

Laying the foundations:

4. Time and the database
5. What's the problem?
6. Intervals
7. Interval operators
8. EXPAND and COLLAPSE
9. PACK and UNPACK I : The single-attribute case
10. PACK and UNPACK II : The multiattribute case
11. Generalizing the relational operators

AGENDA :

I. A review of relational concepts

II. Laying the foundations

III. Building on the foundations

IV. SQL support

V. Appendixes

BREAK :

Next = database design: structure

PART III :

Building on the foundations:

- 12. Database design I : Structure
- 13. Database design II : Keys and related constraints
- 14. Database design III : General constraints
- 15. Queries
- 16. Updates
- 17. Logged time and stated time
- 18. Point and interval types revisited

HOW DO TEMPORAL DB DESIGN ???

Relvar S_DURING is much too simplistic to illustrate temporal DB design problems ...

So let's go back (at least initially) to *nontemporal* relvar S:

VAR S RELATION

```
{ SNO SNO , SNAME NAME , STATUS INTEGER , CITY CHAR }  
KEY { SNO } ;
```

The supplier identified by SNO is under exactly one contract, has exactly one name SNAME, has exactly one status STATUS, and is located in exactly one city CITY

JUST ADD A “TEMPORAL” ATTRIBUTE ???

Semitemporal

```
VAR SSSC_SINCE RELATION
    { SNO      SNO ,
      SNAME    NAME ,
      STATUS   INTEGER ,
      CITY     CHAR ,
      SINCE    DATE }
KEY { SNO } ;
```

Fully temporal

```
VAR SSSC_DURING RELATION
    { SNO      SNO ,
      SNAME    NAME ,
      STATUS   INTEGER ,
      CITY     CHAR ,
      DURING   INTERVAL_DATE }
KEY { SNO , DURING } ;
```

JUST ADD A “TEMPORAL” ATTRIBUTE ???

Semitemporal

```
VAR SSSC_SINCE RELATION
    { SNO      SNO ,
      SNAME    NAME ,
      STATUS   INTEGER ,
      CITY     CHAR ,
      SINCE    DATE }
KEY { SNO } ;
```

Don't do this!

Fully temporal

```
VAR SSSC_DURING RELATION
    { SNO      SNO ,
      SNAME    NAME ,
      STATUS   INTEGER ,
      CITY     CHAR ,
      DURING   INTERVAL_DATE }
KEY { SNO , DURING } ;
```

Don't do this!

In fact, both of these relvars are very BADLY designed

If a semitemporal design is all we want, we need to fix the “since” /* or, very loosely, “current” */ relvar SSSC_SINCE ... But that’s fairly easy to do

If a fully temporal design is what we want, we need to fix the “during” /* or, very loosely, “historical” */ relvar SSSC_DURING ...
We propose:

- **Vertical** decomposition, to deal with the fact that distinct “properties” of the same “entity” vary at different rates
- **Horizontal** decomposition, to deal with the *logical difference* between current and historical information

But first things first ...

SINCE RELVARS :

Merely “semitemporal”—contain “current info only” (?)

Well ... such a relvar does contain *implicit* info re both past and future ... and *explicit* info re the past ... and possibly *explicit* info re the future too! ... as we'll see

Predicate for SSSC_SINCE:

Ever since day SINCE (and not before day SINCE), all four of the following have been true:

- a. Supplier SNO has been under contract*
- b. Supplier SNO has been named SNAME*
- c. Supplier SNO has had status STATUS*
- d. Supplier SNO has been located in city CITY*

Obviously a bad design! E.g.:

SNO	SNAME	STATUS	CITY	SINCE
S1	Smith	20	London	<i>d04</i>

Suppose today = day 10 ... Effective from today, status of supplier S1 is 30 ... Replace tuple shown by:

SNO	SNAME	STATUS	CITY	SINCE
S1	Smith	30	London	<i>d10</i>

Now we've lost info that (e.g.) supplier S1 has been located in London since day 4!

SSSC_SINCE can't represent info for a current supplier that predates time of most recent update to that supplier! ... SINCE "timestamps too much"

Hence redesign */* attrib types omitted for simplicity */* :

```
SSSC_SINCE { SNO ,      SNO_SINCE ,  
              SNAME ,   SNAME_SINCE ,  
              STATUS ,  STATUS_SINCE ,  
              CITY ,    CITY_SINCE }  
KEY { SNO }
```

Supplier SNO has been under contract ever since SNO_SINCE, has been named SNAME ever since SNAME_SINCE, has had status STATUS ever since STATUS_SINCE, and has been located in city CITY ever since CITY_SINCE

RELVAR SSSC_SINCE :

Contains *explicit* info re the **PAST** = “since” values

Contains *implicit* info re the **PAST** = what can be inferred from “since” values (e.g., if CITY = London and today = *d10* and CITY_SINCE = *d04*)

RELVAR SSSC_SINCE :

Contains *explicit* info re the **PAST** = “since” values

Contains *implicit* info re the **PAST** = what can be inferred from “since” values (e.g., if CITY = London and today = *d10* and CITY_SINCE = *d04*)

Might contain *explicit* info re the **FUTURE** = “since” values, if those values refer to the future

/ Important! */* Contains *implicit* info re the **FUTURE** = interval from “since” value to *end of time* ... E.g., “supplier S1 under contract since day 4” = “supplier S1 under contract from day 4 *until the last day*”

- “Ever since” is *open ended*

DURING RELVARS :

“Fully temporal”—contain “historical info only” (?)

Well ... such a relvar might contain *explicit* info re both present and future (obviously enough)

Predicate for SSSC_DURING:

DURING denotes a maximal interval throughout which all four of the following were true:

- a. Supplier SNO was under contract*
- b. Supplier SNO was named SNAME*
- c. Supplier SNO had status STATUS*
- d. Supplier SNO was located in city CITY*

Obviously a bad design! E.g.:

SNO	SNAME	STATUS	CITY	DURING
S2	Jones	10	Paris	[d02:d04]

Surely much more likely that the intervals during which (a) supplier S2 was under contract, (b) supplier S2 was named Jones, (c) supplier S2 had status 10, and (d) supplier S2 was located in Paris, would all be different!

DURING “timestamps too much”

So let's replace the original relvar by four separate relvars, each with its own timestamp ...

VERTICAL DECOMPOSITION :

S_DURING	{ SNO , DURING }	<i>/* who was under */</i>
	KEY { SNO , DURING }	<i>/* contract when; */</i>
		<i>/* necessary ??? */</i>
S_NAME_DURING	{ SNO , SNAME , DURING }	<i>/* who had which */</i>
	KEY { SNO , DURING }	<i>/* name when */</i>
S_STATUS_DURING	{ SNO , STATUS , DURING }	<i>/* who had which */</i>
	KEY { SNO , DURING }	<i>/* status when */</i>
S_CITY_DURING	{ SNO , CITY , DURING }	<i>/* who was where */</i>
	KEY { SNO , DURING }	<i>/* when */</i>

SAMPLE TUPLES :

SNO	DURING
S2	[d01:d09]

SNO	SNAME	DURING
S2	Jones	[d01:d04]

SNO	STATUS	DURING
S2	10	[d01:d06]

SNO	CITY	DURING
S2	Paris	[d01:d03]

SNO	SNAME	DURING
S2	Johns	[d05:d09]

SNO	STATUS	DURING
S2	15	[d07:d09]

SNO	CITY	DURING
S2	Athens	[d04:d09]

Foregoing vertical decomposition is very reminiscent of classical normalization ... Let's take a closer look:

Classical normalization is based on *projection* and *join* ...
PJ/NF = 5NF is ultimate normal form with respect to these operators as classically defined /* but see ETNF! */

Of course, decomposition must be **nonloss**

Decompose “as far as possible” ???—i.e., to **irreducible components** ???

E.g., **nontemporal** S { SNO , SNAME , STATUS , CITY } *could* be so decomposed ... but little point in doing so (?)

Argument for decomposing nontemporal relvar S isn't very strong ... but argument for decomposing temporal analog SSSC_DURING is much stronger:

Name, status, city vary *at different rates*

Name history, status history, city history each much more *digestible concepts* than combined name / status / city history concept

With original SSSC_DURING:

```
/* city history (interesting) */  
USING ( DURING ) : SSSC_DURING { SNO , CITY , DURING }
```

```
/* name / status / city history (boring) */  
SSSC_DURING
```

With vertical decomposition:

```
/* city history (interesting) */  
S_CITY_DURING
```

```
/* name / status / city history (boring) */  
USING ( DURING ) : S_NAME_DURING JOIN  
                    ( USING ( DURING ) : S_STATUS_DURING  
                      JOIN  
                        S_CITY_DURING )
```

Or:

```
USING ( DURING ) : JOIN { S_NAME_DURING ,  
                          S_STATUS_DURING ,  
                          S_CITY_DURING }
```

So SSSC_DURING = this U_join ... Decomposition *is* nonloss!

JOIN DEPENDENCIES (JDs) :

Let H be a heading; then a **JD with respect to H** is an expression of the form $\star \{X_1, \dots, X_n\}$, where the union of X_1, \dots, X_n (“components” of the JD) is equal to H . E.g.:

1. $\star \{ \{ \text{SNO}, \text{SNAME} \}, \{ \text{SNO}, \text{STATUS} \}, \{ \text{SNO}, \text{CITY} \} \}$
2. $\star \{ \{ \text{SNO}, \text{SNAME}, \text{STATUS} \}, \{ \text{SNAME}, \text{CITY} \} \}$
3. $\star \{ \{ \text{SNO}, \text{SNAME}, \text{CITY} \}, \{ \text{CITY}, \text{STATUS} \} \}$

Let relation r have heading H and let $\star \{X_1, \dots, X_n\}$ be a JD, J say, with respect to H . If r is equal to the join of its projections on X_1, X_2, \dots, X_n , then r **satisfies J** ; otherwise r **violates J** .

E.g., the current value of nontemporal relvar S satisfies 1. and 2. above but violates 3.

JDs AND 5NF :

Let relvar R have heading H and let J be a JD with respect to H . Then J **holds** in R if and only if every relation r that can validly be assigned to R satisfies J .

E.g., JD 1. holds in nontemporal relvar S , but JDs 2. and 3. don't

Relvar R can be nonloss decomposed into its projections on X_1, \dots, X_n if and only if $\star \{X_1, \dots, X_n\}$ holds in R

Relvar R is in **fifth normal form** (5NF, aka PJ/NF) if and only if every JD that holds in R is implied by the keys of R

... where J is **implied by the keys** of R if and only if every relation r that satisfies R 's key constraints also satisfies J

“JD IMPLIED BY KEYS” ???

It's easy to prove JD J is implied by the keys of R if and only if the following **membership algorithm** produces a set X containing the heading H of R as a member:

1. $X := \{ X1, \dots, Xn \}$ */* initialize X to components of J */*
2. If two distinct members of X both include the same key K of R , replace them in X by their union
3. Repeat Step 2 until no more replacements are possible

E.g., consider nontemporal relvar S and JD 1. from two pages back:

EXAMPLE :

⊗ { { SNO , SNAME } , { SNO , STATUS } , { SNO , CITY } }

1. $X := \{ \{ \text{SNO} , \text{SNAME} \} , \{ \text{SNO} , \text{STATUS} \} , \{ \text{SNO} , \text{CITY} \} \}$

2. $X := \{ \{ \text{SNO} , \text{SNAME} \} , \{ \text{SNO} , \text{STATUS} , \text{CITY} \} \}$

3. $X := \{ \{ \text{SNO} , \text{SNAME} , \text{STATUS} , \text{CITY} \} \}$

Success! JD 1. *is* implied by the key(s) of S

U_JOIN DEPENDENCIES :

Let H be a heading and let ACL be a subset of H in which every attribute is interval valued; then a **U_JD with respect to ACL and H** is an expression of the form **USING (ACL) :**
⊛ $\{X_1, \dots, X_n\}$, where the union of X_1, \dots, X_n
("components" of the U_JD) is equal to H .

Let relation r have heading H and let **USING (ACL) :**
⊛ $\{X_1, \dots, X_n\}$ be a U_JD, UJ say, with respect to ACL and H . If r is U_equal to the U_join of its U_projections on X_1, X_2, \dots, X_n , then r **satisfies UJ** ; otherwise r **violates UJ** .

Note: In this definition, the U_equality comparison, U_join, and U_projections must all be "with respect to ACL "

U_JDs (cont.) :

Let relvar R have heading H and let UJ be a U_JD with respect to H . Then UJ **holds** in R if and only if every relation r that can validly be assigned to R satisfies UJ .

E.g., following U_JD holds in relvar SSSC_DURING:

USING (DURING) :

⊛ { S_DURING , S_NAME_DURING ,
S_STATUS_DURING , S_CITY_DURING }

/ where “S_DURING” (etc.) refer to corresponding U_projections */*

Hence SSSC_DURING can be nonloss decomposed into its U_projections S_DURING etc. / though in fact the S_DURING projection could be dropped without loss */*

U_JDs AND 6NF :

All JDs are U_JDs! ... Generalize meaning of “JD” accordingly

Relvar R is in **sixth normal form (6NF)** if and only if no JD holds in R at all except for trivial ones

... where J is **trivial** if and only if one of its components is equal to the pertinent heading H in its entirety—i.e., it's of the form $[\text{USING} (ACL) :] \star \{ \dots, H, \dots \}$

Every 6NF relvar is in 5NF ... but is *irreducible*

SSSC_DURING isn't in 6NF!—though it *is* in 5NF

Recommended discipline: If R has interval attributes and isn't in 6NF, decompose it into 6NF projections

BREAK :

Next = “the moving point NOW”

“THE MOVING POINT NOW” :

Have been assuming that history starts at “the beginning of time” and continues up to the present time (OK)

Have also been assuming that present time is recorded as the specific date *d10*—**not OK at all!** (Note *ambiguity* in particular: day 10 as such or “until further notice”?)

- At midnight on day 10, every “until further notice” appearance of *d10* must be replaced by *d11*, instantaneously—?!?
- Think of finer granularity intervals (e.g., msec) !!!

So ... use special marker (“NOW”) ???

Replace (e.g.) [*d04:d10*] by [*d04:NOW*] ???

SOME QUESTIONS :

- If $i = [d04:NOW]$, what is $END(i)$ on day 10?
 - Query: “When does supplier S1’s contract terminate?”
 - Response $d10$ is wrong
 - Response NOW is wrong (maybe less so)
 - What’s the data type of NOW?
- If $i = [NOW:d10]$, what happens to i at midnight on day 10?
(The “creeping DELETE” problem!)
- What’s the result of the comparison $d10 = NOW$?
 - Not *unknown*, please ... PLEASE !!!

-
- What are NOW+1 and NOW-1?
 - Do $i1 = [d01:NOW]$ and $i2 = [d06:d07]$ meet or overlap?
 - What's $\text{EXPAND}(\{[d04:NOW]\})$?
 - What's the cardinality of $\{[d01:NOW], [d01: d04]\}$?
 - Etc., etc., etc.

Note that NOW is really a *variable* ... The idea of VALUES containing VARIABLES is a **logical absurdity!**

ASIDE :

If we have during relvars only, we must put SOMETHING in the DB to represent *until further notice* when *until further notice* is what we mean ...

E.g., supplier whose contract has not yet terminated (and actual termination date not known)

Can use “the last day” ... but you’re lying!

Bad idea to put info in the DB that you *know* is wrong

Violates fundamental principle that tuples in the DB are supposed to correspond to true propositions (CWA!)

Better solution: Choose combination design!

BOTH SINCE AND DURING RELVARS :

Since (“current”) relvars only: Can’t keep proper historical records

During (“historical”) relvars only: Can’t deal with “the moving point *now*” properly

So why not combine, to get the best of both?

Important logical difference:

- *History:* Begin and end times both known
- *Current:* Begin time known, end time not known

(Slight oversimplifications, but good enough for the moment)

Different ***predicates!***

HORIZONTAL DECOMPOSITION :

S_SINCE { SNO , SNO_SINCE , /* exactly like */
 SNAME , SNAME_SINCE , /* SSSC_SINCE, */
 STATUS , STATUS_SINCE , /* except for */
 CITY , CITY_SINCE } /* name change; */
KEY { SNO } /* 5NF, not 6NF */

S_DURING { SNO , DURING } /* no artificial */
KEY { SNO , DURING } /* end times */

S_NAME_DURING { SNO , SNAME , DURING } /* ditto */
KEY { SNO , DURING }

S_STATUS_DURING { SNO , STATUS , DURING } /* ditto */
KEY { SNO , DURING }

S_CITY_DURING { SNO , CITY , DURING } /* ditto */
KEY { SNO , DURING }

OVERSIMPLIFICATIONS :

Might *not* know end time (or even begin time) for historical info—???

- Just a special case of general “missing info” problem
- Not the place for further discussion ... but another good general principle = *Whereof one cannot speak, thereon one must remain silent* ... Document only what you know!

Might know end time for current or future info (e.g., VACATION ex)

- Keep “historical” relvars only?

SUMMARY :

Temporal data IS a little special, in part because of “the moving point *now*”

Best design approach depends on circumstances—combined approach is most general but causes query/update problems
/ to be discussed */*

- Use horizontal decomposition to separate current and historical info */* like operational DB vs. warehouse! */*
- Since relvars (several “since” attributes); normalize to 5NF
- Use vertical decomposition to split during relvars into irreducible (6NF) components, each with its own “during” attribute

FURTHER POINTS :

For during relvars, key *might* not include the “during” attribute
—e.g., suppose suppliers never get a second chance

FURTHER POINTS :

For during relvars, key *might* not include the “during” attribute—e.g., suppose suppliers never get a second chance

Relvar might consist of interval attribs only—e.g., HOLIDAYS (single attribute DATES), giving legal holidays for a certain year

FURTHER POINTS :

For during relvars, key *might* not include the “during” attribute—e.g., suppose suppliers never get a second chance

Relvar might consist of interval attribs only—e.g., HOLIDAYS (single attribute DATES), giving legal holidays for a certain year

Preferred design for shipments:

```
SP_SINCE { SNO , PNO , SINCE }  
          KEY { SNO , PNO }  
          FOREIGN KEY { SNO } REFERENCES S_SINCE /* note! */
```

```
SP_DURING { SNO , PNO , DURING }  
           KEY { SNO , PNO , DURING }
```

PART III cont. :

Building on the foundations:

- 12. Database design I : Structure
- 13. Database design II : Keys and related constraints
- 14. Database design III : General constraints
- 15. Queries
- 16. Updates
- 17. Logged time and stated time
- 18. Point and interval types revisited

BREAK :

Next = keys and related constraints

KEYS AND RELATED CONSTRAINTS :

We already know this has the potential to be a seriously complicated topic ...

Horizontal and vertical decomposition “help” with the problem (sort of), but we have to wade through a lot of complexity first

Ignore attribute pairs **SNAME + SNAME_SINCE**
and **CITY + CITY_SINCE**

Ignore relvars **S_NAME_DURING** and **S_CITY_DURING**

Hence:

DB DESIGN (outline) :

S_SINCE { SNO , SNO_SINCE , STATUS , STATUS_SINCE }
KEY { SNO }

SP_SINCE { SNO , PNO , SINCE }
KEY { SNO , PNO }
FOREIGN KEY { SNO } REFERENCES S_SINCE

S_DURING { SNO , DURING }
KEY { SNO , DURING }

S_STATUS_DURING { SNO , STATUS , DURING }
KEY { SNO , DURING }

SP_DURING { SNO , PNO , DURING }
KEY { SNO , PNO , DURING }

PREDICATES (simplified) :

- *S_SINCE*: Supplier SNO has been under contract since SNO_SINCE and has had status STATUS since STATUS_SINCE
- *SP_SINCE*: Supplier SNO has been able to supply part PNO since SINCE
- *S_DURING*: DURING denotes a maximal interval throughout which supplier SNO was under contract
- *S_STATUS_DURING*: DURING denotes a maximal interval throughout which supplier SNO had status STATUS
- *SP_DURING*: DURING denotes a maximal interval throughout which supplier SNO was able to supply part PNO

SINCE RELVARS : SAMPLE VALUES

S_SINCE

SNO	SNO_SINCE	STATUS	STATUS_SINCE
S1	d04	20	d06
S2	d07	10	d07
S3	d03	30	d03
S4	d04	20	d08
S5	d02	30	d02

SP_SINCE

SNO	PNO	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	P5	d05

DURING RELVARS : SAMPLE VALUES

S_DURING

SNO	DURING
S2	[d02:d04]
S6	[d03:d05]

S_STATUS_DURING

SNO	STATUS	DURING
S1	15	[d04:d05]
S2	5	[d02:d02]
S2	10	[d03:d04]
S4	10	[d04:d04]
S4	25	[d05:d07]
S6	5	[d03:d04]
S6	7	[d05:d05]

SP_DURING

SNO	PNO	DURING
S2	P1	[d02:d04]
S2	P2	[d03:d03]
S3	P5	[d05:d07]
S4	P2	[d06:d09]
S4	P4	[d04:d08]
S6	P3	[d03:d03]
S6	P3	[d05:d05]

Sample values deliberately do NOT correspond exactly to our usual values (though they're close)

They also repay very careful study!

Again ***time is special*** ... Temporal data, by its very nature, has to satisfy certain “denseness” constraints (i.e., certain conditions that must be satisfied *at every point within* certain intervals)

- E.g., if S1 has been under contract ever since day 4 (and not on day 3), then S1 must have had some status ever since day 4 (and not on day 3)

FOREIGN KEYS :

In definition of relvar SP_SINCE:

```
FOREIGN KEY { SNO } REFERENCES S_SINCE
```

```
/* any supplier currently able to supply some part must */  
/* currently be under contract */
```

Correct but inadequate! Really need:

Whenever a given supplier is, was, or will be able to supply some part, that supplier is, was, or will be under contract at that time

Stay tuned ...

KEYS :

Keys are as shown in outline definition

For since relvars, there's little more to say

But for during relvars, there's quite a lot ...

- Consider S_STATUS_DURING specifically:

```
VAR S_STATUS_DURING RELATION
  { SNO SNO , STATUS INTEGER , DURING INTERVAL_DATE }
  KEY { SNO , DURING } ;      /* Warning—inadequate! */
```

KEY spec here is correct but inadequate: *Why?*

BECAUSE IT FAILS TO PREVENT (e.g.) :

SNO	STATUS	DURING
S4	25	[d05:d06]

SNO	STATUS	DURING
S4	25	[d06:d07]

- **Redundancy:** Status for S4 on day 6 appears twice
- Better to pack these two tuples into one:

SNO	STATUS	DURING
S4	25	[d05:d07]

- *Not packing tuples:* Like duplicates! (and relvar in violation of its own predicate)

IT ALSO FAILS TO PREVENT (e.g.) :

SNO	STATUS	DURING
S4	25	[d05:d05]

SNO	STATUS	DURING
S4	25	[d06:d07]

- **Circumlocution:** Two tuples instead of one
- Again better to pack these two tuples into one:

SNO	STATUS	DURING
S4	25	[d05:d07]

- **Not packing tuples:** Relvar in violation of its own predicate

WHAT WE NEED :

Constraint A: If two distinct S_STATUS_DURING tuples are identical except for their DURING values *i1* and *i2*, then *i1* MERGES *i2* must be false

Recall MERGES \equiv OVERLAPS or MEETS:

- Replace MERGES by OVERLAPS : Avoid *redundancy*
- Replace MERGES by MEETS : Avoid *circumlocution*

WHAT WE NEED :

Constraint A: If two distinct S_STATUS_DURING tuples are identical except for their DURING values *i1* and *i2*, then *i1 MERGES i2* must be false

Recall MERGES \equiv OVERLAPS or MEETS:

- Replace MERGES by OVERLAPS : Avoid *redundancy*
- Replace MERGES by MEETS : Avoid *circumlocution*

To enforce *Constraint A*, keep S_STATUS_DURING
packed on DURING

```
VAR S_STATUS_DURING RELATION
  { SNO SNO , STATUS INTEGER , DURING INTERVAL_DATE }
  PACKED ON ( DURING )                                /* Warning—still */
  KEY { SNO , DURING } ;                                /* inadequate! */
```

Stating and enforcing *Constraint A* (via PACKED ON specification) solves the redundancy and circumlocution problems ...

