

# Foundation Matters

*a keynote presentation for VLDB'02*

*Hong Kong, August 2002*

by

**C. J. Date**

Copyright © C. J. Date 2002. All rights reserved. No part of this material may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photographic, or otherwise, without the explicit written permission of the copyright owner.

# THESIS AND AGENDA :

---

Foundations are **IMPORTANT** ...

Yet, foundations in the DB world are:

- Widely perceived as uninteresting or unimportant
- Widely misunderstood or underappreciated
- Widely considered to be a closed subject

Will review some of my own foundation-level investigations from the past twenty years or so

# *THE THIRD MANIFESTO :*

---

C. J. Date and Hugh Darwen:

*Foundation for Future Database Systems: The Third Manifesto*

2nd edition (Addison-Wesley, 2000)



a detailed study of the impact of type theory  
on the relational model of data,  
including a comprehensive model of type inheritance

# WHY ???

---

Because we were concerned over certain current IT trends ...

1. Equating relations and object classes ("**The First Great Blunder**")
2. Mixing pointers and relations ("**The Second Great Blunder**")
3. More generally, failing to understand relational theory

*The Third Manifesto* is **evolutionary**, not revolutionary:  
Builds on (and tidies up) the relational model -- *not* an  
attempt to replace that model

# PRIOR ART :

---

1. Malcolm Atkinson *et al.* (1989): "The Object-Oriented Database System Manifesto"
  - Essentially ignores the relational model
  - *"We are taking a Darwinian approach: We hope that, out of the set of experimental prototypes being built, a fit model will emerge"*
2. Michael Stonebraker *et al.* (1990): "Third Generation Database System Manifesto"
  - Agrees that the relational model must not be discarded ... but fails to face up to the hopelessness of continuing to build on SQL

# A GUIDING PRINCIPLE :

---

*All logical differences are big differences*

-- Wittgenstein (attrib.)

*All logical mistakes are big mistakes*

-- Darwin's corollary

# ONE IMPORTANT LOGICAL DIFFERENCE : VALUES vs VARIABLE

---

- VALUE : *an "individual constant"*
  - no location in time or space
  - cannot be changed
  - can be represented in memory (by some encoding)
- VARIABLE : *a holder for (the representation of) a value*
  - has location in time and space
  - can be updated (i.e., current value can be replaced by another)

Definitions based on J. Craig Cleaveland, *An Introduction to Data Types* (Addison-Wesley, 1986)

Values and variables -- in fact, *types* -- can be arbitrarily complex

# VALUE vs. VARIABLE CONFUSION : AN EXAMPLE

---

"We distinguish the declared type of a variable from ... the type of the object that is the current value of the variable ...

*(so an object is a value)*

"... we distinguish objects from values ...

*(so an object isn't a value after all) -- ???*

"... a MUTATOR [is an operation such that it's] possible to observe its effect on some object."

*(in fact, an object is a variable) -- ?????*

# RELATION VALUES *vs.* RELATION VARIABLES :

---

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

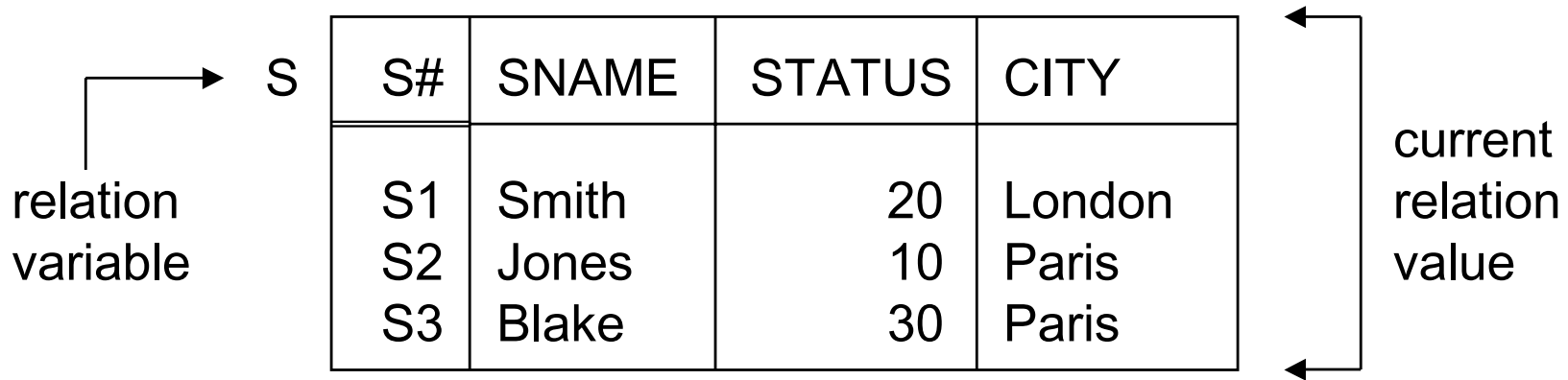
Consider: `DECLARE N INTEGER ...` -- arb pgmg lang

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

Likewise: `CREATE TABLE T ...` -- SQL

T is a relation *variable* or **relvar** whose *values* are relations *per se*\*