... albeit by replacing them by another problem!—namely, the problem of ensuring that updates don't violate the PACKED ON constraint

But we provide some new operators to help with this latter problem (see later)

To repeat, stating and enforcing *Constraint A* (via PACKED ON specification) solves the redundancy and circumlocution problems

- Solves the XFT1 problem (see much earlier)

But still fails to prevent (e.g.):

SNO	STATUS	DURING
S4	10	[<i>d04</i> : <i>d06</i>]

SNO	STATUS	DURING
S4	25	[<i>d05</i> : <i>d07</i>]

- **Contradiction:** Relvar in violation of its own predicate

WHAT WE NEED :

Constraint B: If two S_STATUS_DURING tuples have the same SNO value but differ on their STATUS value, then their DURING values $i1$ and $i2$ must be such that $i1$ OVERLAPS $i2$ is false

WHAT WE NEED :

Constraint B: If two S_STATUS_DURING tuples have the same SNO value but differ on their STATUS value, then their DURING values $i1$ and $i2$ must be such that $i1$ OVERLAPS $i2$ is false

To enforce *Constraint B*:

If S_STATUS_DURING were kept **unpacked** on DURING (ignore the fact that this is impossible because of PACKED ON), then:

- All DURING values would be *unit intervals* (corresponding to individual time points)
- Key would still be {SNO,DURING}; {SNO,DURING} → {STATUS} would still hold; contradiction impossible! Hence:

VAR S_STATUS_DURING RELATION

{ SNO SNO , STATUS INTEGER , DURING INTERVAL_DATE }

PACKED ON (DURING)

WHEN UNPACKED ON (DURING) THEN KEY { SNO , DURING }

KEY { SNO , DURING } ;

```
VAR S_STATUS_DURING RELATION
{ SNO SNO , STATUS INTEGER , DURING INTERVAL_DATE }
PACKED ON ( DURING )
WHEN UNPACKED ON ( DURING ) THEN KEY { SNO , DURING }
KEY { SNO , DURING } ;
```

```
/* clumsy, maybe, but accurate !!! */
```

```
/* should be able to simplify definition in practice */
```

```
/* general syntax = PACKED ON (ACL) and WHEN */
```

```
/* UNPACKED ON (ACL) THEN KEY (BCL) ... every */
```

```
/* attribute in ACL must be mentioned in BCL */
```

BREAK :

Next = combining specifications

COMBINING SPECIFICATIONS :

Eight possible combinations of KEY + PACKED ON + WHEN / THEN

Assume until further notice that at least one KEY spec is *always* required ... Eight possibilities reduce to four

We know PACKED ON + WHEN / THEN might *both* be required ...
So what about:

- PACKED ON and no WHEN / THEN
- WHEN / THEN and no PACKED ON
- Neither PACKED ON nor WHEN / THEN

PACKED ON + NO WHEN / THEN :

S_DURING susceptible to *redundancy* and *circumlocution* but not *contradiction* (because it's “all key”)

So PACKED ON applies but WHEN / THEN does not apply (though not wrong):

```
VAR S_DURING RELATION
  { SNO SNO , DURING INTERVAL_DATE }
  PACKED ON ( DURING )
  KEY { SNO , DURING } ;
```

SP_DURING is analogous

So “all key” implies that WHEN / THEN does not apply ...
but converse not true!—see *later*

WHEN / THEN + NO PACKED ON :

TERM	DURING	PRESIDENT
	[1974:1976]	Ford
	[1977:1980]	Carter
	[1981:1984]	Reagan
	[1985:1988]	Reagan
	[1993:1996]	Clinton
	[1997:2000]	Clinton

PACKED ON must *not* be specified! But WHEN ... THEN ... KEY { DURING } is needed to prevent **contradiction** such as:

DURING	PRESIDENT
[1985:1994]	Reagan

DURING	PRESIDENT
[1993:1996]	Clinton

We do want to prevent **redundancy** such as:

DURING	PRESIDENT
[1993:1995]	Clinton

DURING	PRESIDENT
[1994:1996]	Clinton

We do want to prevent **redundancy** such as:

DURING	PRESIDENT
[1993:1995]	Clinton

DURING	PRESIDENT
[1994:1996]	Clinton

But we *don't* want to prevent “**circumlocution**” such as:

DURING	PRESIDENT
[1993:1996]	Clinton

DURING	PRESIDENT
[1997:2000]	Clinton

An example of “false” circumlocution? So perhaps **PACKED ON** needs to be split into two separate constraints ... ???

But example intuitively fails ... “Circumlocution” is intentional! ...
Two Clinton tuples correspond to *two different presidential terms* ... So add TERMNO attribute (and PACKED ON now does apply):

TERM	DURING	PRESIDENT	TERMNO
	[1974:1976]	Ford	1
	[1977:1980]	Carter	1
	[1981:1984]	Reagan	1
	[1985:1988]	Reagan	2
	[1993:1996]	Clinton	1
	[1997:2000]	Clinton	2

Do we *really* want WHEN / THEN without PACKED ON ???

ANSWER : YES !

TERM with TERMNO is still susceptible to (genuine, as opposed to “false”) **circumlocution** ... e.g., might have two tuples for Carter’s first term with intervals [1977:1978] and [1979:1980]

But the same is true of TERM without TERMNO!

With TERMNO, problem is taken care of by a constraint saying there must be just one tuple for each presidential term, not by PACKED ON as such

I.e., by a **key constraint** on {PRESIDENT, TERMNO} !!!

VAR TERM RELATION

```
{ DURING INTERVAL_ ... ,  
    PRESIDENT NAME , TERMNO INTEGER }  
WHEN UNPACKED ON ( DURING ) THEN KEY { DURING }  
KEY { DURING }  
KEY { PRESIDENT , TERMNO } ;
```

Key constraint on {PRESIDENT,TERMNO} prevents both redundancy and (genuine) circumlocution ... WHEN / THEN is needed to prevent contradiction */* specifically, two presidents at the same time */*

PACKED ON (DURING) now not wrong but would have no effect ... because two tuples whose DURING values “merge” can’t have the same PRESIDENT and TERMNO values

NEITHER PACKED ON NOR WHEN / THEN :

INFLATION	DURING	RATE	Not very well designed !!!
	[m01:m03]	18	
	[m04:m06]	20	
	[m07:m09]	20	
	[m07:m07]	25	
	
	[m01:m12]	20	

PACKED ON (DURING) must *not* be specified! (Unpacking on DURING loses info too.) And only sensible WHEN / THEN is:

WHEN UNPACKED ON (DURING) THEN KEY { DURING , RATE }
/ tells us nothing, because {DURING,RATE} = heading */*

Relvar INFLATION is subject to none of the usual redundancy, circumlocution, and contradiction problems

Or rather, it's impossible to tell the difference between:

- a. A value of INFLATION that does suffer from such problems, and
- b. A value of INFLATION that looks as if it suffers from such problems but is in fact correct and doesn't

INFLATION vs. TERM :

INFLATION and TERM (with TERMNO) both seem not to need PACKED ON ... How do these relvars differ?

$\{ \text{DURING} \} \rightarrow \{ \text{PRESIDENT}, \text{TERMNO} \}$ holds in TERM and
 $\{ \text{DURING} \} \rightarrow \{ \text{RATE} \}$ holds in INFLATION

Unpack both on DURING:

$\{ \text{DURING} \} \rightarrow \{ \text{PRESIDENT}, \text{TERMNO} \}$ still holds
 $\{ \text{DURING} \} \rightarrow \{ \text{RATE} \}$ *doesn't!*

RATE is a property of DURING interval *taken as a whole*
—*not* a property of individual time points

Thus, in INFLATION, DURING intervals are "entities" ... So give them IDs? ... I.e., add ID attribute (PACKED ON and WHEN / THEN now *obviously* do not apply)

INFLATION

ID	DURING	RATE
<i>Q1</i>	<i>[m01 : m03]</i>	18
<i>Q2</i>	<i>[m04 : m06]</i>	20
<i>Q3</i>	<i>[m07 : m09]</i>	20
<i>M7</i>	<i>[m07 : m07]</i>	25
..
<i>Y1</i>	<i>[m01 : m12]</i>	20

But we'd still want the result of INFLATION { DURING , RATE } not to be packed, though

KEY CONSTRAINTS REVISITED :

Most relvars with interval attributes *will* be subject to both PACKED ON and WHEN / THEN (also KEY)

Proposed syntax: **USING (ACL) : KEY { K }**
 / every attribute in ACL must be in K */*

Shorthand for:

PACKED ON (ACL)
WHEN UNPACKED ON (ACL) THEN KEY { K }
KEY { K }

{ K } is a “U_key” (but ...)

Now consider:

USING () KEY : { *K* } */* i.e., ACL is empty */*

Shorthand for:

PACKED ON ()
WHEN UNPACKED ON () THEN KEY { *K* }
KEY { *K* }

PACKED ON () has no effect

UNPACKED ON () has no effect

/ WHEN / THEN just means { K } is a key for the pertinent relvar */*

So regular KEY constraints are just a special case!

DB DESIGN (outline) :

S_SINCE { SNO , SNO_SINCE , STATUS , STATUS_SINCE }
KEY { SNO }

SP_SINCE { SNO , PNO , SINCE }
KEY { SNO , PNO }
FOREIGN KEY { SNO } REFERENCES S_SINCE

S_DURING { SNO , DURING }
USING (DURING) : KEY { SNO , DURING }

S_STATUS_DURING { SNO, STATUS, DURING }
USING (DURING) : KEY { SNO , DURING }

SP_DURING { SNO, PNO, DURING }
USING (DURING) : KEY { SNO , PNO , DURING }

PART III cont. :

Building on the foundations:

- 12. Database design I : Structure
- 13. Database design II : Keys and related constraints
- 14. Database design III : General constraints
- 15. Queries
- 16. Updates
- 17. Logged time and stated time
- 18. Point and interval types revisited

BREAK :

Next = general constraints

GENERAL CONSTRAINTS :

Now adopt a different strategy ...

Stand back from specific DB design shown earlier; instead, consider in very general terms *all* of the constraints we might want a temporal DB involving suppliers and shipments to satisfy */* continue to ignore supplier names and cities */*

Then see what those constraints might look like in terms of our “combination” design */* see textbook for other designs */*

Also briefly consider the possibility of introducing certain ***syntactic shorthands***

NINE GENERAL REQUIREMENTS

(three groups of three) :

- **Requirement R1:** If DB shows supplier S_x as being under contract on day d , it must contain exactly one tuple that shows that fact
- **Requirement R2:** If DB shows supplier S_x as being under contract on days d and $d+1$, it must contain exactly one tuple that shows that fact
- **Requirement R3:** If DB shows supplier S_x as being under contract on day d , it must also show supplier S_x as having some status on day d

Note: R1 has to do with avoiding *redundancy*, R2 with avoiding *circumlocution*, and R3 has to do with *denseness*

-
- **Requirement R4:** If DB shows supplier S_x as having some status on day d , it must contain exactly one tuple that shows that fact
 - **Requirement R5:** If DB shows supplier S_x as having the same status on days d and $d+1$, it must contain exactly one tuple that shows that fact
 - **Requirement R6:** If DB shows supplier S_x as having some status on day d , it must also show supplier S_x as being under contract on day d

Note: R4 has to do with avoiding *redundancy* and *contradiction*, R5 with avoiding *circumlocution*, and R6 has to do with *denseness*

-
- **Requirement R7:** If DB shows supplier S_x as able to supply some specific part P_y on day d , it must contain exactly one tuple that shows that fact
 - **Requirement R8:** If DB shows supplier S_x as able to supply the same part P_y on days d and $d+1$, it must contain exactly one tuple that shows that fact
 - **Requirement R9:** If DB shows supplier S_x as able to supply some part P_y on day d , it must also show supplier S_x as being under contract on day d

Note: R7 has to do with avoiding *redundancy* (not contradiction), R8 has to do with avoiding *circumlocution*, and R9 has to do with *denseness*

POINTS ARISING :

- Requirement **R1** is based on the fact that no supplier can be under two contracts at the same time
- Requirement **R4** is based on the fact that no supplier can have two status values at the same time
- Requirement **R7** is based on the fact that no supplier can have two “abilities to supply” the same part at the same time

POINTS ARISING :

- Requirement **R1** is based on the fact that no supplier can be under two contracts at the same time
- Requirement **R4** is based on the fact that no supplier can have two status values at the same time
- Requirement **R7** is based on the fact that no supplier can have two “abilities to supply” the same part at the same time
- Requirements that specify “exactly one tuple” (i.e., all except R3, R6, R9) can’t possibly be satisfied, in general, if full vertical decomposition hasn’t been done ... in other words, if DB contains any during relvars not in 6NF

Detailed analysis of these requirements is nontrivial!

Mostly, therefore, I'll just present the corresponding formal constraints */* for combination design */* as a *fait accompli* without too much by way of justification ...

For proper analysis and justification (also for since relvars only and during relvars only), see the textbook!

REQUIREMENT R1 :

If DB shows supplier S_x as being under contract on day d , it must contain exactly one tuple that shows that fact

- Avoid **redundancy!** */* re being under contract */*
—within or across S_SINCE and S_DURING
- No such redundancy in S_SINCE because of KEY
- No such redundancy in S_DURING because of PACKED ON

```
CONSTRAINT BR1  /* B for both */  
  IS_EMPTY ( (  $S\_SINCE$  JOIN  $S\_DURING$  )  
              WHERE  $SNO\_SINCE < POST ( DURING )$  );
```


REQUIREMENT R2 :

If DB shows supplier S_x as being under contract on days d and $d+1$, it must contain exactly one tuple that shows that fact

- Avoid **circumlocution!**
—within or across S_SINCE and S_DURING
- No circumlocution in S_SINCE because of KEY
- No circumlocution in S_DURING because of PACKED ON

CONSTRAINT BR2

```
IS_EMPTY ( ( S_SINCE JOIN S_DURING )  
           WHERE SNO_SINCE = POST ( DURING ) ) ;
```

Obviously makes sense to combine constraints BR1 and BR2:

```
CONSTRAINT BR12
  IS_EMPTY ( ( S_SINCE JOIN S_DURING )
             WHERE SNO_SINCE ≤ POST ( DURING ) ) ;
```

Note that END (DURING) can't possibly be “the end of time”

REQUIREMENT R3 :

If DB shows supplier S_x as being under contract on day d , it must also show supplier S_x as having some status on day d

- A **denseness** constraint

REQUIREMENT R3 :

If DB shows supplier S_x as being under contract on day d , it must also show supplier S_x as having some status on day d

- A **denseness** constraint
- *Re being under contract:* Combine info from S_SINCE and S_DURING into S_DURING'

REQUIREMENT R3 :

If DB shows supplier S_x as being under contract on day d , it must also show supplier S_x as having some status on day d

- A **denseness** constraint
- *Re being under contract:* Combine info from S_SINCE and S_DURING into S_DURING'
- *Re having some status:* Combine info from S_SINCE and S_STATUS_DURING into S_STATUS_DURING'

REQUIREMENT R3 :

If DB shows supplier S_x as being under contract on day d , it must also show supplier S_x as having some status on day d

- A **denseness** constraint
- *Re being under contract:* Combine info from S_SINCE and S_DURING into S_DURING'
- *Re having some status:* Combine info from S_SINCE and S_STATUS_DURING into S_STATUS_DURING'
- Then there'll be *a foreign U_key constraint* from S_DURING' to S_STATUS_DURING' /* every {SNO,DURING} value in UNPACK S_DURING' ON (DURING) must also appear in UNPACK S_STATUS_DURING' ON (DURING) */

CONSTRAINT BR3

```
WITH ( t1 := EXTEND S_SINCE : { DURING := INTERVAL_DATE
                                ([ SNO_SINCE : LAST_DATE () ] ) },
      t2 := t1 { SNO , DURING } ,
      t3 := t2 UNION S_DURING ,
```

CONSTRAINT BR3

```
WITH ( t1 := EXTEND S_SINCE : { DURING := INTERVAL_DATE  
                                ( [ SNO_SINCE : LAST_DATE ( ) ] ) ) ,  
      t2 := t1 { SNO , DURING } ,  
      t3 := t2 UNION S_DURING ,  
      t4 := EXTEND S_SINCE : { DURING := INTERVAL_DATE  
                                ( [ STATUS_SINCE : LAST_DATE ( ) ] ) ) ,  
      t5 := t4 { SNO , STATUS , DURING } ,  
      t6 := t5 UNION S_STATUS_DURING ) :  
USING ( DURING ) : t3 { SNO , DURING }  $\subseteq$  t6 { SNO , DURING } ;
```

Note: t3 is S_DURING' ... t6 is S_STATUS_DURING'

If DB shows supplier S_x as being under contract on day d , it must also show supplier S_x as having some status on day d

In particular, a supplier with a contract history must have a status history ...
though the converse is false (why?)

If DB shows supplier Sx as being under contract on day *d*, it must also show supplier Sx as having some status on day *d*

In particular, a supplier with a contract history must have a status history ...
though the converse is false (why?)

So there's *a foreign U_key constraint* from S_DURING to S_STATUS_DURING /* every {SNO,DURING} value in UNPACK S_DURING ON (DURING) must also appear in UNPACK S_STATUS_DURING ON (DURING) */

If DB shows supplier Sx as being under contract on day *d*, it must also show supplier Sx as having some status on day *d*

In particular, a supplier with a contract history must have a status history ...
though the converse is false (why?)

So there's *a foreign U_key constraint* from S_DURING to S_STATUS_DURING /* every {SNO,DURING} value in UNPACK S_DURING ON (DURING) must also appear in UNPACK S_STATUS_DURING ON (DURING) */

```
USING ( DURING ) : FOREIGN KEY { SNO , DURING }  
REFERENCES S_STATUS_DURING
```

If DB shows supplier Sx as being under contract on day *d*, it must also show supplier Sx as having some status on day *d*

In particular, a supplier with a contract history must have a status history ...
though the converse is false (why?)

So there's *a foreign U_key constraint* from S_DURING to S_STATUS_DURING */* every {SNO,DURING} value in UNPACK S_DURING ON (DURING) must also appear in UNPACK S_STATUS_DURING ON (DURING) */*

USING (DURING) : FOREIGN KEY { SNO , DURING }
REFERENCES S_STATUS_DURING

/ part of definition of relvar S_DURING ... see "XFT2" */*

REQUIREMENT R6 :

If DB shows supplier S_x as having some status on day d , it must also show supplier S_x as being under contract on day d

- Inverse of R3 !!!
- *Re having some status:* Combine info from S_SINCE and S_STATUS_DURING into S_STATUS_DURING'
- *Re being under contract:* Combine info from S_SINCE and S_DURING into S_DURING'
- Then there'll be *a foreign U_key constraint* from S_STATUS_DURING' to S_DURING'
- Replace \subseteq in BR3 by \supseteq ... or rather:

CONSTRAINT BR36

```
WITH ( t1 := EXTEND S_SINCE : { DURING := INTERVAL_DATE  
                                ( [ SNO_SINCE : LAST_DATE ( ) ] ) ) ,  
      t2 := t1 { SNO , DURING } ,  
      t3 := t2 UNION S_DURING ,  
      t4 := EXTEND S_SINCE : { DURING := INTERVAL_DATE  
                                ( [ STATUS_SINCE : LAST_DATE ( ) ] ) ) ,  
      t5 := t4 { SNO , STATUS , DURING } ,  
      t6 := t5 UNION S_STATUS_DURING ) :  
USING ( DURING ) : t3 { SNO , DURING } = t6 { SNO , DURING } ;
```

This is a U_EQD (“U_equality dependency”) ... Regular EQDs are a special case, of course

REQUIREMENT R4 :

If DB shows supplier S_x as having some status on day d , it must contain exactly one tuple that shows that fact

- Avoid **redundancy** and **contradiction!**
—within or across S_SINCE and S_STATUS_DURING
- No redundancy or contradiction in S_SINCE because of KEY
- No redundancy or contradiction in S_STATUS_DURING because of PACKED ON and WHEN / THEN

CONSTRAINT BR4

```
IS_EMPTY ( ( S_SINCE JOIN S_STATUS_DURING { SNO , DURING } )  
           WHERE STATUS_SINCE < POST ( DURING ) ) ;
```

REQUIREMENT R5 :

If DB shows supplier S_x as having the same status on days d and $d+1$, it must contain exactly one tuple that shows that fact

- Avoid **circumlocution!**
—within or across S_SINCE and S_STATUS_DURING
- No circumlocution in S_SINCE because of KEY
- No circumlocution in S_STATUS_DURING because of PACKED ON

CONSTRAINT BR5

```
IS_EMPTY ( ( S_SINCE JOIN S_STATUS_DURING )  
           WHERE STATUS_SINCE = POST ( DURING ) ) ;
```


REQUIREMENT R7 :

If DB shows supplier S_x as able to supply some specific part P_y on day d , it must contain exactly one tuple that shows that fact

- Avoid **redundancy!**
—within or across SP_SINCE and SP_DURING
- No redundancy in SP_SINCE because of KEY
- No redundancy in SP_DURING because of PACKED ON

CONSTRAINT BR7

```
IS_EMPTY ( ( SP_SINCE JOIN SP_DURING )  
           WHERE SINCE < POST ( DURING ) ) ;
```

REQUIREMENT R8 :

If DB shows supplier S_x as able to supply the same part P_y on days d and $d+1$, it must contain exactly one tuple that shows that fact

- Avoid **circumlocution!**
—within or across SP_SINCE and SP_DURING
- No circumlocution in SP_SINCE because of KEY
- No circumlocution in SP_DURING because of PACKED ON

CONSTRAINT BR8

```
IS_EMPTY ( ( SP_SINCE JOIN SP_DURING )  
           WHERE SINCE = POST ( DURING ) ) ;
```

Obviously makes sense to combine Constraints BR7 and BR8:

CONSTRAINT BR78

```
IS_EMPTY ( ( SP_SINCE JOIN SP_DURING )  
            WHERE SINCE ≤ POST ( DURING ) ) ;
```

REQUIREMENT R9 :

If the database shows supplier S_x as able to supply some part P_y on day d , it must also show supplier S_x as being under contract on day d

- A **denseness** constraint ... very similar to R3 and R6

CONSTRAINT BR9

```
WITH (  $t1$  := EXTEND S_SINCE : { DURING := INTERVAL_DATE  
                                ( [ SNO_SINCE : LAST_DATE ( ) ] ) ) ,  
       $t2$  :=  $t1$  { SNO , DURING } ,  
       $t3$  :=  $t2$  UNION S_DURING ,  
       $t4$  := EXTEND SP_SINCE : { DURING := INTERVAL_DATE  
                                ( [ SINCE : LAST_DATE ( ) ] ) ) ,  
       $t5$  :=  $t4$  { SNO , DURING } ,  
       $t6$  := SP_DURING { SNO , DURING } ) :  
USING ( DURING ) :  $t3$  { SNO , DURING }  $\supseteq$   $t6$  { SNO , DURING } ;
```

/ takes care of the XFT3 problem (see much earlier) */*

CAN WE DO ANYTHING ABOUT ALL THIS COMPLEXITY ???

Yes!

Consider KEY & FOREIGN KEY syntax in conventional DBs ...

- Really just shorthand (save some writing)
- But effectively *raise the level of abstraction*
- Not to mention possibility of better performance!

Can do the same kind of thing in temporal DBs (i.e., recognize and exploit certain commonly occurring abstract patterns):

CRUCIAL OBSERVATIONS :

1. Each **since** relvar concerns certain **entities** and represents certain time-varying **properties** of those entities
2. Within each since relvar, entities are identified by **key** attributes and properties are represented by other attributes (as usual), and some since relvars have **foreign keys** that reference others (again as usual)
3. Within each since relvar, each time-varying property has a **SINCE** attribute, and so does the key—and no SINCE attribute has a value $<$ that of the one associated with the key (in any given tuple)

-
4. The key and each time-varying property each have an associated **during** relvar, consisting of:
 - a. An attribute (combination) corresponding to the since relvar key
 - b. An attribute (combination) corresponding to the since relvar time-varying property
 - c. An associated **DURING** attribute
 5. Each during relvar is kept **packed** on DURING

-
6. Each during relvar is [either “all key” or] subject to a certain **WHEN ... THEN ...** constraint
 7. Each combination of a time-varying property (or key) in a since relvar and the corresponding during relvar is subject to certain constraints as implied by Requirements R1-R9

Hence:

TENTATIVE SYNTACTIC EXTENSIONS :

```
VAR S_SINCE RELATION
{ SNO          SNO ,
  SNO_SINCE    DATE SINCE_FOR { SNO } ,
  STATUS       INTEGER ,
  STATUS_SINCE DATE SINCE_FOR { STATUS } }
KEY { SNO } ;
```

```
VAR SP_SINCE RELATION
{ SNO  SNO ,
  PNO  PNO ,
  SINCE DATE SINCE_FOR { SNO , PNO } }
KEY { SNO , PNO }
FOREIGN KEY { SNO } REFERENCES S ;
```

Implies *inter alia* that following must be true:

IS_EMPTY (S_SINCE WHERE STATUS_SINCE < SNO_SINCE)

```
VAR S_SINCE RELATION
{ SNO          SNO ,
  SNO_SINCE     DATE SINCE_FOR { SNO }
                                HISTORY_IN ( S_DURING ) ,
  STATUS        INTEGER ,
  STATUS_SINCE  DATE SINCE_FOR { STATUS }
                                HISTORY_IN ( S_STATUS_DURING ) }
KEY { SNO } ;
```

```
VAR SP_SINCE RELATION
{ SNO  SNO ,
  PNO  PNO ,
  SINCE DATE SINCE_FOR { SNO , PNO }
                                HISTORY_IN ( SP_DURING ) }
KEY { SNO , PNO }
FOREIGN KEY { SNO } REFERENCES S ;
```

System knows that S_DURING, S_STATUS_DURING, and SP_DURING *must* be defined ... ***Definitions could be automated!***

VAR S_DURING RELATION

```
{ SNO      SNO ,  
  DURING INTERVAL_DATE }  
USING ( DURING ) : KEY { SNO , DURING } ;
```

VAR S_STATUS_DURING RELATION

```
{ SNO      SNO ,  
  STATUS INTEGER ,  
  DURING INTERVAL_DATE }  
USING ( DURING ) : KEY { SNO , DURING } ;
```

VAR SP_DURING RELATION

```
{ SNO      SNO ,  
  PNO      PNO ,  
  DURING INTERVAL_DATE }  
USING ( DURING ) : KEY { SNO , PNO , DURING } ;
```

We conjecture that the system should now be able to *infer* constraints for Requirements R1-R9 for itself

We further conjecture that the system might now be able, automatically, to perform certain *compensatory actions*—analogous to (e.g.) cascade delete—whenever certain constraints would otherwise be violated, thereby simplifying DB maintenance

Further research needed

PART III cont. :

Building on the foundations:

- 12. Database design I : Structure
- 13. Database design II : Keys and related constraints
- 14. Database design III : General constraints
- 15. Queries
- 16. Updates
- 17. Logged time and stated time
- 18. Point and interval types revisited

BREAK :

Next = queries

SAMPLE QUERIES :

- Q1:** Get the status of S1 on day dn
- Q2:** Get pairs of SNOs such that the indicated suppliers were assigned some status on the same day
- Q3:** Get SNOs for suppliers able to supply both P1 and P2 at the same time
- Q4:** Get SNOs for suppliers never able to supply both P1 and P2 at the same time

-
- Q5:** Get SNOs for suppliers who, while under some specific contract, changed their status since they most recently became able to supply some part under that contract
- Q6:** Get intervals during which at least one supplier was under contract
- Q7:** Let BUSY = result of Q6; use BUSY to get intervals during which no supplier was under contract at all
- Q8:** Get SNOs for suppliers currently under contract who also had an earlier contract

-
- Q9:** Get SNO-PARTS-DURING triples such that the indicated supplier was able to supply the indicated range of parts during the indicated interval
- Q10:** Let S_PARTS_DURING = result of Q9; use S_PARTS_DURING to get SNO-PNO-DURING triples such that the indicated supplier was able to supply the indicated part during the indicated interval
- Q11:** Given relvar TERM, with attributes DURING, PRESIDENT, and TERMNO, and both {DURING} and {PRESIDENT, TERMNO} as U_keys, get DURING-PRESIDENT pairs such that the indicated president held office throughout the indicated interval
- Q12:** Given relvar TERM as above, get pairs of presidents who held office in the same year

NOTE :

Some of these queries (e.g., Q6, Q7, and Q9) can be handled only very inelegantly—sometimes not at all!—in some of the other temporal database approaches described in the literature

Why?

Because they typically violate *The Information Principle !!!*

(E.g., consider Q6 ... result would be “hidden attribute” only)

Anyway, let's see how the queries might be formulated **in terms of our “combination” design** /* see textbook for other designs */

QUERY Q1 :

Get the status of S1 on day dn

Desired info could be in either S_SINCE or S_STATUS_DURING ...

```
WITH ( t1 := S_SINCE WHERE SNO = SNO ( 'S1' ) ,  
      t2 := EXTEND t1 : { DURING :=  
                          INTERVAL_DATE ( [ STATUS_SINCE : LAST_DATE ( ) ] ) } ,  
      t3 := t2 { STATUS , DURING } ,  
      t4 := S_STATUS_DURING WHERE SNO = SNO ( 'S1' ) ,  
      t5 := t4 { STATUS , DURING } ,  
      t6 := t3 UNION t5 ) :  
( t6 WHERE  $dn \in$  DURING ) { STATUS }
```

QUERY Q2 :

Get pairs of SNOs such that the indicated suppliers were assigned some status on the same day

```
WITH ( t1 := ( EXTEND S_STATUS_DURING :  
                { STATUS_SINCE := BEGIN ( DURING ) } )  
                { SNO , STATUS_SINCE } ,  
  t2 := t1 UNION S_SINCE { SNO , STATUS_SINCE } ) ,  
  t3 := t2 RENAME { SNO AS XNO } ,  
  t4 := t2 RENAME { SNO AS YNO } ,  
  t5 := t3 JOIN t4 ,  
  t6 := t5 WHERE XNO < YNO ) :  
t6 { XNO , YNO }
```

QUERY Q3 :

Get SNOs for suppliers able to supply both P1 and P2 at the same time

```
WITH ( t1 := ( EXTEND SP_SINCE : { DURING :=  
                                INTERVAL_DATE ( [ SINCE : LAST_DATE ( ) ] ) } )  
                                { SNO , PNO , DURING } ,  
      t2 := SP_DURING UNION t1 ,  
      t3 := ( t2 WHERE PNO = PNO ( 'P1' ) ) { SNO , DURING } ,  
      t4 := ( t2 WHERE PNO = PNO ( 'P2' ) ) { SNO , DURING } ,  
      t5 := USING ( DURING ) : t3 JOIN t4 ) :  
t5 { SNO }
```

QUERY Q4 :

Get SNOs for suppliers never able to supply both P1 and P2 at the same time

```
WITH ( t1 := ( EXTEND SP_SINCE : { DURING :=  
                                INTERVAL_DATE ( [ SINCE : LAST_DATE ( ) ] ) } )  
                                { SNO , PNO , DURING } ,  
      t2 := SP_DURING UNION t1 ,  
      t3 := ( t2 WHERE PNO = PNO ( 'P1' ) ) { SNO , DURING } ,  
      t4 := ( t2 WHERE PNO = PNO ( 'P2' ) ) { SNO , DURING } ,  
      t5 := USING ( DURING ) : t3 JOIN t4 ) :  
t2 { SNO } MINUS t5 { SNO }
```

QUERY Q5 :

Get SNOs for suppliers who, while under some specific contract, changed their status since they most recently became able to supply some part under that contract

```
WITH ( t1 := ( EXTEND S_SINCE : { DURING :=  
                                INTERVAL_DATE ( [ SNO_SINCE : LAST_DATE ( ) ] ) } )  
                                { SNO , DURING } ,  
t2 := t1 UNION S_DURING ,  
t3 := ( EXTEND S_SINCE : { DURING :=  
                                INTERVAL_DATE ( [ STATUS_SINCE : LAST_DATE ( ) ] ) } )  
                                { SNO , STATUS , DURING } ,  
t4 := t3 UNION S_STATUS_DURING ,  
t5 := ( EXTEND SP_SINCE : { DURING :=  
                                INTERVAL_DATE ( [ SINCE : LAST_DATE ( ) ] ) } )  
                                { SNO , PNO , DURING } ,  
t6 := t5 UNION SP_DURING ,  
  
/* cont. */
```

QUERY Q5 (cont.) :

```
t7 := ( t4 RENAME { DURING AS X } ) { SNO , X } ,
t8 := t7 JOIN t2 ,
t9 := t8 WHERE X  $\subseteq$  DURING ,
t10 := ( t6 RENAME { DURING AS Y } ) { SNO , Y } ,
t11 := t10 JOIN t9 ,
t12 := t11 WHERE Y  $\subseteq$  DURING ,
t13 := EXTEND t12 { SNO , DURING } :
      { BXMAX := MAX ( !!t12 , BEGIN ( X ) ) ,
        BYMAX := MAX ( !!t12 , BEGIN ( Y ) ) } ,
t14 := t13 WHERE BXMAX > BYMAX ) :
t14 { SNO }
```


QUERY Q6 :

Get intervals during which at least one supplier was under contract

```
WITH ( t1 := EXTEND S_SINCE : { DURING :=  
                                INTERVAL ( [ SNO_SINCE : LAST_DATE ( ) ] ) } ,  
      t2 := t1 { SNO , DURING } UNION S_DURING ) :  
USING ( DURING ) : t2 { DURING }
```

QUERY Q7 :

Let BUSY = result of Q6; use BUSY to get intervals during which no supplier was under contract at all

```
USING ( DURING ) :  
    RELATION  
        { TUPLE { DURING  
                    INTERVAL_DATE  
                    ( [ FIRST_DATE ( ) : LAST_DATE ( ) ] ) } }  
    MINUS BUSY
```

Note: We would prefer that the implementation not physically materialize the results of the implicit UNPACKs here ... especially the first one!

QUERY Q8 :

Get SNOs for suppliers currently under contract who also had an earlier contract

S_SINCE { SNO } JOIN S_DURING { SNO }

QUERY Q9 :

Get SNO-PARTS-DURING triples such that the indicated supplier was able to supply the indicated range of parts during the indicated interval

```
WITH ( t1 := EXTEND SP_SINCE :  
        { PARTS := INTERVAL_PNO ( [ PNO : PNO ] ) ,  
          DURING := INTERVAL_DATE  
            ( [ SINCE : LAST_DATE ( ) ] ) } ,  
  t2 := t1 { SNO , PARTS, DURING } ,  
  t3 := EXTEND SP_DURING :  
        { PARTS := INTERVAL_PNO ( [ PNO : PNO ] ) } ,  
  t4 := t3 { SNO , PARTS , DURING } ) :  
USING ( PARTS , DURING ) : t3 UNION t4
```

QUERY Q10 :

Let S_PARTS_DURING = result of Q9;
use S_PARTS_DURING to get SNO-PNO-DURING triples such that
the indicated supplier was able to supply the indicated part during the
indicated interval

```
WITH ( t1 := UNPACK S_PARTS_DURING ON ( PARTS ) ,  
      t2 := EXTEND t1 : { PNO := POINT FROM PARTS } ) :  
USING ( DURING ) : t2 { ALL BUT PARTS }
```

QUERY Q11 :

Given relvar TERM, with attributes DURING, PRESIDENT, and TERMNO, and both {DURING} and {PRESIDENT, TERMNO} as U_keys, get DURING-PRESIDENT pairs such that the indicated president held office throughout the indicated interval

PACK TERM { DURING , PRESIDENT } ON (DURING)

QUERY Q12 :

Given relvar TERM, get pairs of presidents who held office in the same year

```
WITH ( t1 := TERM RENAME { DURING AS D1 , PRESIDENT AS P1 } ,  
      t2 := TERM RENAME { DURING AS D2 , PRESIDENT AS P2 } ,  
      t3 := ( t1 JOIN t2 ) WHERE P1 ≠ P2 AND END ( D1 ) = PRE ( D2 ) ) :  
t3 { P1 , P2 }
```

CAN THESE QUERIES BE SIMPLIFIED ???

Yes!—using *virtual relvars* (“views”)

Define a set of views that have the effect of conceptually undoing the horizontal and vertical decompositions described earlier

Undo *horizontal* decompositions:

```
VAR S_DURING' VIRTUAL
  ( S_DURING UNION
    ( EXTEND S_SINCE :
      { DURING := INTERVAL_DATE
        ( [ SNO_SINCE , LAST_DATE ( ) ] ) } )
      { SNO , DURING } ) ;
```

```
VAR S_STATUS_DURING' VIRTUAL
  ( S_STATUS_DURING UNION
    ( EXTEND S_SINCE :
      { DURING := INTERVAL_DATE
        ( [ STATUS_SINCE , LAST_DATE ( ) ] ) } )
      { SNO , STATUS , DURING } ) ;

VAR SP_DURING' VIRTUAL
  ( SP_DURING UNION
    ( EXTEND SP_SINCE :
      { DURING := INTERVAL_DATE
        ( [ SINCE , LAST_DATE ( ) ] ) } )
      { SNO , PNO , DURING } ) ;
```

Undo *vertical* decompositions:

```
VAR S" VIRTUAL  
  ( USING ( DURING ) : S_DURING' JOIN  
                                     S_STATUS_DURING' ) ;
```

```
VAR SP" VIRTUAL  
  ( SP_DURING' ) ;
```

Exercises: What are the keys? What are the predicates?

Queries now somewhat simpler (though often still nontrivial)

***Provide foregoing definitions automatically, via some
“COMBINED_IN” option?***

PART III cont. :

Building on the foundations:

- 12. Database design I : Structure
- 13. Database design II : Keys and related constraints
- 14. Database design III : General constraints
- 15. Queries
- 16. Updates
- 17. Logged time and stated time
- 18. Point and interval types revisited

BREAK :

Next = updates

UPDATES :

- Since relvars only */* actually not much to say */*
- During relvars only
 - U_ updates
 - PORTION specifications
 - Multiple assignment
- Both since and during relvars

*A good way to think about updates in general is to think in terms of adding, removing, and replacing **propositions** (not tuples)*

SINCE RELVARS : SAMPLE VALUES

S_SINCE

SNO	SNO_SINCE	STATUS	STATUS_SINCE
S1	d04	20	d06
S2	d07	10	d07
S3	d03	30	d03
S4	d14	20	d14
S5	d02	30	d02

SP_SINCE

SNO	PNO	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	P5	d14

To repeat, a good way to think about updates in general is to think in terms of adding, removing, and replacing ***propositions*** rather than tuples ... Following predicates deliberately given in simplified form to match propositions in the examples more closely

S_SINCE: *Supplier S_x has been under contract since day d_c and has had status st since day d_s*

SP_SINCE: *Supplier S_x has been able to supply part P_y since day d*

U1: Add proposition to show S9 has just been placed under contract, with status 15

```
INSERT S_SINCE
      RELATION { TUPLE { SNO          SNO ( 'S9' ) ,
                          SNO_SINCE    TODAY ( ) ,
                          STATUS        15 ,
                          STATUS_SINCE  TODAY ( ) } } ;
```

Assume availability of built in op **TODAY ()** ... Adds proposition
“Supplier S9 has been under contract since day *dc* and has had
status 15 since day *ds*” (where *dc* and *ds* are both the date today)

Note: D_INSERT might be better than INSERT in this example

U2: Remove proposition showing S5 is under contract

DELETE S_SINCE WHERE SNO = SNO ('S5') ;

Removes proposition “*Supplier S5 has been under contract since day dc and has had status st since day ds*” (where *dc*, *st*, and *ds* are whatever they happened to be in the current tuple for S5)

Note: I_DELETE might be better than DELETE in this example

U3: Replace proposition showing S1 was placed under contract on day 4 by one for day 3 instead

```
UPDATE S_SINCE WHERE SNO = SNO ( 'S1' ) :  
                { SNO_SINCE := d03 } ;
```

Or:

```
S_SINCE := ( S_SINCE WHERE NOT ( SNO = SNO ( 'S1' ) ) )  
            UNION  
            ( EXTEND ( S_SINCE WHERE SNO = SNO ( 'S1' ) ) :  
              { SNO_SINCE := d03 } ) ;
```

Removes one proposition, adds another

UPDATES cont. :

- Since relvars only
- During relvars only
 - U_updates
 - PORTION specifications
 - Multiple assignment
- Both since and during relvars

DURING RELVARS : SAMPLE VALUES

S_DURING

SNO	DURING
S2	[d02:d04]
S6	[d03:d05]
S7	[d03:d99]

S_STATUS_DURING

SNO	STATUS	DURING
S2	5	[d02:d02]
S2	10	[d03:d04]
S6	5	[d03:d04]
S6	7	[d05:d05]
S7	15	[d03:d08]
S7	20	[d09:d99]

SP_DURING

SNO	PNO	DURING
S2	P1	[d02:d04]
S2	P2	[d03:d03]
S2	P5	[d03:d04]
S6	P3	[d03:d05]
S6	P4	[d04:d04]
S6	P5	[d04:d05]
S7	P1	[d03:d04]
S7	P1	[d06:d07]
S7	P1	[d09:d99]

Simplified predicates:

S_DURING:	<i>Supplier Sx was under contract during interval i</i>
S_STATUS_DURING:	<i>Supplier Sx had status st during interval i</i>
SP_DURING:	<i>Supplier Sx was able to supply part Py during interval i</i>

Focus on SP_DURING (mostly) until further notice ...

U4: Add proposition “S2 was able to supply P4 on day 2”

```
INSERT SP_DURING
  RELATION { TUPLE
    { SNO      SNO ( 'S2' ) ,
      PNO      PNO ( 'P4' ) ,
      DURING   INTERVAL_DATE ( [ d02 : d02 ] ) } } ;
```

U4: Add proposition “S2 was able to supply P4 on day 2”

```
INSERT SP_DURING
  RELATION { TUPLE
    { SNO      SNO ( 'S2' ) ,
      PNO      PNO ( 'P4' ) ,
      DURING    INTERVAL_DATE ( [ d02 : d02 ] ) } } ;
```

*But suppose it was part **P5**, not part P4 ...*

INSERT violates PACKED ON constraint! If accepted:

SNO	PNO	DURING
S2	P5	[d02:d02]

SNO	PNO	DURING
S2	P5	[d03:d04]

U_INSERT :

USING (*ACL*) : INSERT *R r*

Shorthand for *R* := USING (*ACL*) : *R* UNION *r* ... i.e.:

R := PACK ((UNPACK *R* ON (*ACL*))
UNION
(UNPACK *r* ON (*ACL*))) ON (*ACL*)

U_INSERT :

USING (ACL) : INSERT R r

Shorthand for $R := \text{USING (ACL) : } R \text{ UNION } r \dots$ i.e.:

```
R := PACK ( ( UNPACK R ON ( ACL ) )  
            UNION  
            ( UNPACK r ON ( ACL ) ) ) ON ( ACL )
```

Now:

```
USING ( DURING ) : INSERT SP_DURING  
RELATION { TUPLE  
    { SNO      SNO ( 'S2' ) ,  
      PNO      PNO ( 'Py' ) , /* OK if Py = P4 or P5 */  
      DURING    INTERVAL_DATE ( [ d02 : d02 ] ) } } ;
```

U_INSERT (cont.) :

Can decrease cardinality of target relvar!

If *ACL* empty, U_INSERT reduces to regular INSERT

Regular INSERT can violate a key constraint ... U_INSERT can violate a **U_key** constraint (more precisely, a WHEN / THEN constraint): e.g.,

```
USING ( DURING ) : INSERT S_STATUS_DURING
RELATION { TUPLE
    { SNO      SNO ( 'S2' ) ,
      STATUS  20 ,
      DURING  INTERVAL_DATE ( [ d03 : d03 ] ) } } ;
```

U_INSERT (cont.) :

Regular INSERT can violate a foreign key constraint ...
U_INSERT can violate a **foreign U_key** constraint: e.g.,

```
USING ( DURING ) : INSERT SP_DURING  
RELATION { TUPLE  
            { SNO      SNO ( 'S2' ) ,  
              PNO      PNO ( 'P4' ) ,  
              DURING INTERVAL_DATE ( [ d01 : d03 ] ) } } ;
```

Note: We can and do additionally define “disjoint U_INSERT”

U_DELETE :

U5: Remove proposition “S6 was able to supply P3 from day 3 to day 5”

/ Why? Isn't this supposed to be a historical DB? See */*
/ “valid time vs. transaction time”, later */*

```
DELETE SP_DURING
      WHERE SNO = SNO ( 'S6' )
      AND     PNO = PNO ( 'P3' )
      AND     DURING = INTERVAL_DATE ( [ d03 : d05 ] ) ;
```

U_DELETE :

U5: Remove proposition “S6 was able to supply P3 from day 3 to day 5”

/ Why? Isn't this supposed to be a historical DB? See */
/* “valid time vs. transaction time”, later */*

```
DELETE SP_DURING
      WHERE SNO = SNO ( 'S6' )
      AND    PNO = PNO ( 'P3' )
      AND    DURING = INTERVAL_DATE ( [ d03 : d05 ] ) ;
```

But suppose it was day **4**, not day 3 ...

DELETE analogous to the one above has no effect!
(No such tuple in SP_DURING)

U_DELETE (cont.) :

USING (*ACL*) : DELETE *R* WHERE *bx*

Shorthand for $R := \text{USING (} \textit{ACL} \text{) : } R \text{ WHERE NOT (} \textit{bx} \text{) ... i.e.:$

$R := \text{PACK}$
 $((\text{UNPACK } R \text{ ON (} \textit{ACL} \text{)) WHERE NOT (} \textit{bx} \text{))}$
 $\text{ON (} \textit{ACL} \text{)}$

U_DELETE (cont.) :

USING (ACL) : DELETE R WHERE *bx*

Shorthand for $R := \text{USING (ACL) : } R \text{ WHERE NOT (} bx \text{) ... i.e.:}$

$R := \text{PACK}$
 $((\text{UNPACK } R \text{ ON (ACL)) WHERE NOT (} bx \text{))}$
 ON (ACL)

Now:

USING (DURING) : DELETE SP_DURING
 WHERE SNO = SNO ('S6')
 AND PNO = PNO ('P3')
 AND DURING \subseteq INTERVAL_DATE ([*d04* : *d05*]) ;

↑
note!

U_DELETE (cont.) :

Can increase cardinality of target relvar!

If *ACL* empty, U_DELETE reduces to regular DELETE

Note: Final PACK step *is* necessary ... E.g., consider

SNO	PARTS	DURING
S1	[P1 : P2]	[d01 : d03]
S1	[P3 : P3]	[d01 : d02]

/ packed on (DURING,PARTS) */*

```
USING ( PARTS , DURING ) : DELETE S_PARTS_DURING
WHERE DURING  $\subseteq$  INTERVAL_DATE ( [d01 : d02] ) ;
```

/ exercise for the reader! */*

U_DELETE (cont.) :

USING (ACL) : DELETE R r

Shorthand for $R := \text{USING (ACL) : } R \text{ MINUS } r \dots$ i.e.:

```
R := PACK
      ( ( UNPACK R ON ( ACL ) )
        MINUS
        ( UNPACK r ON ( ACL ) ) )
      ON ( ACL )
```

Can be useful / see textbook */ ... But we take “U_DELETE” to mean “U_DELETE WHERE” specifically, most of the time*

Note: We can and do additionally define “included I_DELETE”

U_UPDATE :

U6: Replace proposition “S2 was able to supply P5 from day 3 to day 4” by “S2 was able to supply P5 from day 2 to day 4”

```
UPDATE SP_DURING
    WHERE SNO = SNO ( 'S2' )
    AND     PNO = PNO ( 'P5' )
    AND     DURING = INTERVAL_DATE ( [ d03 : d04 ] ) :
    { DURING := INTERVAL_DATE ( [ d02 : d04 ] ) } ;
```

But suppose we wanted to replace “S2 was able to supply P5 on day 3” by “S2 was able to supply P5 on day 2”

UPDATE analogous to the one above has no effect!
(No such tuple in SP_DURING)

U_UPDATE (cont.) :

USING (*ACL*) : UPDATE *R* WHERE *bx* : { *attribute assignments* }

Shorthand for: WITH (*t1* := UNPACK *R* ON (*ACL*) ,
 t2 := *t1* WHERE NOT (*bx*) ,
 t3 := *t1* MINUS *t2* ,
 t4 := EXTEND *t3* : { *attribute assignments* } ,
 t5 := *t2* UNION *t4*) :
 R := PACK *t5* ON (*ACL*)

U_UPDATE (cont.) :

USING (ACL) : UPDATE *R* WHERE *bx* : { *attribute assignments* }

Shorthand for: WITH (*t1* := UNPACK *R* ON (ACL) ,
 t2 := *t1* WHERE NOT (*bx*) ,
 t3 := *t1* MINUS *t2* ,
 t4 := EXTEND *t3* : { *attribute assignments* } ,
 t5 := *t2* UNION *t4*) :
 R := PACK *t5* ON (ACL)

Now:

USING (DURING) : UPDATE SP_DURING
 WHERE SNO = SNO ('S2')
 AND PNO = PNO ('P5')
 AND DURING = INTERVAL_DATE ([*d03* : *d03*]) :
 { DURING := INTERVAL_DATE ([*d02* : *d02*]) } ;

If ACL empty, U_UPDATE reduces to regular UPDATE

U_UPDATE (cont.) :

But there's another way to do the foregoing UPDATE, a way that some might find more intuitively pleasing ...

```
UPDATE SP_DURING
  WHERE SNO = SNO ( 'S2' )
  AND    PNO = PNO ( 'P5' ) :
  PORTION { DURING { INTERVAL_DATE ( [ d03 : d03 ] ) } } :
  { DURING := INTERVAL_DATE ( [ d02 : d02 ] ) } ;
```

/ replace “S2 was able to supply P5 on day 3” by “S2 was able to supply P5 on day 2” */*

UPDATES cont. :

- Since relvars only
- During relvars only
 - U_ updates
 - PORTION specifications
 - Multiple assignment
- Both since and during relvars

PORTION DELETE :

U7: Remove proposition “S6 was able to supply P3 from day 4 to day 5” */* the modified form of U5 */*

```
DELETE SP_DURING
      WHERE SNO = SNO ( 'S6' )
      AND    PNO = PNO ( 'P3' ) :
      PORTION { DURING { INTERVAL_DATE ( [ d04 : d05 ] ) } } ;
```

Previous formulation / just to remind you */ :*

```
USING ( DURING ) : DELETE SP_DURING
      WHERE SNO = SNO ( 'S6' )
      AND    PNO = PNO ( 'P3' )
      AND    DURING  $\subseteq$  INTERVAL_DATE ( [ d04 : d05 ] ) ;
```

PORTION (cont.) :

DELETE R WHERE $bx : \text{PORTION} \{ A \{ ix \} \}$

Shorthand for:

```
WITH (  $t1 := R$  WHERE (  $bx$  ) AND  $A$  OVERLAPS  $ix$  ,  
       $t2 := R$  MINUS  $t1$  ,  
       $t3 := \text{UNPACK } t1$  ON (  $A$  ) ,  
       $t4 := t3$  WHERE NOT (  $A \subseteq ( ix )$  ) ,  
       $t5 := t2$  UNION  $t4$  ) :  
 $R := \text{PACK } t5$  ON (  $A$  )
```

PORTION UPDATE is analogous

General form of PORTION specification allows reference to any number of interval attributes with any number of portions on each
/ see textbook */*

ASIDE :

PORTION could be useful with restrict too ... E.g.,

S_DURING

PORTION { DURING { INTERVAL_DATE ([d05 : d08]) } }

S_DURING

SNO	DURING
S2	[d02:d04]
S6	[d03:d05]
S7	[d03:d99]

result

SNO	DURING
S6	[d05:d05]
S7	[d05:d08]

BREAK :

Next = updates (cont.)

UPDATES cont. :

- Since relvars only
- During relvars only
 - U_ updates
 - PORTION specifications
 - Multiple assignment
- Both since and during relvars

MULTIPLE ASSIGNMENT :

Now turn to relvars S_DURING and S_STATUS_DURING:

```
S_DURING { SNO , DURING }  
  USING ( DURING ) : KEY { SNO , DURING }  
  USING ( DURING ) : FOREIGN KEY { SNO , DURING }  
    REFERENCES S_STATUS_DURING
```

```
/* every {SNO,DURING} value in UNPACK S_DURING ON ( DURING ) */  
/* must appear as an {SNO,DURING} value—i.e., a key value—in */  
/* UNPACK S_STATUS_DURING ON ( DURING ) */
```

MULTIPLE ASSIGNMENT :

Now turn to relvars *S_DURING* and *S_STATUS_DURING*:

```
S_DURING { SNO , DURING }  
  USING ( DURING ) : KEY { SNO , DURING }  
  USING ( DURING ) : FOREIGN KEY { SNO , DURING }  
    REFERENCES S_STATUS_DURING
```

```
/* every {SNO,DURING} value in UNPACK S_DURING ON ( DURING ) */  
/* must appear as an {SNO,DURING} value—i.e., a key value—in */  
/* UNPACK S_STATUS_DURING ON ( DURING ) */
```

```
S_STATUS_DURING { SNO , STATUS , DURING }  
  USING ( DURING ) : KEY { SNO, DURING }  
  USING ( DURING ) : FOREIGN KEY { SNO , DURING }  
    REFERENCES S_DURING
```

```
/* every {SNO,DURING} value in UNPACK S_STATUS_DURING ON */  
/* ( DURING ) must appear as an {SNO,DURING} value—i.e., a key */  
/* value—in UNPACK S_DURING ON ( DURING ) */
```

MULTIPLE ASSIGNMENT (cont.) :

U12:* Add propositions to show S9 has just been placed under contract, with status 15 */* same as U1 */*

```
INSERT S_DURING RELATION { TUPLE
    { SNO      SNO ( 'S9' ) ,
      DURING INTERVAL_DATE ( [ TODAY ( ) : d99 ] ) } } ,
```

```
INSERT S_STATUS_DURING RELATION { TUPLE
    { SNO      SNO ( 'S9' ) ,
      STATUS 15 ,
      DURING INTERVAL_DATE ( [ TODAY ( ) : d99 ] ) } } ;
```

Need two INSERTs ... *but only one statement* (note the comma separator)! Really a **multiple assignment**:

* *Numbering as in textbook*

MULTIPLE ASSIGNMENT (cont.) :

```
S_DURING :=  
  S_DURING UNION RELATION { TUPLE  
    { SNO      SNO ( 'S9' ) ,  
      DURING INTERVAL_DATE ( [ TODAY ( ) : d99 ] ) } } ,
```

```
S_STATUS_DURING :=  
  S_STATUS_DURING UNION RELATION { TUPLE  
    { SNO      SNO ( 'S9' ) ,  
      STATUS 15  
      DURING INTERVAL_DATE ( [ TODAY ( ) : d99 ] ) } } ;
```

- Semantics:
1. Evaluate source expressions
 2. Update target variables “in parallel” *
 3. Check applicable constraints

* *But assignments to the same target are done sequentially*

MULTIPLE ASSIGNMENT (cont.) :

U13: Remove all propositions showing S7 as being under contract

```
DELETE SP_DURING WHERE SNO = SNO ( 'S7' ) ,
```

```
DELETE S_STATUS_DURING WHERE SNO = SNO ( 'S7' ) ,
```

```
DELETE S_DURING WHERE SNO = SNO ( 'S7' ) ;
```


MULTIPLE ASSIGNMENT (cont.) :

U14: S7's contract has just been terminated /* like U2 */

```
UPDATE S_DURING WHERE SNO = SNO ( 'S7' )  
                AND TODAY ( ) ∈ DURING :  
                { END ( DURING ) := TODAY ( ) } ,
```

```
UPDATE S_STATUS_DURING WHERE SNO = SNO ( 'S7' )  
                AND TODAY ( ) ∈ DURING :  
                { END ( DURING ) := TODAY ( ) } ,
```

```
UPDATE SP_DURING WHERE SNO = SNO ( 'S7' )  
                AND TODAY ( ) ∈ DURING :  
                { END ( DURING ) := TODAY ( ) } ;
```

-
- Note direct assignments to END (DURING) ... See *The Third Manifesto*
 - Note that “deletion” of certain propositions is done by means of UPDATE operations!
 - Often no simple correspondence between inserting or deleting or replacing *propositions* and inserting or deleting or replacing *tuples*
 - Of course, any individual update within a multiple assignment can be a “U_update”

For interest, here's an explicit assignment equivalent to the first of those three UPDATES:

```
WITH ( t := S_DURING WHERE SNO = SNO ( 'S7' )  
      AND TODAY ( ) ∈ DURING ) :  
  DELETE S_DURING t ,  
  INSERT S_DURING  
    ( EXTEND t : { END ( DURING ) := TODAY ( ) } ) ;
```

This expansion shows very clearly that an UPDATE can be thought of as a DELETE followed by an INSERT ***on the same target*** ... in which case the individual assignments are executed in sequence as written

UPDATES cont. :

- Since relvars only
- During relvars only
 - U_ updates
 - PORTION specifications
 - Multiple assignment
- Both since and during relvars

SINCE RELVARS : SAMPLE VALUES

S_SINCE

SNO	SNO_SINCE	STATUS	STATUS_SINCE
S1	d04	20	d06
S2	d07	10	d07
S3	d03	30	d03
S4	d04	20	d08
S5	d02	30	d02

SP_SINCE

SNO	PNO	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	P5	d05

DURING RELVARS : SAMPLE VALUES

S_DURING

SNO	DURING
S2	[d02:d04]
S6	[d03:d05]

S_STATUS_DURING

SNO	STATUS	DURING
S1	15	[d04:d05]
S2	5	[d02:d02]
S2	10	[d03:d04]
S4	10	[d04:d04]
S4	25	[d05:d07]
S6	5	[d03:d04]
S6	7	[d05:d05]

SP_DURING

SNO	PNO	DURING
S2	P1	[d02:d04]
S2	P2	[d03:d03]
S3	P5	[d05:d07]
S4	P2	[d06:d09]
S4	P4	[d04:d08]
S6	P3	[d03:d03]
S6	P3	[d05:d05]

U15: Add proposition “S4 has been able to supply P4 since day 10”

```
INSERT SP_SINCE
      RELATION { TUPLE { SNO    SNO ( 'S4' ) ,
                          PNO    PNO ( 'P4' ) ,
                          SINCE  d10 } } ;
```

U15: Add proposition “S4 has been able to supply P4 since day 10”

```
INSERT SP_SINCE
      RELATION { TUPLE { SNO    SNO ( 'S4' ) ,
                        PNO    PNO ( 'P4' ) ,
                        SINCE d10 } } ;
```

But suppose it was day **9**, not day 10 ...

INSERT violates Requirement R8! (“If DB shows Sx able to supply same part Py on days *d* and *d*+1, it must contain exactly one tuple that shows that fact”). If accepted:

SNO	PNO	SINCE
S4	P4	<i>d09</i>

SNO	PNO	DURING
S4	P4	[<i>d04</i> : <i>d08</i>]

What we need to do:

- Delete the tuple for S4 and P4 with DURING = [d04:d08] from SP_DURING
- Insert a tuple for S4 and P4 with SINCE = d04 into SP_SINCE

```
DELETE SP_DURING WHERE SNO = SNO ( 'S4' )  
                        AND    PNO = PNO ( 'P4' )  
                        AND    DURING = [d04:d08] ,
```

```
INSERT SP_SINCE RELATION { TUPLE { SNO    SNO ( 'S4' ) ,  
                                   PNO    PNO ( 'P4' ) ,  
                                   SINCE  d04 } } ;
```

Foregoing code is specific to a certain specific update and a certain specific set of existing data values ...

Generic code to insert proposition “Sx has been able to supply Py since day d ”:

```
WITH ( temp := SP_DURING WHERE SNO = Sx AND PNO = Py AND
                                              $d \leq$  POST ( DURING ) ) :
CASE ;
    WHEN IS_EMPTY ( temp )
        THEN INSERT SP_SINCE RELATION { TUPLE
                                         { SNO Sx , PNO Py , SINCE  $d$  } } ;
    ELSE DELETE SP_DURING temp ,
        INSERT SP_SINCE RELATION { TUPLE { SNO Sx, PNO Py,
                                             SINCE MIN ( temp , BEGIN ( DURING ) ) } } ;
END CASE ;
```

POINTS ARISING :

- Not intended that *users* should actually have to write code like this ... !!!
- Key constraint violation if SP_SINCE tuple for Sx & Py already exists
- Foreign key constraint violation if S_SINCE tuple for Sx does not already exist
- And a bunch of other constraint violations, possibly

U16: S1 is no longer able to supply P1

INSERT SP_DURING RELATION

{ TUPLE { SNO SNO ('S1') , PNO PNO ('P1') ,
DURING INTERVAL_DATE ([d04 : TODAY ()]) } } ,

DELETE SP_SINCE

WHERE SNO = SNO ('S1') AND PNO = PNO ('P1') ;

U16: S1 is no longer able to supply P1

INSERT SP_DURING RELATION

```
{ TUPLE { SNO SNO ( 'S1' ) , PNO PNO ( 'P1' ) ,  
        DURING INTERVAL_DATE ( [ d04 : TODAY ( ) ] ) } } ,
```

DELETE SP_SINCE

```
WHERE SNO = SNO ( 'S1' ) AND PNO = PNO ( 'P1' ) ;
```

Note tacit assumption: viz., original proposition was true!
Suppose we simply made a mistake and tuple should never
have been inserted in the first place ...

... then we need the DELETE but not the INSERT

Ignore such considerations henceforth, for simplicity

Generic code to remove proposition “Sx has been able to supply Py since day d ”:

```
WITH ( temp := SP_SINCE WHERE SNO = Sx AND PNO = Py ) :  
IF IS_NOT_EMPTY ( temp ) THEN  
    INSERT SP_DURING RELATION  
        { TUPLE { SNO Sx , PNO Py , DURING INTERVAL_DATE  
            ( [ SINCE FROM ( TUPLE FROM temp ) : TODAY ( ) ] ) } } ,  
    DELETE SP_SINCE temp ;  
END IF ;
```

U17: Replace “S2 has been able to supply P1 since day 8” by
“S2 has been able to supply P1 since day 7”

```
UPDATE SP_SINCE
      WHERE SNO = SNO ( 'S2' ) AND PNO = PNO ( 'P1' ) :
           { SINCE := d07 } ;
```

U17: Replace “S2 has been able to supply P1 since day 8” by
“S2 has been able to supply P1 since day 7”

UPDATE SP_SINCE

WHERE SNO = SNO ('S2') AND PNO = PNO ('P1') :
{ SINCE := *d07* } ;

But suppose it was day **5**, not day 7 ...

UPDATE violates requirement R8! (“If DB shows S_x able to supply same part P_y on days *d* and *d*+1, it must contain exactly one tuple that shows that fact”). If accepted:

SNO	PNO	SINCE
S2	P1	<i>d05</i>

SNO	PNO	DURING
S2	P1	[<i>d02</i> : <i>d04</i>]

Generic code to replace proposition “Sx has been able to supply Py since day d ” by “Sx has been able to supply Py since day d' ”

```
WITH ( temp := SP_DURING WHERE SNO = Sx AND PNO = Py AND
                                              $d \leq \text{POST ( DURING )}$  ) :
CASE ;
  WHEN IS_EMPTY ( temp )
  THEN UPDATE SP_SINCE WHERE SNO = Sx AND PNO = Py :
                                             { SINCE :=  $d'$  } ;
  ELSE DELETE SP_DURING temp ,
        UPDATE SP_SINCE WHERE SNO = Sx AND PNO = Py :
        { SINCE := MIN ( temp , BEGIN ( DURING ) ) } ;
END CASE ;
```

EXERCISES FOR THE READER :

- Have considered attempts to change SINCE component of an SP_SINCE tuple but not SNO or PNO components—but similar considerations apply. Try:
 - Replace PNO component of SP_SINCE tuple for S4 & P5 by P4
 - Replace SNO component of SP_SINCE tuple for S1 & P4 by S4
- Have considered updates affecting SP_SINCE and SP_DURING but not S_SINCE, S_DURING, and S_STATUS_DURING—but similar considerations apply

CONCLUDING REMARKS :

- Updating a temporal DB has the potential to be a seriously complicated matter! ... **but views can help**
- Rarely seems to be possible to talk in terms of inserting, deleting, or updating *tuples*; seems to make more sense to talk in terms of inserting, deleting or updating *propositions*
- If DB involves mixture of current and historical relvars, then:
 - Often impossible to talk about just one of INSERT, DELETE, and UPDATE in isolation
 - Often impossible to talk about updating just current relvars or just historical relvars in isolation

UPDATES cont. :

- Since relvars only
- During relvars only
 - U_ updates
 - PORTION specifications
 - Multiple assignment
- Both since and during relvars

PART III cont. :

Building on the foundations:

- 12. Database design I : Structure
- 13. Database design II : Keys and related constraints
- 14. Database design III : General constraints
- 15. Queries
- 16. Updates
- 17. Logged time and stated time
- 18. Point and interval types revisited

BREAK :

Next = logged time and stated time

RECALL :

S_DURING tuple

SNO	DURING
S2	[d02:d04]

/* Proposition: *Supplier S2 was under contract* */
/* *from day 2 to day 4* */

Valid time for proposition “Supplier S2 was under contract”
= interval from day 2 to day 4 */* note that DURING values in our examples prior to this point all represented “valid times” */*

RECALL :

S_DURING tuple

SNO	DURING
S2	[d02:d04]

/* Proposition: *Supplier S2 was under contract* */
/* *from day 2 to day 4* */

Valid time for proposition “Supplier S2 was under contract”
= interval from day 2 to day 4 */* note that DURING values in our examples prior to this point all represented “valid times” */*

Suppose tuple exists in DB from $t1$ to $t2$ (only). Then
transaction time for proposition “Supplier S2 was under contract from day 2 to day 4” = interval from $t1$ to $t2$

POINTS ARISING :

Valid times: Apply to **something we currently believe to be true** (updatable; can refer to past, present, or future) ... *Concept applies only to temporal relvars*

Transaction times: Apply to **when the database said something was true** (not updatable; never refer to the future) ... *Concept applies to every relvar in the DB*

The two “somethings” are, in general, *different!*

Both “times” are **sets of intervals** (in general)—possibly an empty set of intervals

A CLOSER LOOK :

S_DURING tuple

SNO	DURING
S2	[d02:d04]

/* Proposition: *Supplier S2 was under contract* */
/* *from day 2 to day 4* */

Warning: *This gets tricky ...*

Let p be “Supplier S2 was under contract from day 2 to day 4”

Let q be “Supplier S2 was under contract”

Valid time for q is [d02:d04] (that’s what p says)

Transaction time for p is [t1:t2]

-
- p does NOT imply q (“Supplier S2 was under contract”)!—rather, it implies “Supplier S2 was under contract *at some time*” /* not the same thing, logically speaking */

-
- p does NOT imply q (“Supplier S2 was under contract”)!—rather, it implies “Supplier S2 was under contract *at some time*” /* not the same thing, logically speaking */
 - This latter proposition also has transaction time $[t1:t2]$, but proposition q (“Supplier S2 was under contract”) does NOT—in fact, proposition q isn’t represented in the DB at all, not even implicitly

-
- p does NOT imply q (“Supplier S2 was under contract”)!—rather, it implies “Supplier S2 was under contract *at some time*” /* not the same thing, logically speaking */
 - This latter proposition also has transaction time $[t1:t2]$, but proposition q (“Supplier S2 was under contract”) does NOT—in fact, proposition q isn’t represented in the DB at all, not even implicitly
 - Thus, (a) derived propositions in general have **transaction** times too, and (b) **valid** times typically apply to propositions that don’t appear in the DB at all, neither explicitly nor implicitly!

-
- p does NOT imply q (“Supplier S2 was under contract”)!—rather, it implies “Supplier S2 was under contract *at some time*” /* not the same thing, logically speaking */
 - This latter proposition also has transaction time $[t1:t2]$, but proposition q (“Supplier S2 was under contract”) does NOT— in fact, proposition q isn’t represented in the DB at all, not even implicitly
 - Thus, (a) derived propositions in general have **transaction** times too, and (b) **valid** times typically apply to propositions that don’t appear in the DB at all, neither explicitly nor implicitly!
 - *But note that if vt is the valid time for q , it’s also the valid time for any proposition implied by q (e.g, “Some supplier was under contract”)*

-
- Statement “**Transaction time for p is $[t1:t2]$** ” is itself a proposition!—and can be represented as a tuple:

SNO	DURING	X_DURING
S2	$[d02:d04]$	$[t1:t2]$

/ a “bitemporal tuple” */*
/ —see later */*

- Note that the two timestamps apply to **two different propositions** ... Loosely, the tuple shows the transaction time tt for the valid time vt for proposition q

SIMPLER EXAMPLE :

FIGURE	NO_OF_SIDES
Triangle	3

Proposition z:
“Triangles have
three sides”

- z has no “valid time” component
- *z does have a valid time!*—viz., “always” (we believe z is, always was, and always will be true)—which is precisely why we don’t record it explicitly
- In general, a proposition represented in the DB without an explicit “valid time” timestamp ***implicitly*** has a “valid time” timestamp of “always”
- Certainly has a transaction time, just like any other proposition represented at any time in the DB ...

BITEMPORAL TUPLE REPEATED :

SNO	DURING	X_DURING
S2	[<i>d02</i> : <i>d04</i>]	[<i>t1</i> : <i>t2</i>]

/ “tuple bt” */*

Suppose tuple *bt* currently appears in the DB

/ see later! */*

Proposition is “Transaction time for *p* is [*t1*:*t2*]”—where *p* is “Supplier S2 was under contract from day 2 to day 4”

This proposition has (implicit) valid time “always” and transaction time from *t* to the present time (assuming *bt* was inserted into DB at time *t* and $t \geq t2$)

PRECISE DEFINITIONS :

- The **transaction time** for a proposition p is the set of times t such that, according to what the database stated at time t , p was true
 - Times (in the past) when ***the database SAID we believed*** something is, was, or will be true ... p must have been represented in the DB at some time (possibly implicitly) and can refer to the past and/or the present and/or the future

PRECISE DEFINITIONS :

- The **transaction time** for a proposition p is the set of times t such that, according to what the database stated at time t , p was true
 - Times (in the past) when ***the database SAID we believed*** something is, was, or will be true ... p must have been represented in the DB at some time (possibly implicitly) and can refer to the past and/or the present and/or the future
- The **valid time** for a proposition q is the set of times t such that, according to what the database currently states (which is to say, according to our current beliefs), q is, was, or will be true at time t
 - Times (past and/or present and/or future) when, ***according to what we believe right now***, something is, was, or will be true ... q probably isn't represented in the DB at all (tho' it might be)

THE DATABASE AND THE LOG :

Significant operational difference between valid and transaction time concepts:

Valid times are kept in the DB, transaction times are kept in the log

To elaborate:

- A DB is really a *variable*; “updating the DB” causes the current value to be replaced by another value
- Values are **DB values**, variable is a **DB variable**
- Another way to think about it: Updating the DB doesn’t *replace* current DB value by another—rather, it derives a “new” DB value from the “old” one (and makes it current), *but keeps the old value around in the system as well*

-
- Overall DB = ***sequence*** or stack of DB values, each timestamped with the time of the update that produced it, ordered chronologically (most recent = current)

-
- Overall DB = ***sequence*** or stack of DB values, each timestamped with the time of the update that produced it, ordered chronologically (most recent = current)
 - *Only* kind of update we can apply takes the current DB value and derives a new one from it, which then becomes current in turn

-
- Overall DB = ***sequence*** or stack of DB values, each timestamped with the time of the update that produced it, ordered chronologically (most recent = current)
 - *Only* kind of update we can apply takes the current DB value and derives a new one from it, which then becomes current in turn
 - That sequence is the ***log!*** (conceptually)—provides full historical record of every update that has ever been made to the DB; most recent entry records our current beliefs

-
- Overall DB = ***sequence*** or stack of DB values, each timestamped with the time of the update that produced it, ordered chronologically (most recent = current)
 - *Only* kind of update we can apply takes the current DB value and derives a new one from it, which then becomes current in turn
 - That sequence is the ***log!*** (conceptually)—provides full historical record of every update that has ever been made to the DB; most recent entry records our current beliefs
 - “The DB is not the DB” — ???

TRANSACTION TIMES :

Can't be updated—but it must be possible to *query* them (e.g., for audit purposes)

Two immediate problems:

1. System doesn't really maintain the log in the form we've described ... Hence, highly unlikely that we'll be allowed to formulate queries against “the DB value at time t ” directly
2. Timestamp t isn't part of that DB value but is, rather, a kind of tag on that value ... Hence, can't formulate a relational query referencing that timestamp directly

So timestamps must be made available in *standard relational form* as part of *current DB value* ... ***Wait just a moment!***

TERMINOLOGY :

Don't often *need* a term for “valid time” (why not?)

Note that the concept was hardly mentioned prior to the present discussion!

But we do need to discuss “transaction time”, and would be nice to have a better term than “valid time” too

We selected:

Stated time (sometimes *currently* stated time, for emphasis)
for “valid time”

Logged time for “transaction time”

Both terms can be used in singular or plural

LOGGED TIME RELVARS :

VAR S_DURING RELATION

{ SNO SNO , DURING INTERVAL_DATE }

USING (DURING) : KEY { SNO , DURING }

LOGGED_TIMES_IN (S_DURING_LOG) ;

System provides relvar S_DURING_LOG, with attributes SNO, DURING, and X_DURING, and with tuples representing logged times for all tuples that have ever appeared, explicitly or implicitly, in relvar S_DURING

Like a virtual relvar, but values derived from log, not from other relvars

E.g., suppose today is day 75. Possible values:

S_DURING

SNO	DURING
S2	[d02:d04]
S6	[d03:d05]

S_DURING_LOG

1
2
3
4
5
6
7

SNO	DURING	X_DURING
S2	[d02:d04]	[d04:d07]
S2	[d02:d04]	[d10:d20]
S2	[d02:d04]	[d50:d75]
S6	[d02:d05]	[d15:d25]
S6	[d03:d05]	[d26:d75]
S1	[d01:d01]	[d20:d30]
S1	[d05:d06]	[d40:d50]

Question: Aren't those "present day" values in relvar S_DURING_LOG (i.e., d75) a bad idea?

Answer: **No!**

1. Relvar materialized only when referenced, and then only in part
2. X_DURING is updated by the system, not the user
3. In any case d75 is correct!

```
VAR S_SINCE RELATION
  { SNO SNO , SNO_SINCE DATE ,
    STATUS INTEGER , STATUS_SINCE DATE }
  KEY { SNO }
  LOGGED_TIMES_IN ( S_SINCE_LOG ) ;
```

System provides relvar S_SINCE_LOG, with attributes SNO, SNO_SINCE, STATUS, STATUS_SINCE, and X_DURING, and with tuples representing logged times for all tuples that have ever appeared, explicitly or implicitly, in relvar S_SINCE

/ see textbook for sample values */*

SAMPLE QUERIES :

X1: When if ever did the DB show supplier S6 as being under contract on day 4? */* assume we know supplier S6 isn't currently under contract */*

```
WITH ( t1 := S_DURING_LOG WHERE SNO = SNO ( 'S6' )  
                                AND d04 ∈ DURING ) :  
USING ( X_DURING ) : t1 { X_DURING }
```

SAMPLE QUERIES :

X2: When if ever did the DB show supplier S2 as being under contract on day 4?

```
WITH ( t1 := ( S_SINCE_LOG WHERE SNO = SNO ( 'S2' )  
              AND d04 ≥ SNO_SINCE ) { X_DURING } ,  
      t2 := ( S_DURING_LOG WHERE SNO = SNO ( 'S2' )  
              AND d04 ∈ DURING ) { X_DURING } ) :  
USING ( X_DURING ) : t1 UNION t2
```


SAMPLE QUERIES :

X3: On day 8, what did the DB say was supplier S2's term of contract?

```
WITH ( t1 := S_SINCE_LOG WHERE SNO = SNO ( 'S2' )  
      AND d08 ∈ X_DURING ,  
      t2 := EXTEND t1 : { DURING :=  
      INTERVAL_DATE ( [ SNO_SINCE : d99 ] ) } ,  
      t3 := t2 { DURING } ,  
      t4 := S_DURING_LOG WHERE SNO = SNO ( 'S2' )  
      AND d08 ∈ X_DURING ,  
      t5 := t4 { DURING } ) :  
USING ( DURING ) : t3 UNION t5
```

PART III cont. :

Building on the foundations:

- 12. Database design I : Structure
- 13. Database design II : Keys and related constraints
- 14. Database design III : General constraints
- 15. Queries
- 16. Updates
- 17. Logged time and stated time
- 18. Point and interval types revisited

BREAK :

Next = type inheritance

POINT AND INTERVAL TYPES REVISITED :

- Type inheritance (background)
- Point types revisited
- NUMERIC point types
- Granularity revisited
- Interval types revisited

POINT TYPES : *recap*

Type T can be used as a point type if all of the following are defined for it:

- A **total ordering** (“ \leq ” etc. available for any pair of values of type T)
- Niladic **FIRST** and **LAST** operators
- Monadic **NEXT** and **PRIOR** operators (can fail)
 - **NEXT** = *successor function*
 - Successor function *assumed unique* (?)

I.e., if T is an **ordinal type** /* see next page */ ... Thus, e.g., DATE is a valid point type

ORDERED vs. ORDINAL :

An **ordered** type is a type for which a total ordering is defined

An **ordinal** type is an ordered type for which certain additional operators are required (*first, last, next, prior ...*)

INTEGER is an ordinal type ... So is DATE

RATIONAL is an ordered type but not an ordinal type

/ in mathematics, at least, if not in computer arithmetic */* ... No “next” rational number immediately following p/q

INTERVALS : *recap*

Let T be a point type. Then an **interval** (value) i of type $\text{INTERVAL_}T$ is a scalar value for which two monadic ops, BEGIN and END , and one dyadic op, \in , are defined, such that:

- $\text{BEGIN}(i)$ and $\text{END}(i)$ each return a value of type T
- $\text{BEGIN}(i) \leq \text{END}(i)$
- If p is a value of type T , then $p \in i$ is true iff $\text{BEGIN}(i) \leq p$ and $p \leq \text{END}(i)$ are both true

Intervals are always nonempty

MUCH MORE TO BE SAID !

Stated conditions are SUFFICIENT for T to be usable as a point type—but are they NECESSARY ??? (No)

In particular, might need *two distinct* successor functions (e.g., for type DATE: next day, next month) ???

/ or next year, next week, next business day ... etc., etc. */*

Actually “unique successor function” assumption will turn out to be not too far off the mark after all ... But I need to lay a lot of groundwork first ...

TYPE INHERITANCE :

The key to the problem! (What problem?)

I'll briefly sketch major features of our type inheritance model (based on *The Third Manifesto* and described in the *Manifesto* book)

In fact, we believe most other approaches to inheritance (including those implemented in commercial products) are *logically questionable*

TYPE INHERITANCE :

The key to the problem! (What problem?)

I'll briefly sketch major features of our type inheritance model (based on *The Third Manifesto* and described in the *Manifesto* book)

In fact, we believe most other approaches to inheritance (including those implemented in commercial products) are *logically questionable*

Consider type “calendar dates” (assume NOT built in)

Type name = **DDATE** (accurate to the *day*)

DDATE is an ordinal type—values are ordered chronologically, successor function = “next day”

```
TYPE DDATE POSSREP DPRI
  { DI INTEGER CONSTRAINT DI ≥ 1 AND DI ≤ N } ;
      └────────── DDATE type constraint──────────┘
```

Every type requires among other things an *equals* op (“=”) and, for each declared “possrep”, a *selector* op and a set of *THE_* ops (one *THE_* op for each possrep component)

In practice, definitions for some at least of these ops should be provided automatically (i.e., by the system), but we show them explicitly for clarity:

```
OPERATOR DPRI ( DI INTEGER ) RETURNS DDATE ;  
    /* code to return the DDATE value represented */  
    /* by the given integer DI */  
END OPERATOR ;
```

```
OPERATOR THE_DI ( DD DDATE ) RETURNS INTEGER ;  
    /* code to return the integer that represents */  
    /* the given DDATE value DD */  
END OPERATOR ;
```

```
OPERATOR "=" ( DD1 DDATE , DD2 DDATE ) RETURNS BOOLEAN ;  
    RETURN ( THE_DI ( DD1 ) = THE_DI ( DD2 ) ) ;  
END OPERATOR ;
```

Can now specify or *select* a DDATE value by supplying appropriate positive integer—e.g.:

DPRI (59623)

Not very user friendly! So:

TYPE DDATE

POSSREP DPRI { DI INTEGER

CONSTRAINT $DI \geq 1$ AND $DI \leq N$ }

POSSREP DPRC { DC CHAR /* *string of form yyyy/mm/dd* */
CONSTRAINT ... } ;

Every DPRI value must be representable as a DPRC value and vice versa ... We also need:

```
OPERATOR DPRC ( DC CHAR ) RETURNS DDATE ;  
    /* code to return the DDATE value represented */  
    /* by the given yyyy/mm/dd string DC          */  
END OPERATOR ;
```

```
OPERATOR THE_DC ( DD DDATE ) RETURNS CHAR ;  
    /* code to return the yyyy/mm/dd string that */  
    /* represents the given DDATE value DD      */  
END OPERATOR ;
```

E.g., $\text{DPRC}('2014/01/18') \Rightarrow \text{January 18th, 2014}$
 $\text{THE_DC}(d) \Rightarrow \text{string representation of DDATE value } d$

(In practice we would want many other ops too)

Sometimes not interested in dates accurate to the day—sometimes accuracy to (e.g.) the *month* is all we need (day within month is irrelevant)

Equivalently: Might be interested only in DDATE values that correspond to the first of the month

- Like counting in tens!

If we're interested only in a proper subset of the values that make up some type T , then—by definition!—we're dealing with a **subtype** T' of T ... /* as with, e.g., *SQUARE* and *RECTANGLE* */

SOME BASIC DEFINITIONS :

- T' is a **subtype** of T iff every value of type T' is a value of type T (hence T is a subtype of itself)

SOME BASIC DEFINITIONS :

- T' is a **subtype** of T iff every value of type T' is a value of type T (hence T is a subtype of itself) ... and if T'' is a subtype of T' and T' is a subtype of T , then T'' is a subtype of T /* *but not an **immediate** one* */

SOME BASIC DEFINITIONS :

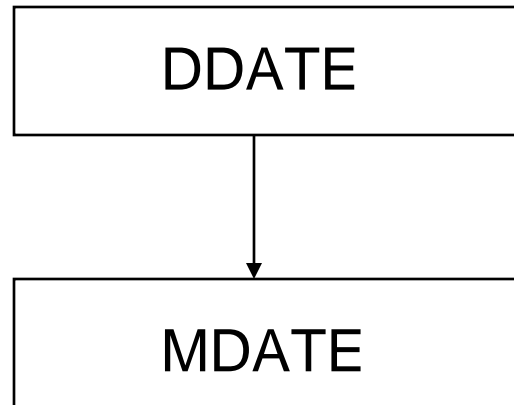
- T' is a **subtype** of T iff every value of type T' is a value of type T (hence T is a subtype of itself) ... and if T'' is a subtype of T' and T' is a subtype of T , then T'' is a subtype of T /* *but not an **immediate** one* */
- If T' is a subtype of T *and* there's at least one value of type T that's *not* a value of type T' , then T' is a **proper** subtype of T

SOME BASIC DEFINITIONS :

- T' is a **subtype** of T iff every value of type T' is a value of type T (hence T is a subtype of itself) ... and if T'' is a subtype of T' and T' is a subtype of T , then T'' is a subtype of T /* *but not an **immediate** one* */
- If T' is a subtype of T *and* there's at least one value of type T that's *not* a value of type T' , then T' is a **proper** subtype of T
- Iff T' is a subtype of T , then T is a **supertype** of T' (and T is a *proper* supertype of T' if and only if T' is a proper subtype of T)

MDATE AS A SUBTYPE OF DDATE :

Simple
*type
hierarchy*



Assume no other
types, for
simplicity

```
TYPE MDATE  
  IS DDATE  
  CONSTRAINT FIRST_OF_MONTH ( DDATE )  
  POSSREP MPRI { MI = THE_DI ( DDATE ) }  
  POSSREP MPRC { MC = SUBSTR ( THE_DC ( DDATE ) , 1 , 7 ) } ;  
  /* MDATE values as yyyy/mm strings */
```

DDATE value d is an MDATE value iff $\text{FIRST_OF_MONTH}(d)$ is true

MDATE AS A SUBTYPE OF DDATE (cont.) :

OPERATOR MPRI (MI INTEGER) RETURNS MDATE ;

/ code to return the MDATE value represented by given integer */*

/ MI (of course, MI must correspond to the first of the month) */*

END OPERATOR ;

OPERATOR THE_MI (MD MDATE) RETURNS INTEGER ;

/ code to return the integer that represents given MDATE value */*

/ MD */*

END OPERATOR ;

OPERATOR MPRC (MC CHAR) RETURNS MDATE ;

/ code to return the MDATE value represented by given */*

/ given yyyy/mm string MC */*

END OPERATOR ;

OPERATOR THE_MC (MD MDATE) RETURNS CHAR ;

/ code to return the yyyy/mm string that represents given */*

/ MDATE value MD */*

END OPERATOR ;

VALUE SUBSTITUTABILITY :

Wherever the system expects a value of type DDATE, we can always substitute a value of type MDATE /* because MDATE values **are** DDATE values */

- *This is THE WHOLE POINT, in fact!*

If *Op* is defined for DDATE values in general, we can invoke *Op* with an MDATE value specifically (*Op* is **polymorphic**)

E.g., ops “=”, THE_DI, THE_DC defined above for type DDATE apply to type MDATE as well (they’re **inherited** by type MDATE from type DDATE)

Converse not true, of course

Hence, e.g., following is valid:

```
VAR I INTEGER ;  
VAR MDX MDATE ;  
VAR DDX DDATE ;
```

```
I := THE_DI ( MDX ) ;  
IF MDX = DDX THEN ... END IF ;  
/* etc., etc., etc. */
```

```
DDX := MPRC ( '2014/09' ) ;
```

```
/* declared type of DDX is DDATE, but current most */
```

```
/* specific type is now MDATE */
```

```
/* same distinction applies to arbitrary exps */
```

DDX := DPRC ('2012/10/17') ;

/ current most specific type of DDX now DDATE again */*

DDX := DPRC ('2012/10/01') ;

/ DPRC invocation returns a value of type MDATE !!! */*

/ —because it satisfies FIRST_OF_MONTH constraint */*

/ —**specialization by constraint** (very important!) */*

DDX := DPRC ('2013/10/17') ;

/ current most specific type of DDX now DDATE again */*

/ —**generalization by constraint** (also important!) */*

Assignment: Most specific type of source value can be any subtype—not necessarily proper—of declared type of target variable

Equality comparison: Comparand most specific types must have a common supertype

$v1 = v2$ true iff $v1$ and $v2$ are the same value (and so have the same most specific type a fortiori)

/ end of digression on inheritance */*

POINT AND INTERVAL TYPES REVISITED :

- Type inheritance (background)
- Point types revisited
- NUMERIC point types
- Granularity revisited
- Interval types revisited

DDATE AS A POINT TYPE :

```
TYPE DDATE ORDINAL /* note new keyword */  
    POSSREP DPRI { DI INTEGER  
                    CONSTRAINT  $DI \geq 1$  AND  $DI \leq N$  }  
    POSSREP DPRC { DC CHAR  
                    CONSTRAINT ... } ;
```

ORDINAL specification implies that “ \leq ” (etc.), “first”, “last”, “next”, and “prior” ops *must* be defined

- Real language might require names of those ops to be included as part of the ORDINAL spec

Appropriate definitions:

DDATE AS A POINT TYPE (cont.) :

```
OPERATOR "≤" ( DD1 DDATE , DD2 DDATE ) RETURNS BOOLEAN ;  
    RETURN ( THE_DI ( DD1 ) ≤ THE_DI ( DD2 ) ) ;  
END OPERATOR ;
```

```
OPERATOR FIRST_DDATE ( ) RETURNS DDATE ;  
    RETURN DPRI ( 1 ) ;  
END OPERATOR ;
```

```
OPERATOR LAST_DDATE ( ) RETURNS DDATE ;  
    RETURN DPRI ( N ) ;  
END OPERATOR ;
```

DDATE AS A POINT TYPE (cont.) :

```
OPERATOR NEXT_DDATE ( DD DDATE ) RETURNS DDATE ;  
    RETURN DPRI ( THE_DI ( DD ) + 1 ) ;  
END OPERATOR ;
```

```
OPERATOR PRIOR_DDATE ( DD DDATE ) RETURNS DDATE ;  
    RETURN DPRI ( THE_DI ( DD ) - 1 ) ;  
END OPERATOR ;
```

NOW CONSIDER TYPE MDATE :

- Can regard it as an ordinal type if we want to (and we do)
- Therefore we need “ \leq ” etc., “first”, “last”, “next”, and “prior” ops for type MDATE
- “ \leq ” etc. are (*must be*) inherited from type DDATE
- So too are NEXT_DDATE etc.—but these are *not* the “next” (etc.) ops we need for type MDATE!
- E.g., given MDATE value August 1st, 2015, NEXT_DDATE will return August 2nd, 2015, not September 1st, 2015
- So we need:

MDATE AS A POINT TYPE (cont.) :

```
OPERATOR FIRST_MDATE ( ) RETURNS MDATE ;  
    RETURN MPRI ( op ( 1 ) ) ;  
    /* op ( 1 ) computes the integer corresponding */  
    /* to the first day of the first month */  
END OPERATOR ;
```

```
OPERATOR LAST_MDATE ( ) RETURNS MDATE ;  
    RETURN MPRI ( op ( N ) ) ;  
    /* op ( N ) computes the integer corresponding */  
    /* to the first day of the last month */  
END OPERATOR ;
```

MDATE AS A POINT TYPE (cont.) :

```
OPERATOR NEXT_MDATE ( MD MDATE ) RETURNS MDATE ;  
    RETURN MDATE ( THE_MI ( MD ) + incr ) ;  
    /* incr is 28, 29, 30, or 31, as applicable */  
END OPERATOR ;
```

```
OPERATOR PRIOR_MDATE ( MD MDATE ) RETURNS MDATE ;  
    RETURN MDATE ( THE_MI ( MD ) - decr ) ;  
    /* decr is 28, 29, 30, or 31, as applicable */  
END OPERATOR ;
```

And now (at last!) we see why FIRST, LAST, NEXT, and PRIOR need that *_T* qualifier ... Even when there's an argument (i.e., even for NEXT and PRIOR), argument type is insufficient to pin down the operator precisely, in general

TO SUM UP SO FAR :

- Can have intervals involving *days* (point type DDATE)
- Can have intervals involving *months* (point type MDATE)
- MDATE is a subtype of DDATE
- MDATE and DDATE have different successor functions
- Each of these types has a *unique* successor function

POINT AND INTERVAL TYPES REVISITED :

- Type inheritance (background)
- Point types revisited
- **NUMERIC point types**
- Granularity revisited
- Interval types revisited

BREAK :

Next = NUMERIC types

NUMERIC POINT TYPES :

Type generator NUMERIC (fixed point numbers) ... e.g., SQL:

```
DECLARE X NUMERIC ( 5 , 2 ) ; /* precision 5, scale factor 2 */
```

Possible values:

-999.99, -999.98, ..., -0.01, 0.0, 0.01, ..., 999.99

/ usual conventions regarding omission of insignificant leading */*
/ and trailing zeros in numeric literals */*

Precision = total number of decimal digits

Scale factor = position of assumed decimal point:

positive scale factor q = q places to left,

negative scale factor $-q$ = q places to the right,
of rightmost decimal digit

NUMERIC POINT TYPES :

Type generator NUMERIC (fixed point numbers) ... e.g., SQL:

```
DECLARE X NUMERIC ( 5 , 2 ) ; /* precision 5, scale factor 2 */
```

Possible values:

-999.99, -999.98, ..., -0.01, 0.0, 0.01, ..., 999.99

/ usual conventions regarding omission of insignificant leading */*
/ and trailing zeros in numeric literals */*

Precision = total number of decimal digits

Scale factor = position of assumed decimal point:

positive scale factor q = q places to left,

negative scale factor $-q$ = q places to the right,
of rightmost decimal digit

Precision and scale together constitute the type constraint!

Values of type NUMERIC(p,q) have an *assumed* decimal point

I.e., if x is such a value, then x can be thought of as a p -digit integer, n say, but the *actual value* denoted by that p -digit integer = $n * (10 \text{ to the power } -q)$

Multiplier 10 to the power $-q$ = **scale** defined by q
(one hundredth in the case of variable X)

Every value of the type is evenly divisible by the scale

Concept of scale applies to other types too ... E.g.:

Type DDATE: Scale = one day

Type MDATE: Scale = one month

ANALOGOUSLY :

Can define (e.g.):

- Point type where scale is calendar quarters and “next” op is “add three months”
- Point type where scale is decades and “next” op is “add 10 years”
- Etc., etc., etc.

EXAMPLES OF NUMERIC TYPES :

<i>Type</i>	<i>Scale</i>	<i>Picture</i>
NUMERIC(4,1)	1/10	xxx.x
NUMERIC(3,1)	1/10	xx.x
NUMERIC(4,2)	1/100	xx.xx
NUMERIC(3,0)	1	xxx.
NUMERIC(3,-2)	100	xxx00.
NUMERIC(3,5)	1/100000	.00xxx

NUMERIC($p,0$) /* for some p */ often spelled INTEGER

Negative scale factor = least significant digits of integer part are missing (assumed to be zeros) /* values are all integers */

Scale factor > precision = most significant digits of the fractional part are missing (assumed to be zeros)

NUMERIC SUPERTYPES AND SUBTYPES :

- Consider types NUMERIC(3,1) and NUMERIC(4,1)
- Every value of type NUMERIC(3,1) is a value of type NUMERIC(4,1) as well
- E.g., 99.9 : type NUMERIC(3,1)
vs. 099.9 : type NUMERIC(4,1) */* same value */*
- But 999.9 is of type NUMERIC(4,1) and not type NUMERIC(3,1),
so not every value of type NUMERIC(4,1) is a value of type NUMERIC(3,1)
- *NUMERIC(3,1) is a proper subtype of NUMERIC(4,1)*

By the way, note the difference between a *literal* and a *value*! ... A literal is a *symbol* that denotes a value

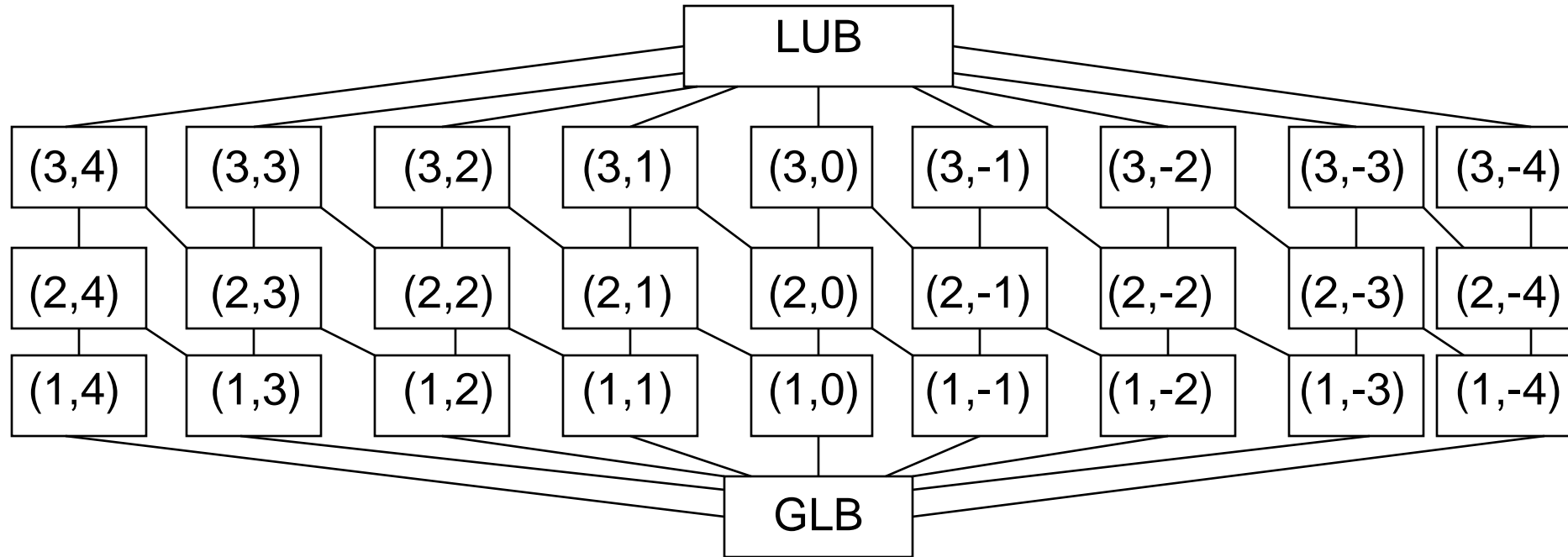
- Consider, e.g., literal 099.9
- Type is apparently NUMERIC(4,1)
- But it satisfies the type constraint for NUMERIC(3,1)—which is a subtype of NUMERIC(4,1)—and specialization by constraint occurs!
- Similarly, 5.00 denotes a value of type INTEGER

NUMERIC SUPERTYPES AND SUBTYPES :

- Consider NUMERIC(3,1) and NUMERIC(4,1) ... Every value of type NUMERIC(3,1) is a value of type NUMERIC(4,1) as well (e.g., 99.9 vs. 099.9) ... *NUMERIC(3,1) is a proper subtype of NUMERIC(4,1)*
- Consider NUMERIC(3,1) and NUMERIC(4,2) ... Every value of type NUMERIC(3,1) is a value of type NUMERIC(4,2) as well (e.g., 99.9 vs. 99.90) ... *NUMERIC(3,1) is a proper subtype of NUMERIC(4,2) as well*
- Neither of NUMERIC(4,1) and NUMERIC(4,2) is a subtype of the other (e.g., consider 999.9 and 99.99)
- Hence ***multiple inheritance!***

NUMERIC TYPE LATTICE :

Suppose max and min precision are 3 and 1, respectively, and max and min scale factors are 4 and -4, respectively



LUB = all possible NUMERIC values (“least upper bound”)

GLB = single value zero only (“greatest lower bound”)

MULTIPLE INHERITANCE :

Only extension needed to our single inheritance model (so far as we're concerned here) is:

If types $T1$ and $T2$ overlap, they must have both a common supertype T and a common subtype T' . The subtype T' (*intersection type for $T1$ and $T2$*) must be such that every value that is of both types $T1$ and $T2$ is in fact a value of type T' .

Satisfied in our NUMERIC type lattice! E.g., NUMERIC(2,1) and NUMERIC(2,0) overlap—e.g., integer 9 is a common value—and:

Common supertype = NUMERIC(3,1)

Common subtype = NUMERIC(1,0), and x is of both types iff it's a value of this common subtype

Assignment: Most specific type of source value can be any subtype—not necessarily proper—of declared type of target variable

Equality comparison: Comparand most specific types must have a common supertype

$v1 = v2$ true iff $v1$ and $v2$ are the same value (and so have the same most specific type a fortiori)

/ same as before! */*

NUMERIC types can all be used as point types (they all have a total ordering), *but*:

- “First” and “last” ops *implied* by precision
- “Next” and “prior” ops *implied* by scale

E.g., type NUMERIC(3,1):

-99.9, -99.8, ..., -0.1, 0, 0.1, ..., 99.9

“First” = -99.9

“Last” = 99.9

“Next” = add one tenth

“Prior” = subtract one tenth

Ops should really be named FIRST_NUMERIC(3,1) etc. !!!

OTHER SCALES :

NUMERIC(p,q) shorthand works *only if the desired scale is a power of ten*

Consider point type “even integers” ... must define **explicit** subtype:

```
TYPE EVEN_INTEGER ORDINAL
    IS INTEGER
    CONSTRAINT MOD ( INTEGER , 2 ) = 0 ... ;
```

Operators:

- “ \leq ” etc. inherited from type INTEGER
- “First” (etc.):

```
OPERATOR FIRST_EVEN_INTEGER ( ) RETURNS EVEN_INTEGER ;  
    RETURN -99998 ;  
END OPERATOR ;
```

```
OPERATOR LAST_EVEN_INTEGER ( ) RETURNS EVEN_INTEGER ;  
    RETURN 99998 ;  
END OPERATOR ;
```

```
OPERATOR NEXT_EVEN_INTEGER ( I EVEN_INTEGER )  
                                RETURNS EVEN_INTEGER ;  
    RETURN I + 2 ;  
END OPERATOR ;
```

```
OPERATOR PRIOR_EVEN_INTEGER ( I EVEN_INTEGER )  
                                RETURNS EVEN_INTEGER ;  
    RETURN I - 2 ;  
END OPERATOR ;
```

Note implicit appeals to specialization by constraint!

POINT AND INTERVAL TYPES REVISITED :

- Type inheritance (background)
- Point types revisited
- NUMERIC point types
- Granularity revisited
- Interval types revisited

GRANULARITY (REMEMBER THIS ?) :

“Size” of applicable points (?) or
“size” of gap between adjacent points (?)

<i>Point Type</i>	<i>Granularity</i>	<i>Scale</i>
DDATE	1 day	1 day
MDATE	1 month	1 month
NUMERIC(5,2)	1/100	1/100
NUMERIC(4,1)	1/10	1/10
NUMERIC(3,1)	1/10	1/10
NUMERIC(4,2)	1/100	1/100
NUMERIC(3,0)	1	1
NUMERIC(3,-2)	100	100
NUMERIC(3,5)	1/100000	1/100000
EVEN INTEGER	2	2

What do you conclude?

GRANULARITY NEEDS UNITS

(in general) :

TYPE HEIGHT POSSREP HIPR

{ HI INTEGER CONSTRAINT $HI > 0$ AND $HI \leq 120$ } ;

Possible HEIGHT values:

HIPR (1), HIPR (2), ..., HIPR (120)

Granularity = one inch ...

(scale) or two half inches, or 1/12 of a foot ...
 or 2.54 cm, or 25.4 mm, or ...

Many different ways to say the same thing!

*Don't have different types for different units of measure
(use different possreps instead)*

GRANULARITY MIGHT NOT MAKE SENSE (even if scale does) :

TYPE RICHTER ORDINAL

```
POSSREP RPR { R NUMERIC ( 3 , 1 )  
              CONSTRAINT R > 0.0 } ;
```

Possible Richter values:

```
RPR ( 0.1 ), RPR ( 0.2 ), . . . , RPR ( 1.0 ),  
RPR ( 1.1 ), RPR ( 1.2 ), . . . ,  
. . . . . , RPR ( 99.9 )  /* help! */
```

Scale = one tenth

Granularity makes no sense! /* same true for MDATE ??? */

Richter scale is *nonlinear*

GRANULARITY AND SCALE MIGHT ***BOTH NOT MAKE SENSE :***

```
TYPE PRIME ORDINAL  /* prime numbers */  
  IS INTEGER  
  CONSTRAINT ... ;
```

```
OPERATOR FIRST_PRIME ( ) RETURNS PRIME ;  
  RETURN 2 ;  
END OPERATOR ;
```

```
/* LAST_PRIME is analogous */
```

```
OPERATOR NEXT_PRIME ( P PRIME ) RETURNS PRIME ;  
  RETURN ( np ( P ) ) ;  
  /* np ( P ) computes the next prime */  
END OPERATOR ;
```

```
/* PRIOR_PRIME is analogous */
```

THUS, THE GRANULARITY CONCEPT :

- Isn't formally defined
- Doesn't always apply
- Is the same as scale when it does apply

So why so much emphasis in the literature ??? (Or, rather, why so much *confusion* ???) ... From *The Consensus Glossary of Temporal Database Concepts—February 1998 Version*:

“[The] *timestamp granularity* is the size of each chronon in a timestamp interpretation. For example, if the timestamp granularity is one second, then the duration of each chronon in the timestamp interpretation is one second (and vice versa) ... [The] *timestamp interpretation* gives the meaning of each timestamp bit pattern in terms of some time-line clock chronon (or group of chronons).”

POINT AND INTERVAL TYPES REVISITED :

- Type inheritance (background)
- Point types revisited
- NUMERIC point types
- Granularity revisited
- Interval types revisited

BREAK :

Next = interval types revisited

INTERVAL TYPES REVISITED :

To review: An interval type is a *generated* type of the form `INTERVAL_`*T* (where *T* is a point type)

Generic ops:

(interval selectors)

BEGIN and END

PRE, POST, POINT FROM

\in and \ni

COUNT

Allen's operators

interval UNION, INTERSECT, and MINUS

Effect of inheritance model on these concepts?

INHERITANCE EFFECTS :

Following remarks are very slightly simplified ...

If T' = proper subtype of T , `INTERVAL_ T'` is **not** a subtype of `INTERVAL_ T`
... because intervals of type `INTERVAL_ T'` are **not** intervals of type `INTERVAL_ T` !

E.g., $T = \text{INTEGER}$ $T' = \text{EVEN_INTEGER}$... Consider

$i = [2:6]$

$i' = [2:6]$

/ different intervals*

**/*

INHERITANCE EFFECTS :

Following remarks are very slightly simplified ...

If T' = proper subtype of T , $\text{INTERVAL_}T'$ is **not** a subtype of $\text{INTERVAL_}T$
... because intervals of type $\text{INTERVAL_}T'$ are **not** intervals of type $\text{INTERVAL_}T$!

E.g., $T = \text{INTEGER}$ $T' = \text{EVEN_INTEGER}$... Consider

$i = [2:6]$	$i' = [2:6]$	<i>/* different intervals</i>	<i>*/</i>
-------------	--------------	-------------------------------	-----------

$i = [2:2]$	$i' = [2:2]$	<i>/* still different intervals</i>	<i>*/</i>
		<i>/* precisely because they're</i>	<i>*/</i>
		<i>/* of different types</i>	<i>*/</i>

Interval type *can't* be inferred from begin / end point types (in general) ...

Hence, interval selector invocation (e.g.)

INTERVAL ([2:6])

must be a syntax error! ... Instead:

INTERVAL_INTEGER ([2:6])

INTERVAL_EVEN_INTEGER ([2:6])

I.e., interval **selectors** need a “*_T*” qualifier (just like FIRST, LAST, NEXT, etc., do)

We've just seen that if T' is a proper subtype of T ,
 $\text{INTERVAL_}T'$ is **not** a subtype of $\text{INTERVAL_}T$...

In fact, more generally, if $IT = \text{INTERVAL_}T$, there's **no**
 $IT' = \text{INTERVAL_}T'$ that's a proper subtype of IT !!!

Because if there were, every interval $i' = [b':e']$ of type
 IT' must be an interval of type IT as well ... but i' can't
be an interval of type IT ,* because its contained points
are determined by $\text{NEXT_}T'$, which by definition is
distinct from $\text{NEXT_}T$

* *Even if b' and e' happen to be values of type T*

Intervals are *values* and therefore (like all values) carry their type around with them: i.e., values are *typed*—see *The Third Manifesto*

Every interval has **just one** type (“the” type)

```
POST ( INTERVAL_INTEGER ( [2:6] ) )           = 7 (“e+1”)
POST ( INTERVAL_EVEN_INTEGER ( [2:6] ) )       = 8 (“e+1”)
```

POST needs no “_T” qualifier!—argument interval type is known, and hence successor function is known

Likewise for all other interval ops (PRE, BEGIN, END, ...)—only **selectors** need the “_T” qualifier
/ and they’re not really interval ops, as such, anyway */*

BUT THERE'S STILL MORE TO SAY :

Consider distinct interval types $IT1$ and $IT2$, with distinct underlying point types $T1$ and $T2$

$IT1$ isn't a subtype of $IT2$ even if $T1$ is a subtype of $T2$
... but suppose $T1$ *is* a subtype of $T2$... more generally,
suppose $T1$ and $T2$ have a common supertype T —e.g.:

T = integers

..., 1,2,3,4,5,6,7,8,9,10,11,12, ...

$T1$ = multiples of two

..., 2,4,6,8,10,12, ...

$T2$ = multiples of three

..., 3,6,9,12, ...

Comparison $v1 \leq v2$ ($v1$ and $v2$ of types $T1$ and $T2$ respectively)
now valid and might give true ... Implications ???

Allen's operators can be generalized slightly*
(except for "=", whose definition MUST remain unchanged)

- Let $IT = \text{INTERVAL_}T$ correspond to common supertype T
- Let $i1 = [b1:e1]$ be of type $IT1$ and let $j1 = [b1:e1]$ be corresponding but distinct interval of type IT , obtained from $i1$ by *interpolation* (e.g., if $i1$ is 2, 4, 6, then $j1$ is 2, 3, 4, 5, 6) ... Replace $i1$ by $j1$
- Replace $i2$ by $j2$ analogously
- Allen's ops can now be applied directly to $j1$ and $j2$

* *Whether they should be is another matter!*

EXAMPLES AND CONSEQUENCES :

Inclusion: If $i1 = [4:8]$ and $i2 = [3:12]$, $i1 \subset i2$ true ... *Hmmm ...*

EXAMPLES AND CONSEQUENCES :

Inclusion: If $i1 = [4:8]$ and $i2 = [3:12]$, $i1 \subset i2$ true ... *Hmmm ...*

BEFORE and AFTER: If $i1 = [4:6]$ and $i2 = [9:18]$,
 $i1$ BEFORE $i2$ and $i2$ AFTER $i1$ both true

EXAMPLES AND CONSEQUENCES :

Inclusion: If $i1 = [4:8]$ and $i2 = [3:12]$, $i1 \subset i2$ true ... *Hmmm ...*

BEFORE and AFTER: If $i1 = [4:6]$ and $i2 = [9:18]$,
 $i1$ BEFORE $i2$ and $i2$ AFTER $i1$ both true

BEGINS and ENDS: If $i1 = [6:10]$ and $i2 = [6:18]$,
 $i1$ BEGINS $i2$ true /* ENDS is analogous */

EXAMPLES AND CONSEQUENCES :

Inclusion: If $i1 = [4:8]$ and $i2 = [3:12]$, $i1 \subset i2$ true ... *Hmmm ...*

BEFORE and AFTER: If $i1 = [4:6]$ and $i2 = [9:18]$,
 $i1$ BEFORE $i2$ and $i2$ AFTER $i1$ both true

BEGINS and ENDS: If $i1 = [6:10]$ and $i2 = [6:18]$,
 $i1$ BEGINS $i2$ true /* ENDS is analogous */

MEETS: If $i1 = [4:8]$ and $i2 = [9:18]$,
 $i1$ MEETS $i2$ and $i2$ MEETS $i1$ both true

EXAMPLES AND CONSEQUENCES :

Inclusion: If $i1 = [4:8]$ and $i2 = [3:12]$, $i1 \subset i2$ true ... *Hmmm ...*

BEFORE and AFTER: If $i1 = [4:6]$ and $i2 = [9:18]$,
 $i1$ BEFORE $i2$ and $i2$ AFTER $i1$ both true

BEGINS and ENDS: If $i1 = [6:10]$ and $i2 = [6:18]$,
 $i1$ BEGINS $i2$ true /* ENDS is analogous */

MEETS: If $i1 = [4:8]$ and $i2 = [9:18]$,
 $i1$ MEETS $i2$ and $i2$ MEETS $i1$ both true

OVERLAPS: If $i1 = [4:10]$ and $i2 = [9:18]$,
 $i1$ OVERLAPS $i2$ and $i2$ OVERLAPS $i1$ both true ...
Hmmmmmm ...

Interval UNION, INTERSECT, and MINUS: Generalization applies, but result interval is of type *IT*

If $i1 = [4:10]$ and $i2 = [9:18]$...

- $i1 \text{ UNION } i2 = [4:18]$ of type INTERVAL_INTEGER
- $i1 \text{ INTERSECT } i2 = [9:10]$ of type INTERVAL_INTEGER
- $i1 \text{ MINUS } i2 = [4:8]$ of type INTERVAL_INTEGER

PACK and UNPACK: Analogous

Hmmmmm ...

POINT AND INTERVAL TYPES REVISITED :

- Type inheritance (background)
- Point types revisited
- NUMERIC point types
- Granularity revisited
- Interval types revisited

PART III review :

Building on the foundations:

- 12. Database design I : Structure
- 13. Database design II : Keys and related constraints
- 14. Database design III : General constraints
- 15. Queries
- 16. Updates
- 17. Logged time and stated time
- 18. Point and interval types revisited

AGENDA :

I. A review of relational concepts

II. Laying the foundations

III. Building on the foundations

IV. SQL support

V. Appendixes

BREAK :

Next = SQL

PART IV :

SQL support:

- Periods
- Database design
- Queries
- Updates
- System time
- Bitemporal tables
- Summary and assessment

SQL STANDARD :

Limited temporal support was added to the standard as part of the most recent version (“SQL:2011”)

“Periods” represented by “from” and “to” values—
no period *type* as such* */* see next page */*

Periods quite specifically **temporal**

Obvious omissions: No PACK and UNPACK operators ...
No packed form

* *SQL already uses the keyword INTERVAL to mean something else—
actually a duration (e.g., “3 hours”, “90 days”)*

WHY NO INTERVALS AS SUCH ?

From “Temporal Features in SQL:2011,” by Krishna Kulkarni and Jan-Eike Michels (2012) / with permission */ :*

Many treatments of temporal databases introduce a period data type [*sic*], defined as an ordered pair of two datetime values ... SQL:2011 has not taken this route. Adding a new data type to the SQL standard (or to an SQL product) is a costly venture because of the need to support the new data type in the tools and other software packages that form the ecosystem surrounding SQL. For example, if a period type were added to SQL, then it would have to also be added to the stored procedure language, to all database APIs such as JDBC, ODBC, and .NET, as well as to the surrounding technology such as ETL products, replication solutions, and others. There must also be some means of communicating period values to host languages that do not support period as a native data type, such as C or Java.

So “the ecosystem surrounding SQL” apparently has no knowledge of temporal aspects of SQL databases and/or applications ... ??? *What are the implications of this state of affairs ???*

In any case, don't the foregoing arguments apply equally to all user defined types as well ???

Implication: Set of system defined types in SQL is cast in concrete !!! */* but look at SQL:1999 ... */*

Anyway, let's see how our running example would look in SQL ...

FIRST, RECALL :

First and simplest (but still fully temporal) version of suppliers and shipments, with intervals:

```
VAR S_DURING BASE RELATION
  { SNO SNO , DURING INTERVAL_DATE }
  USING ( DURING ) : KEY { SNO , DURING }
```

```
VAR SP_DURING BASE RELATION
  { SNO SNO , PNO PNO , DURING INTERVAL_DATE }
  USING ( DURING ) : KEY { SNO, PNO , DURING }
  USING ( DURING ) : FOREIGN KEY { SNO , DURING }
  REFERENCES S_DURING ;
```

Note: Throughout what follows, “table” means a base table specifically, unless the context demands otherwise

SQL ANALOGS :

Suppliers:

```
CREATE TABLE S_FROM_TO  
  ( SNO      SNO NOT NULL ,  
    DFROM DATE NOT NULL ,  
    DTO      DATE NOT NULL ,  
    PERIOD FOR DPERIOD ( DFROM , DTO ) ,  
    UNIQUE ( SNO , DPERIOD WITHOUT OVERLAPS ) ;
```

“PERIOD FOR” :

Implies “from” < “to” */* periods always closed:open */*

Columns must be both of **type DATE**

/ assume this case until further notice */*

or both of **same TIMESTAMP type**

DPERIOD denotes “**application time**”

/ SQL term for valid or stated time */*

No table can have more than one application time period

DPERIOD is **not** a column! Period name can appear in just one context where a column reference might have been expected */* see later ... can also appear in PORTION ... again, see later */*

TYPE DATE (SQL) :

“first” = DATE '0001-01-01'

“last” = DATE '9999-12-31'

/ not in any period! */*

“next” = DFROM + INTERVAL '1' DAY */* for example */*

“prior” = DFROM – INTERVAL '1' DAY

By the way ...

INTERVAL '1' DAY here is an SQL *interval literal*

“WITHOUT OVERLAPS” :

Prevents **overlapping** and **contradiction**
but not *circumlocution* ...

... which turns out to be a rather serious problem!
See the textbook for specifics

/ Hint: Try stating the predicate! */*

Note: WITHOUT OVERLAPS isn't quite equivalent
to either PACKED ON or WHEN / THEN

Note that the specified “key” is definitely a proper superkey

SQL ANALOGS (cont.) :

Shipments:

```
CREATE TABLE SP_FROM_TO
( SNO      SNO  NOT NULL ,
  PNO      PNO  NOT NULL ,
  DFROM    DATE NOT NULL ,
  DTO      DATE NOT NULL ,
  PERIOD FOR DPERIOD ( DFROM , DTO ) ,
  UNIQUE ( SNO , PNO , DPERIOD WITHOUT OVERLAPS ) ,
  FOREIGN KEY ( SNO , PERIOD DPERIOD ) REFERENCES
    S_FROM_TO ( SNO , PERIOD DPERIOD ) ) ;
```

Every SNO-DPERIOD value in “unpacked form of SP_FROM_TO on DPERIOD” must appear in “unpacked form of S_FROM_TO on DPERIOD”

SAMPLE VALUES :

S_FROM_TO

SNO	DFROM	DTO
S1	d04	d99
S2	d02	d05
S2	d07	d99
S3	d03	d99
S4	d04	d99
S5	d02	d99

SP_FROM_TO

SNO	PNO	DFROM	DTO
S1	P1	d04	d99
S1	P2	d05	d99
S1	P3	d09	d99
S1	P4	d05	d99
S1	P5	d04	d99
S1	P6	d06	d99
S2	P1	d02	d05
S2	P1	d08	d99
S2	P2	d03	d04
S2	P2	d09	d99
S3	P2	d08	d99
S4	P2	d06	d10
S4	P4	d04	d09
S4	P5	d05	d99

“PERIOD SELECTORS” :

No *period type*, so no period *variables*, so no period variable *references* ... more generally, no period valued *expressions*

However, there *is* a kind of “period selector” (with “period literals” as a special case)

But only context where they can be used is as an operand to one of Allen’s operators, as a “period predicand” in a “period predicate”

/ typically in a WHERE clause */*

Syntax: **PERIOD (*f* , *t*)** */* corresp. to [f:t) or [f:t-1] */*
 / ... don’t get confused! */*

FOR EXAMPLE :

```
SELECT DISTINCT SNO  
FROM S_FROM_TO  
WHERE PERIOD ( DFROM , DTO ) OVERLAPS  
PERIOD ( DATE '2012-01-01' , DATE '2013-01-01' )
```

Two “period predicands” here, looking something like hypothetical “period selector” invocations ... Second looks something like a hypothetical “period literal”

FOR EXAMPLE :

```
SELECT DISTINCT SNO
FROM   S_FROM_TO
WHERE  PERIOD ( DFROM , DTO ) OVERLAPS
        PERIOD ( DATE '2012-01-01' , DATE '2013-01-01' )
```

Two “period predicands” here, looking something like hypothetical “period selector” invocations ... Second looks something like a hypothetical “period literal”

If a period predicand denotes a period explicitly defined to be part of some (base) table, **period name** can be used instead:

```
SELECT DISTINCT SNO
FROM   S_FROM_TO
WHERE  DPERIOD /* like a hypothetical column ref */ OVERLAPS
        PERIOD ( DATE '2012-01-01' , DATE '2013-01-01' )
```

PERIOD OPERATORS :

Assume “period predicand” $p = \text{PERIOD}(f, t) \dots$

<i>Operator (not SQL!)</i>	<i>SQL analog</i>
BEGIN (p) END (p) PRE (p) POST(p) POINT FROM p $x \in p$ $p \ni x$	f $t - \text{INTERVAL '1' DAY}$ $f - \text{INTERVAL '1' DAY}$ t <i>no direct support</i> <i>no direct support</i> $p \text{ CONTAINS } x$

Note that $t \in p$ is false!

ALLEN'S OPERATORS :

Operator (not SQL!)	SQL analog
$p1 = p2$	$p1$ EQUALS $p2$
$p1 \supseteq p2$	$p1$ CONTAINS $p2$
$p1 \supset p2$	no direct support
$p1 \subseteq p2$	no direct support
$p1 \subset p2$	no direct support
$p1$ BEFORE $p2$	$p1$ PRECEDES $p2$
$p1$ AFTER $p2$	$p1$ SUCCEEDS $p2$
$p1$ MEETS $p2$	$p1$ IMMEDIATELY PRECEDES $p2$ OR $p1$ IMMEDIATELY SUCCEEDS $p2$
$p1$ MERGES $p2$	no direct support
$p1$ BEGINS $p2$	no direct support
$p1$ ENDS $p2$	no direct support

OTHER OPERATORS :

COUNT :

CAST (($t - f$) AS INTEGER)

OTHER OPERATORS :

COUNT :

`CAST ((t – f) AS INTEGER)`

UNION, INTERSECT, MINUS:

No direct support

OTHER OPERATORS :

COUNT :

CAST ((t – f) AS INTEGER)

UNION, INTERSECT, MINUS:

No direct support

PACK, UNPACK, U_ operators, U_ updates:

*No direct support ... **Very unfortunately!***

PART IV cont. :

SQL support:

- Periods
- Database design
- Queries
- Updates
- System time
- Bitemporal tables
- Summary and assessment

DB DESIGN :

Bring supplier status history back into the picture:

```
CREATE TABLE S_FROM_TO
( SNO      SNO  NOT NULL ,
  DFROM    DATE NOT NULL ,
  DTO      DATE NOT NULL ,
  PERIOD FOR DPERIOD ( DFROM , DTO ) ,
  UNIQUE ( SNO , DPERIOD WITHOUT OVERLAPS ) ,
  FOREIGN KEY ( SNO , PERIOD DPERIOD ) REFERENCES
    S_STATUS_FROM_TO ( SNO , PERIOD DPERIOD ) );
```

By the way ... note the tiny syntactic inconsistency between key and foreign key specifications

DB DESIGN (cont.) :

```
CREATE TABLE S_STATUS_FROM_TO
( SNO      SNO      NOT NULL ,
  STATUS INTEGER NOT NULL ,
  DFROM DATE      NOT NULL ,
  DTO      DATE      NOT NULL ,
  PERIOD FOR DPERIOD ( DFROM , DTO ) ,
  UNIQUE ( SNO , DPERIOD WITHOUT OVERLAPS ) ,
  FOREIGN KEY ( SNO , PERIOD DPERIOD ) REFERENCES
      S_FROM_TO ( SNO , PERIOD DPERIOD ) ) ;
```

DB DESIGN (cont.) :

```
CREATE TABLE SP_FROM_TO
( SNO      SNO  NOT NULL ,
  PNO      PNO  NOT NULL ,
  DFROM    DATE NOT NULL ,
  DTO      DATE NOT NULL ,
  PERIOD FOR DPERIOD ( DFROM , DTO ) ,
  UNIQUE ( SNO , DPERIOD WITHOUT OVERLAPS ) ,
  FOREIGN KEY ( SNO , PERIOD DPERIOD ) REFERENCES
    S_FROM_TO ( SNO , PERIOD DPERIOD ) ) ;
```

SQL more or less assumes a “historical tables only” design
(analogous to during relvars only)

SAMPLE VALUES :

S_FROM_TO

SNO	DFROM	DTO
S2	d02	d05
S6	d03	d06
S7	d03	d99

S_STATUS_FROM_TO

SNO	STATUS	DFROM	DTO
S2	5	d02	d03
S2	10	d03	d05
S6	5	d03	d05
S6	7	d05	d06
S7	15	d03	d09
S7	20	d09	d99

SP_FROM_TO

SNO	PNO	DFROM	DTO
S2	P1	d02	d05
S2	P2	d03	d04
S2	P5	d03	d05
S6	P3	d03	d06
S6	P4	d04	d05
S6	P5	d04	d06
S7	P1	d03	d05
S7	P1	d06	d08
S7	P1	d09	d99

“APPLICATION TIME” :

- SQL term for “valid time,” recall
- **At most one** application time period per table, recall
- *Implying that those tables had better be in 6NF!**—for otherwise “the timestamp will timestamp too much”
- ***And implying also that, e.g., join loses “valid time” information ...***
I.e., periods don’t “carry through” operational expressions

* *Speaking somewhat loosely ...
sadly, 6NF is a slightly muddled concept in the temporal SQL context*

RE “CARRYING THROUGH” :

Suppose table T has a period P ... Then, e.g.,

```
SELECT *  
FROM    $T$   
WHERE   $P$  CONTAINS  $d$ 
```

is legal, but

```
SELECT *  
FROM   ( SELECT * FROM  $T$  ) AS POINTLESS  
WHERE   $P$  CONTAINS  $d$ 
```

isn't!

Implications for, e.g., view definition?

PART IV cont. :

SQL support:

- Periods
- Database design
- Queries
- Updates
- System time
- Bitemporal tables
- Summary and assessment

BREAK :

Next = SQL queries and updates

QUERY Q1 :

Get the status of S1 on day *dn*, together with the associated period

```
SELECT STATUS , DFROM , DTO
FROM   S_STATUS_FROM_TO
WHERE  SNO = SNO ( 'S1' )
AND    DPERIOD CONTAINS dn
```

“Period reference” OK as “period predicand” (i.e., in boolean expression in WHERE clause) but **not** as a SELECT clause element

Result table has no period, as such, at all!

QUERY Q2 :

Get pairs of SNOs such that the indicated suppliers were assigned their current status on the same day

```
WITH t1 AS ( SELECT SNO , DFROM
              FROM   S_STATUS_FROM_TO
              WHERE DPERIOD CONTAINS CURRENT_DATE ) ,
      t2 AS ( SELECT SNO AS XNO , DFROM
              FROM   t1 ) ,
      t3 AS ( SELECT SNO AS YNO , DFROM
              FROM   t1 ) ,
      t4 AS ( SELECT XNO , YNO
              FROM t2 NATURAL JOIN t3 )
SELECT XNO , YNO
FROM   t4
WHERE  XNO < YNO
```

QUERY Q3 :

Get SNOs for suppliers able to supply both P1 and P2 at the same time

```
SELECT DISTINCT t1.SNO
FROM   SP_FROM_TO AS t1 , SP_FROM_TO AS t2
WHERE  t1.SNO = t2.SNO
AND    t1.PNO = PNO ( 'P1' )
AND    t2.PNO = PNO ( 'P2' )
AND    t1.DPERIOD OVERLAPS t2.DPERIOD
```

SQL has no “U_join” (etc.), so we effectively have to spell out the definition of that operator ... though at least we can use Allen’s OVERLAPS operator

QUERIES Q4 & Q5 :

Get SNOs for suppliers never able to supply both P1 and P2 at the same time

/ not very interesting */*

Get SNOs for suppliers who, while under some specific contract, changed their status since they most recently became able to supply some part under that contract

/ complicated!—see the textbook */*

QUERY Q6 :

Get periods during which at least one supplier was under contract

```
SELECT DISTINCT DFROM , DTO  
FROM    S_FROM_TO
```

Result not properly packed, in general, even if S_FROM_TO is kept in packed form

QUERY Q7 :

Let BUSY = result of Q6; use BUSY to get periods during which no supplier was under contract at all

Let ETERNITY be result of:

```
SELECT TEMP.*  
FROM   ( VALUES ( DATE '0001-01-01' , DATE '9999-12-31' )  
          AS TEMP ( DFROM , DTO )
```

Then unpack ETERNITY, unpack BUSY, form the difference (in that order), and pack the result

/ details left as an exercise */*

QUERIES Q8 & Q9 :

Get SNOs for suppliers currently under contract who also had an earlier contract

/ not very interesting */*

Get (SNO, pf , pt , df , dt) quintuples such that supplier SNO was able to supply all parts with PNOs in the range [pt , pf) throughout the time period [df , dt)

/ problems: 1. nontemporal “period” 2. result has two distinct periods */*

Remember **Query A** ???—Get SNO-DURING pairs such that DURING designates a maximal interval during which supplier SNO was able to supply at least one part

```
USING ( DURING ) : SP_DURING { SNO , DURING }  
/* “U_projection” */
```

*Can we do this query in SQL ??? Ditto **Query B** ???*

Well ... “U_ops” are defined in terms of PACK and UNPACK, which are defined in terms of EXPAND and COLLAPSE ...

“EXPAND” DEFINITION :

Let X be a set of intervals all of the same type, IT say, and let i and j be intervals of type IT . Let $i = [b:e]$. Then EXPAND (X) is defined as:

$$\{ i : b = e \text{ AND EXISTS } j \in X (b \in j) \}$$

“COLLAPSE” DEFINITION :

Let X be a set of intervals all of the same type IT , and let the underlying point type be T . Let $i, i1, i2$, and j be intervals of type IT , and let p be a point of type T . Let $i = [b:e]$, $i1 = [b1:e1]$, and $i2 = [b2:e2]$. Then COLLAPSE (X) is defined as:

```
{ i: FORALL p ∈ i ( EXISTS j ∈ X ( p ∈ j ) )
  AND
  EXISTS i1 ∈ X ( EXISTS i2 ∈ X
    ( b = b1 AND e = e2 AND b1 ≤ b2 AND e1 ≤ e2
      AND
      IF b2 ≠ FIRST_T ( ) THEN
        IF e1 < PRE ( i2 ) THEN
          FORALL p ∈ ( e1 : b2 )
            ( EXISTS j ∈ X ( p ∈ j ) ) END IF END IF
        AND
        FORALL p ∈ i
          ( IF p ≠ FIRST_T ( ) THEN
            NOT EXISTS j ∈ X ( PRE ( i ) ∈ j ) END IF
          AND
            IF p ≠ LAST_T ( ) THEN
              NOT EXISTS j ∈ X ( POST ( i ) ∈ j ) END IF ) ) )
    )
  }
```

PART IV cont. :

SQL support:

- Periods
- Database design
- Queries
- Updates
- System time
- Bitemporal tables
- Summary and assessment

UPDATE U12 :

Add proposition(s) to show supplier S9 has just been placed under contract, with status 15

```
INSERT INTO S_FROM_TO ( SNO , DFROM , DTO )  
VALUES ( SNO ( 'S9' ) , CURRENT_DATE , DATE '9999-12-31' ) ;
```

```
INSERT INTO S_STATUS_FROM_TO ( SNO , STATUS , DFROM , DTO )  
VALUES ( SNO ( 'S9' ) , 15 , CURRENT_DATE , DATE '9999-12-31' ) ;
```

Two problems: 1. Referential cycle

/ so need transactions and COMMIT time checking */*

2. CURRENT_DATE

/ so code defensively */*

UPDATE U13 :

Remove proposition(s) showing supplier S7 as under contract

```
DELETE FROM SP_FROM_TO  
WHERE SNO = SNO ( 'S7' ) ;
```

```
DELETE FROM S_STATUS_FROM_TO  
WHERE SNO = SNO ( 'S7' ) ;
```

```
DELETE FROM S_FROM_TO  
WHERE SNO = SNO ( 'S7' ) ;
```

All part of same transaction

UPDATE U14 :

Supplier S7's contract has just been terminated

```
SET X = CURRENT_DATE ;  
SET Y = X + INTERVAL '1' DAY ;
```

```
UPDATE S_FROM_TO SET DTO = Y  
WHERE SNO = SNO ( 'S7' ) AND DPERIOD CONTAINS X ;
```

```
UPDATE S_STATUS_FROM_TO SET DTO = Y  
WHERE SNO = SNO ( 'S7' ) AND DPERIOD CONTAINS X ;
```

```
UPDATE SP_FROM_TO SET DTO = Y  
WHERE SNO = SNO ( 'S7' ) AND DPERIOD CONTAINS X ;
```

All part of same transaction

UPDATE U4 :

Add proposition “S2 was able to supply P4 on day 2”

```
INSERT INTO SP_FROM_TO ( SNO , PNO , DFROM , DTO )  
VALUE ( SNO ( 'S2' ) , PNO ( 'P4' ) , d02 , d03 ) ;
```

UPDATE U4 :

Add proposition “S2 was able to supply P4 on day 2”

```
INSERT INTO SP_FROM_TO ( SNO , PNO , DFROM , DTO )  
VALUE ( SNO ( 'S2' ) , PNO ( 'P4' ) , d02 , d03 ) ;
```

But suppose it was part P5, not part P4 ...

Result not properly packed! If accepted:

SNO	PNO	DFROM	DTO
S2	P5	d02	d03

SNO	PNO	DFROM	DTO
S2	P5	d03	d05

But it *will* be accepted, in SQL ... **SQL has no “U_INSERT”**

UPDATE U5 :

Remove proposition “S6 was able to supply P3 from day 3 to day 5”

```
DELETE FROM SP_FROM_TO  
WHERE SNO = SNO ( 'S6' )  
AND     PNO = PNO ( 'P3' )  
AND     DPERIOD EQUALS PERIOD ( d03 , d06 ) ; /* OK */
```


UPDATE U5 modified :

Remove proposition “S7 was able to supply P1 from day 4 to day 11”

```
DELETE FROM SP_FROM_TO
WHERE SNO = SNO ( 'S7' )
AND     PNO = PNO ( 'P1' )
AND     DPERIOD EQUALS PERIOD ( d04 , d12 ) ; /* no effect */
```

By contrast, following does have the desired effect:

```
DELETE FROM SP_FROM_TO
FOR     PORTION OF DPERIOD FROM d04 TO d12
WHERE SNO = SNO ( 'S7' ) ↑—must be application period name
AND     PNO = PNO ( 'P1' ) ;
```

/ for detailed semantics see textbook */*

S_FROM_TO

SNO	DFROM	DTO
S2	<i>d02</i>	<i>d05</i>
S6	<i>d03</i>	<i>d06</i>
S7	<i>d03</i>	<i>d99</i>

S_STATUS_FROM_TO

SNO	STATUS	DFROM	DTO
S2	5	<i>d02</i>	<i>d03</i>
S2	10	<i>d03</i>	<i>d05</i>
S6	5	<i>d03</i>	<i>d05</i>
S6	7	<i>d05</i>	<i>d06</i>
S7	15	<i>d03</i>	<i>d09</i>
S7	20	<i>d09</i>	<i>d99</i>

SP_FROM_TO

SNO	PNO	DFROM	DTO
S2	P1	<i>d02</i>	<i>d05</i>
S2	P2	<i>d03</i>	<i>d04</i>
S2	P5	<i>d03</i>	<i>d05</i>
S6	P3	<i>d03</i>	<i>d06</i>
S6	P4	<i>d04</i>	<i>d05</i>
S6	P5	<i>d04</i>	<i>d06</i>
S7	P1	<i>d03</i>	<i>d04</i>
S7	P1	<i>d12</i>	<i>d99</i>

UPDATE U5 cont. :

```
DELETE FROM SP_FROM_TO  
FOR      PORTION OF DPERIOD FROM d04 TO d12  
WHERE SNO = SNO ( 'S7' )  
AND      PNO = PNO ( 'P1' ) ;
```

/ PORTION can be used with UPDATE too */*

Suppose foregoing DELETE ... PORTION is followed by:

```
INSERT INTO SP_FROM_TO ( SNO , PNO , DFROM , DTO )  
VALUES ( SNO ( 'S7' ) , PNO ( 'P1' ) , d04 , d12 ) ;
```

Result:

S_FROM_TO

SNO	DFROM	DTO
S2	d02	d05
S6	d03	d06
S7	d03	d99

S_STATUS_FROM_TO

SNO	STATUS	DFROM	DTO
S2	5	d02	d03
S2	10	d03	d05
S6	5	d03	d05
S6	7	d05	d06
S7	15	d03	d09
S7	20	d09	d99

SP_FROM_TO

SNO	PNO	DFROM	DTO
S2	P1	d02	d05
S2	P2	d03	d04
S2	P5	d03	d05
S6	P3	d03	d06
S6	P4	d04	d05
S6	P5	d04	d06
S7	P1	d03	d04
S7	P1	d04	d12
S7	P1	d12	d99

PART IV cont. :

SQL support:

- Periods
- Database design
- Queries
- Updates
- System time
- Bitemporal tables
- Summary and assessment

BREAK :

Next = SQL “system time”

“SYSTEM TIME” :

Any base table can have a *system time period* (at most one).*

E.g. */* note no application time period, for simplicity */*:

```
CREATE TABLE XS_STATUS_FROM_TO
( SNO      SNO NOT NULL ,
  STATUS INTEGER NOT NULL ,
  XFROM  TIMESTAMP(12)    /* column name arbitrary */
      GENERATED ALWAYS AS ROW START NOT NULL ,
  XTO     TIMESTAMP(12)    /* column name arbitrary */
      GENERATED ALWAYS AS ROW END NOT NULL ,
  PERIOD FOR SYSTEM_TIME ( XFROM , XTO ) ,
  UNIQUE ( SNO ) ,
  FOREIGN KEY ( SNO ) REFERENCES XS_FROM_TO ( SNO ) )
WITH SYSTEM VERSIONING ;
```

* *But the table had better be in 6NF (modulo earlier remarks on 6NF and SQL)*

SNO and STATUS are the only columns the user can update directly

PERIOD FOR SYSTEM_TIME: Name SYSTEM_TIME fixed by standard ... Implies “from” < “to” */* periods always closed:open */* ... Columns must be both of type DATE */* unlikely */* or both of same TIMESTAMP type

GENERATED ALWAYS ... : *Must be specified /* see later */*

WITH SYSTEM VERSIONING: *Must be specified if table is to be auditable /* “system versioned table” */ ... Else ... ???*

UNIQUE and FOREIGN KEY: See later

UPDATES :

```
INSERT INTO XS_STATUS_FROM_TO ( SNO , STATUS )  
VALUES ( SNO ( 'S1' ) , 20 ) ; /* at time t02 “by system clock” */
```

SNO	STATUS	XFROM	XTO
S1	20	<i>t02</i>	<i>t99</i>

/ by the way, t99 is a lie! */*

UPDATES :

```
INSERT INTO XS_STATUS_FROM_TO ( SNO , STATUS )  
VALUES ( SNO ( 'S1' ) , 20 ) ; /* at time t02 “by system clock” */
```

SNO	STATUS	XFROM	XTO
S1	20	t02	t99

```
UPDATE XS_STATUS_FROM_TO SET STATUS = 25  
WHERE SNO = SNO ( 'S1' ) ; /* at time t06 “by system clock” */
```

SNO	STATUS	XFROM	XTO
S1	25	t06	t99
S1	20	t02	t06

UPDATES cont. :

DELETE FROM XS_STATUS_FROM_TO
WHERE SNO = SNO ('S1') ; */* at time t45 “by system clock” */*

SNO	STATUS	XFROM	XTO
S1	25	t06	t45
S1	20	t02	t06

Note: “By system clock” is NOT what the standard says ...
See the textbook for the specifics

KEYS AND FOREIGN KEYS :

Current row = row with XTO “the end of time”

Historical row = row that's not a current row

Only current rows can be directly updated, and only columns other than XFROM and XTO

/ so FOR PORTION OF on system time not allowed */*

UNIQUE and FOREIGN KEY specifications apply to current rows only

System versioned tables are really a kind of combination of two separate tables ...

QUERIES :

By default, queries apply to current rows only ... E.g., given:

SNO	STATUS	XFROM	XTO
S1	25	<i>t06</i>	<i>t99</i>
S1	20	<i>t02</i>	<i>t06</i>

```
SELECT STATUS
FROM   XS_STATUS_FROM_TO
WHERE  SNO = SNO ( 'S1' )
```

STATUS
25

QUERIES (cont.) :

By contrast:

SNO	STATUS	XFROM	XTO
S1	25	t06	t99
S1	20	t02	t06

```
SELECT STATUS
FROM XS_STATUS_FROM_TO FOR SYSTEM_TIME AS OF t04
WHERE SNO = SNO ( 'S1' )
```

STATUS
20

applies to system time;
qualifies a table ref in a FROM clause
(contrast FOR PORTION OF, which
applies to application time and
qualifies DELETE or UPDATE as such)

```
SELECT * FROM S
      FOR SYSTEM_TIME AS OF timestamp-1
/* rows where [XFROM:XTO) contains timestamp-1 */
```

```
SELECT * FROM S
      FOR SYSTEM_TIME FROM timestamp-1
                        TO      timestamp-2
/* rows where [XFROM:XTO) overlaps [timestamp-1:timestamp-2) */
```

```
SELECT * FROM S
      FOR SYSTEM_TIME BETWEEN timestamp-1
                        AND      timestamp-2
/* rows where [XFROM:XTO) overlaps [timestamp-1:timestamp-2] */
```

A CHALLENGE :

Write an SQL query for:

When did the DB say some supplier could supply all of the parts supplied?

PART IV cont. :

SQL support:

- Periods
- Database design
- Queries
- Updates
- System time
- **Bitemporal tables** */* not an official term */*
- Summary and assessment

E.G. :

```
CREATE TABLE BS_FROM_TO
( SNO      SNO NOT NULL ,
  DFROM    DATE NOT NULL ,
  DTO      DATE NOT NULL ,
  PERIOD FOR DPERIOD ( DFROM , DTO ) ,
  UNIQUE ( SNO , DPERIOD WITHOUT OVERLAPS )
  FOREIGN KEY ( SNO , PERIOD DPERIOD ) REFERENCES
      BS_STATUS_FROM_TO ( SNO , PERIOD DPERIOD ) )
XFROM    TIMESTAMP(12)
          GENERATED ALWAYS AS ROW START NOT NULL ,
XTO      TIMESTAMP(12)
          GENERATED ALWAYS AS ROW END NOT NULL ,
  PERIOD FOR SYSTEM_TIME ( XFROM , XTO ) )
WITH SYSTEM VERSIONING ;
```

```
CREATE TABLE BS_STATUS_FROM_TO
( SNO      SNO      NOT NULL ,
  STATUS INTEGER NOT NULL ,
  DFROM DATE      NOT NULL ,
  DTO      DATE      NOT NULL ,
  PERIOD FOR DPERIOD ( DFROM , DTO ) ,
  UNIQUE ( SNO , DPERIOD WITHOUT OVERLAPS )
  FOREIGN KEY ( SNO , PERIOD DPERIOD ) REFERENCES
      BS_FROM_TO ( SNO , PERIOD DPERIOD ) )
XFROM TIMESTAMP(12)
      GENERATED ALWAYS AS ROW START NOT NULL ,
XTO      TIMESTAMP(12)
      GENERATED ALWAYS AS ROW END NOT NULL ,
  PERIOD FOR SYSTEM_TIME ( XFROM , XTO ) )
WITH SYSTEM VERSIONING ;
```

UPDATES :

```
INSERT INTO BS_STATUS_FROM_TO ( SNO , STATUS , DFROM , DTO )  
VALUES ( SNO ( 'S2' ) , 5 , d02 , d05 ) ; /* at time t11 “by system clock” */
```

SNO	STATUS	DFROM	DTO	XFROM	XTO
S2	5	d02	d05	t11	t99

UPDATES :

```
INSERT INTO BS_STATUS_FROM_TO ( SNO , STATUS , DFROM , DTO )  
VALUES ( SNO ( 'S2' ) , 5 , d02 , d05 ) ; /* at time t11 “by system clock” */
```

SNO	STATUS	DFROM	DTO	XFROM	XTO
S2	5	d02	d05	t11	t99

```
UPDATE BS_STATUS_FROM_TO  
FOR      PORTION OF DPERIOD FROM d03 TO d04  
SET      STATUS = 10  
WHERE    SNO = SNO ( 'S2' ) ; /* at time t22 “by system clock” */
```

SNO	STATUS	DFROM	DTO	XFROM	XTO
S2	5	d02	d05	t11	t22
S2	5	d02	d03	t22	t99
S2	10	d03	d04	t22	t99
S2	5	d04	d05	t22	t99

UPDATES (cont.) :

DELETE FROM BS_STATUS_FROM_TO

FOR PORTION OF DPERIOD FROM *d03* TO *d05*

WHERE SNO = SNO ('S2') ; */* at time t33 “by system clock” */*

SNO	STATUS	DFROM	DTO	XFROM	XTO
S2	5	<i>d02</i>	<i>d05</i>	<i>t11</i>	<i>t22</i>
S2	5	<i>d02</i>	<i>d03</i>	<i>t22</i>	<i>t99</i>
S2	10	<i>d03</i>	<i>d04</i>	<i>t22</i>	<i>t33</i>
S2	5	<i>d04</i>	<i>d05</i>	<i>t22</i>	<i>t33</i>

UPDATES (cont.) :

DELETE

FROM BS_STATUS_FROM_TO WHERE SNO = SNO ('S2') ;

/ at time t44 “by system clock” */*

SNO	STATUS	DFROM	DTO	XFROM	XTO
S2	5	d02	d05	t11	t22
S2	5	d02	d03	t22	t44
S2	10	d03	d04	t22	t33
S2	5	d04	d05	t22	t33

PART IV cont. :

SQL support:

- Periods
- Database design
- Queries
- Updates
- System time
- Bitemporal tables
- Summary and assessment

BREAK :

Next = SQL summary & assessment

DB DESIGN :

SQL implicitly opts for “historical tables only”

1. Hence ad hoc support for “the moving point *now*” ...
2. ... and tables probably need to be in 6NF /* *as it were* */

SQL quite rightly fails to legislate on 2. ... but tacitly does seem to assume DB design will be done by simply adding timestamp columns ... e.g. /* *from Kulkarni & Michels* */:

The choice of returning current ... rows as the default [*i.e., from a query on a system-versioned table*] ... helps with ... database migration in that applications running on non system-versioned tables would continue to work and produce the same results when those tables are converted to system-versioned tables.

DB DESIGN (cont.) :

1. WITHOUT OVERLAPS prevents redundancy and contradiction but *not* circumlocution
2. UNIQUE and FOREIGN KEY on tables with application time period are *almost* equivalent to analogous U_key and foreign U_key specs

But keeping tables in packed form is the user's responsibility, and analogs of U_INSERT etc. are *not* provided

No SINCE_FOR or HISTORY_IN or COMBINED_IN options

FEATURE LIST :

Our approach

design choice:

since relvars only
during relvars only
combined design
6NF

SINCE_FOR / HISTORY_IN

COMBINED_IN (“automatic views”) no

PACKED ON

WHEN / THEN

“NOW” **not** supported

SQL:2011

/ do it yourself */*

/ this case tacitly assumed */*

/ do it yourself */*

/ do it yourself */*

no

no

no */* but WITHOUT OVERLAPS
prevents redundancy */*

no */* but WITHOUT OVERLAPS
prevents contradiction */*

yes

FEATURE LIST (cont.) :

Our approach

intervals (values as such)

interval type generator

no style forced by system

“from” < “to” (automatic)

nontemporal intervals

point type any ordinal type

SQL:2011

periods (but value pairs)

NO /* periods not “first class”
—can’t be used orthogonally */

closed:open (well, usually)

yes

no

DATE & TIMESTAMP types only

FEATURE LIST (cont.) :

Our approach

successor function (NEXT_*T*)

PRIOR_*T*

FIRST_*T*

LAST_*T*

SQL:2011

yes /* e.g.,
 $d + \text{INTERVAL '1' DAY}$ */

yes /* e.g.,
 $d - \text{INTERVAL '1' DAY}$ */

yes /* e.g., DATE '0001-01-01' ...
 but user has to know value */

yes /* e.g., DATE '9999-12-31' ...
 *but no period can ever contain
 this value* */

FEATURE LIST (cont.) :

Our approach

interval attribute reference

interval selector invocation
(including interval literals)

BEGIN

END

PRE

SQL:2011

period name /* *but only as Allen operand or in PORTION */*

period predicand /* *but only as Allen operand */*

yes /* e.g., DFROM */

yes /* e.g.,
DTO – INTERVAL '1' DAY */

yes /* e.g.,
DFROM – INTERVAL '1' DAY */

FEATURE LIST (cont.) :

Our approach

POST

POINT FROM

∈

∃

= (Allen)

⊆

⊂

SQL:2011

yes /* e.g., DTO */

no

no

CONTAINS

EQUALS

no

no

FEATURE LIST (cont.) :

Our approach

⊇

⊃

BEFORE

AFTER

OVERLAPS

MEETS

SQL:2011

CONTAINS /* note overloading */

no

PRECEDES

SUCCEEDS

OVERLAPS

IMMEDIATELY PRECEDES OR
IMMEDIATELY SUCCEEDS

FEATURE LIST (cont.) :

Our approach

SQL:2011

MERGES

no

BEGINS

no

ENDS

no

COUNT

yes /* e.g., DTO – DFROM ...
*but must cast to INTEGER */*

UNION

no

INTERSECT

no

FEATURE LIST (cont.) :

Our approach

SQL:2011

MINUS

no

2+ interval attributes

no /* except for
“bitemporal” tables */

intervals “carry through”

no

EXPAND and COLLAPSE

no

PACK / UNPACK on 1 attribute

no

PACK / UNPACK on 2+ attributes

no

FEATURE LIST (cont.) :

Our approach

SQL:2011

“auto” packing of query results

no

updating beliefs re past

yes

U_ operators (U_ join etc.)

no

U_ comparisons

no

interval-only relations

yes

“Queries A and B”

/ intolerably clumsy */*

FEATURE LIST (cont.) :

Our approach

“nine requirements”

multiple assignment

U_ updates

PORTION (1 interval)

PORTION (2+ intervals)

SQL:2011

prevent redundancy: yes
prevent contradiction: yes
prevent circumlocution: no
handle denseness: yes /* *but
requires deferred checking */*

no

no

yes

no

FEATURE LIST (cont.) :

Our approach

stated time

logged time

LOGGED_TIMES_IN

“auto” packing of logged time
query results

SQL:2011

application time / base
tables only */*

system time / base tables
only, explicitly part of table */*

no

no

FEATURE LIST (cont.) :

Our approach

more than one successor function
/* e.g., DDATE vs. MDATE */

numeric point types
/* i.e., NUMERIC(p,q) */

cyclic point types

SQL:2011

yes /* e.g.,
 $d + \text{INTERVAL '1' DAY}$ vs.
 $d + \text{INTERVAL '1' MONTH}$
... *but just one point type* */

no

no

BRIEF ANALYSIS :

Our approach

general purpose

based on widely applicable
and widely useful *interval*
abstraction; ***relational!***

time concept:

only in (a) design specifics
& (b) LOGGED_TIMES_IN

SQL:2011

specific to time as such

based on time-specific &
ad hoc notion of column
pair “period” and special
case syntax

time concept:

all pervasive

violates *Interchangeability*
Principle, Assignment
Principle, Information
Principle (?)

numerous ad hoc limitations

DB2 SUPPORT :

- SQL:2011 support is “based on” DB2 support (?) ... Some syntactic differences but on the whole fairly similar
- **Caveat:** DB2’s support for *dates and timestamps*—and especially for *datetime arithmetic*—is significantly different from that of the standard
- **Terminology:** System time vs. **business** time

PART IV review :

SQL support:

- Periods
- Database design
- Queries
- Updates
- System time
- Bitemporal tables
- Summary and assessment

AGENDA :

I. A review of relational concepts

II. Laying the foundations

III. Building on the foundations

IV. SQL support

V. Appendixes

BREAK :

Next = appendixes

APPENDIXES :

A. Cyclic point types

B. Is ordinality necessary?

C. Generalizing PACK and UNPACK

D. A **Tutorial D** grammar

E. Implementation considerations

F. References and bibliography

CYCLIC POINT TYPES :

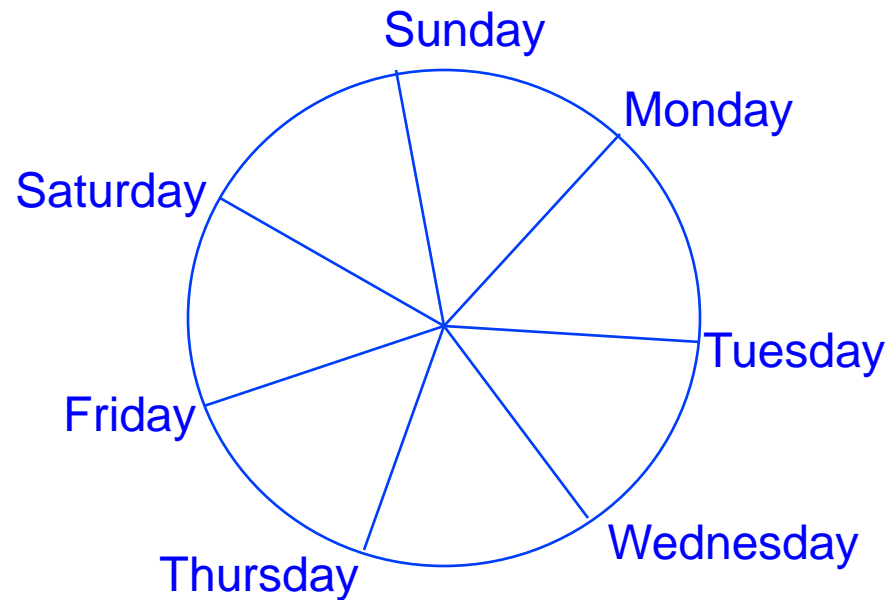
E.g., time of day, day of week ... *different kettle of fish!*

Raise the possibility of *wraparound intervals*
(e.g., [2300:0700], [Thu:Tue], etc.)

Share with modular arithmetic the property that available values are like points on the circumference of a circle ...

No “first” or “last”, and “next” and “prior” never fail

CYCLIC POINT TYPES :



TYPE WEEKDAY CYCLIC

POSSREP WDPRI { WDI INTEGER
CONSTRAINT $WDI \geq 0$ AND $WDI \leq 6$ }

POSSREP WDPRC { WDC CHAR
CONSTRAINT WDC = 'Sun'
OR WDC = 'Mon'
.....
OR WDC = 'Sat' } ;

TYPE WEEKDAY CYCLIC

```
POSSREP WDPRI { WDI INTEGER
                CONSTRAINT WDI ≥ 0 AND WDI ≤ 6 }
POSSREP WDPRC { WDC CHAR
                CONSTRAINT WDC = 'Sun'
                   OR WDC = 'Mon'
                   .....
                   OR WDC = 'Sat' } ;
```

```
OPERATOR WDPRI ( WDI INTEGER ) RETURNS WEEKDAY ;
    /* code to return the WEEKDAY value represented by N, where */
    /* N = nonnegative remainder after dividing WDI by 7 */
END OPERATOR ;
```

```
OPERATOR THE_WDI ( WD WEEKDAY ) RETURNS INTEGER ;
    /* code to return the integer 0-6 that represents the WEEKDAY */
    /* value WD */
END OPERATOR ;
```

```
OPERATOR WDPRI ( WDC CHAR ) RETURNS WEEKDAY ;
  RETURN CASE
    WHEN WDC = 'Sun' THEN WDPRI ( 0 )
    WHEN WDC = 'Mon' THEN WDPRI ( 1 )
    .....
    WHEN WDC = 'Sat' THEN WDPRI ( 6 )
  END CASE ;
END OPERATOR ;
```

```
OPERATOR THE_WDC ( WD WEEKDAY ) RETURNS CHAR ;
  RETURN CASE
    WHEN THE_WDI (WD) = 0 THEN 'Sun'
    WHEN THE_WDI (WD) = 1 THEN 'Mon'
    .....
    WHEN THE_WDI (WD) = 6 THEN 'Sat'
  END CASE ;
END OPERATOR ;
```

WEEKDAY AS A POINT TYPE ???

- Makes good intuitive sense ...
- Does have an ordering but that ordering is **cyclic**
- “>” defined with respect to WDPRI ???
/ not very useful! */*

WEEKDAY AS A POINT TYPE ???

- Makes good intuitive sense ...
- Does have an ordering but that ordering is **cyclic**
- “>” defined with respect to that ordering ???
/ not very useful! */*

So (to repeat):

- No “first” or “last”, and “next” and “prior” never fail

CYCLIC specification implies need for:

```
OPERATOR NEXT_WEEKDAY ( D WEEKDAY )  
                                RETURNS WEEKDAY ;  
    RETURN WDPRI ( THE_WDI ( D ) + 1 ) ;  
END OPERATOR ;
```

```
OPERATOR PRIOR_WEEKDAY ( D WEEKDAY )  
                                RETURNS WEEKDAY ;  
    RETURN WDPRI ( THE_WDI ( D ) - 1 ) ;  
END OPERATOR ;
```

Also **COUNT_WEEKDAY ()** ... *returns 7*

INTERVAL SELECTORS, ETC. :

INTERVAL_WEEKDAY ([$b:e$]) : denotes interval consisting of weekdays $b, b+1, \dots, e$, in which *no weekday appears more than once*

E.g.:

```
INTERVAL_WEEKDAY ( [ WDPRC('Mon') : WDPRC('Fri') ] )  
INTERVAL_WEEKDAY ( [ WDPRC('Fri') : WDPRC('Mon') ] )  
INTERVAL_WEEKDAY ( [ WDPRC('Wed') : WDPRC('Wed') ] )  
INTERVAL_WEEKDAY ( [ WDPRC('Wed') : WDPRC('Tue') ] )
```

BEGIN, END, PRE, POST, POINT FROM, \in , \ni , COUNT ops are all straightforward! ... *but what about that last example?*

ALLEN'S OPERATORS :

Are, e.g., *universal intervals* [Sun:Sat] and [Wed:Tue] equal ???

No!

Recall $i1 = i2$:

True iff $b1 = b2$ and $e1 = e2$ both true

ALLEN'S OPERATORS :

Are, e.g., *universal intervals* [Sun:Sat] and [Wed:Tue] equal ???

No!

Recall $i1 = i2$:

True iff $b1 = b2$ and $e1 = e2$ both true

More generally, for any $i1$ and $i2$:

$i1 \subseteq i2$:

True iff starting at $b2$, we encounter $b1$, $e1$, $e2$ in that order

/ better definition: starting at $b2$, we encounter both $b1$ and $e1$, and we don't encounter $e1$ before $b1$, and we don't encounter $e2$ before $e1$ */*

$i1 \subseteq i2$: */* b2 then b1 then e1 then e2 */*

True iff starting at $b2$, we encounter both $b1$ and $e1$, and we don't encounter $e1$ before $b1$, and we don't encounter $e2$ before $e1$

$i1 \subset i2$, $i1 \supseteq i2$, $i1 \supset i2$ defined in terms of $i1 \subseteq i2$ in usual way

Examples:

[Tue:Thu]	\subset	[Mon:Fri]	:	TRUE
[Sat:Wed]	\supset	[Sun:Mon]	:	TRUE
[Mon:Fri]	\supseteq	[Thu:Sat]	:	FALSE
[Thu:Tue]	\subseteq	[Mon:Fri]	:	FALSE
[Sun:Sat]	\subseteq	[Wed:Tue]	:	FALSE

i1 BEFORE i2 : */* informally, b1 then e1 then b2 then e2 */*

True iff starting at *b1*, we encounter *e1* before *b2*, and we encounter *e2* before we encounter *b1* again

i1 BEFORE i2 : */* informally, b1 then e1 then b2 then e2 */*

True iff starting at *b1*, we encounter *e1* before *b2*, and we encounter *e2* before we encounter *b1* again

i2 BEFORE i1 : True iff *i1 BEFORE i2* true (!)

i1 AFTER i2 : True iff *i1 BEFORE i2* true (!)

i1 BEFORE i2 : */* informally, b1 then e1 then b2 then e2 */*

True iff starting at *b1*, we encounter *e1* before *b2*, and we encounter *e2* before we encounter *b1* again

i2 BEFORE i1 : True iff *i1 BEFORE i2* true (!)

i1 AFTER i2 : True iff *i1 BEFORE i2* true (!)

BEFORE and AFTER both reduce to DISJOINT! ... *Examples:*

[Tue:Wed]	BEFORE	[Fri:Sat]	: TRUE
[Fri:Sat]	BEFORE	[Tue:Wed]	: TRUE
[Tue:Wed]	AFTER	[Fri:Sat]	: TRUE
[Fri:Sat]	AFTER	[Tue:Wed]	: TRUE
[Tue:Fri]	BEFORE	[Wed:Sat]	: FALSE

i1 OVERLAPS i2 : */* informally, b1 then b2 then e1 then e2 */*

True iff *i1 DISJOINT i2* false

Note: Overlapping can happen **at both ends** ... *Examples:*

[Tue:Thu]	OVERLAPS	[Wed:Fri]	: TRUE
[Tue:Thu]	OVERLAPS	[Mon:Wed]	: TRUE
[Tue:Thu]	OVERLAPS	[Mon:Tue]	: TRUE
[Tue:Thu]	OVERLAPS	[Fri:Wed]	: TRUE
[Thu:Tue]	OVERLAPS	[Mon:Fri]	: TRUE

[Tue:Thu]	OVERLAPS	[Sat:Sun]	: FALSE
[Tue:Thu]	OVERLAPS	[Fri:Sun]	: FALSE
[Tue:Thu]	OVERLAPS	[Sun:Mon]	: FALSE

i1 MEETS i2 : */* informally, b1 then e1
then b2 (= e1 + 1) then e2 */*

True iff $b2 = e1 + 1$ true or $b1 = e2 + 1$ true

Meeting can happen at both ends ... Also, given pair of intervals can meet at one end and overlap at the other ... *Examples:*

```
[Tue:Thu] MEETS [Fri:Sat] : TRUE
[Tue:Thu] MEETS [Sun:Mon] : TRUE
[Tue:Thu] MEETS [Fri:Wed] : TRUE
```

```
[Tue:Thu] MEETS [Sat:Sat] : FALSE
[Tue:Thu] MEETS [Sat:Sun] : FALSE
```

***i1* MERGES *i2* :**

True iff *i1* *OVERLAPS* *i2* true or *i1* *MEETS* *i2* true

Example:

[Mon:Wed] MERGES [Fri:Sat] : FALSE

i1 BEGINS i2 : */* informally, b1 (= b2) then e1 then e2 */*

True iff $b1 = b2$ and $e1 \in i2$ both true

i1 ENDS i2 : */* informally, b2 then b1 then e1 (= e2) */*

True iff $e1 = e2$ and $b1 \in i2$ both true

Examples:

[Tue:Wed]	BEGINS	[Tue:Sat]	:	TRUE
[Tue:Fri]	BEGINS	[Tue:Thu]	:	FALSE
[Tue:Wed]	ENDS	[Sun:Wed]	:	TRUE
[Fri:Mon]	ENDS	[Sat:Mon]	:	FALSE

“SET OPERATORS” :

UNION, INTERSECT, and MINUS :

Definitions are tricky! /* see the textbook */

Examples on next few pages ...

PACK and UNPACK :

OK (definitions rely on interval UNION and INTERSECT)

UNION :

Examples:

```
[Mon:Wed] UNION [Fri:Sat] : undefined
[Mon:Thu] UNION [Tue:Fri] = [Mon:Fri]
[Tue:Fri] UNION [Mon:Thu] = [Mon:Fri]
[Mon:Thu] UNION [Sat:Tue] = [Sat:Thu]
[Thu:Sat] UNION [Mon:Thu] = [Mon:Sat]
[Sat:Sat] UNION [Sat:Sat] = [Sat:Sat]
[Tue:Fri] UNION [Sat:Tue] = [Sat:Fri]
[Tue:Fri] UNION [Sat:Mon] = [Tue:Mon]
[Sat:Mon] UNION [Tue:Fri] = [Sat:Fri]
```

INTERSECT :

Examples:

[Mon:Wed]	INTERSECT	[Fri:Sat]	:	<i>undefined</i>
[Mon:Thu]	INTERSECT	[Tue:Fri]	=	[Tue:Thu]
[Tue:Fri]	INTERSECT	[Mon:Thu]	=	[Tue:Thu]
[Mon:Thu]	INTERSECT	[Sat:Tue]	=	[Mon:Tue]
[Thu:Sat]	INTERSECT	[Mon:Thu]	=	[Thu:Thu]
[Sat:Sat]	INTERSECT	[Sat:Sat]	=	[Sat:Sat]
[Tue:Fri]	INTERSECT	[Sat:Tue]	=	[Tue:Tue]
[Tue:Fri]	INTERSECT	[Sat:Mon]	:	<i>undefined</i>
[Sat:Mon]	INTERSECT	[Tue:Fri]	:	<i>undefined</i>
[Thu:Wed]	INTERSECT	[Sat:Fri]	=	[Sat:Fri]
[Sat:Fri]	INTERSECT	[Thu:Wed]	=	[Thu:Wed]

MINUS :

Examples:

[Mon:Wed]	MINUS	[Fri:Sat]	=	[Mon:Wed]
[Mon:Thu]	MINUS	[Tue:Fri]	=	[Mon:Mon]
[Tue:Fri]	MINUS	[Mon:Thu]	=	[Fri:Fri]
[Mon:Thu]	MINUS	[Sat:Tue]	=	[Wed:Thu]
[Thu:Sat]	MINUS	[Mon:Thu]	=	[Fri:Sat]
[Sat:Sat]	MINUS	[Sat:Sat]	:	<i>undefined</i>
[Tue:Fri]	MINUS	[Sat:Tue]	=	[Wed:Fri]
[Tue:Fri]	MINUS	[Sat:Mon]	=	[Tue:Fri]
[Sat:Mon]	MINUS	[Tue:Fri]	=	[Sat:Mon]
[Tue:Sun]	MINUS	[Sat:Wed]	:	[Thu:Fri]
[Sat:Fri]	MINUS	[Thu:Wed]	:	<i>undefined</i>
[Tue:Thu]	MINUS	[Thu:Fri]	=	[Tue:Wed]

NET OF ALL THIS :

- Cyclic types are valid point types
- Such types behave normally, except that *first* and *last* don't apply and *next* and *prior* never fail
- Corresponding interval types also behave more or less normally, except that we don't (can't!) insist that $b \leq e$ and certain operators need somewhat revised definitions
- Conditions stated previously for a type to be usable as a point type are sufficient but not all necessary

APPENDIXES :

- A. Cyclic point types
- B. Is ordinality necessary?
- C. Generalizing PACK and UNPACK
- D. A **Tutorial D** grammar
- E. Implementation considerations
- F. References and bibliography

AGENDA :

- I. A review of relational concepts
- II. Laying the foundations
- III. Building on the foundations
- IV. SQL support
- V. Appendixes

THE END :

Thank you for listening !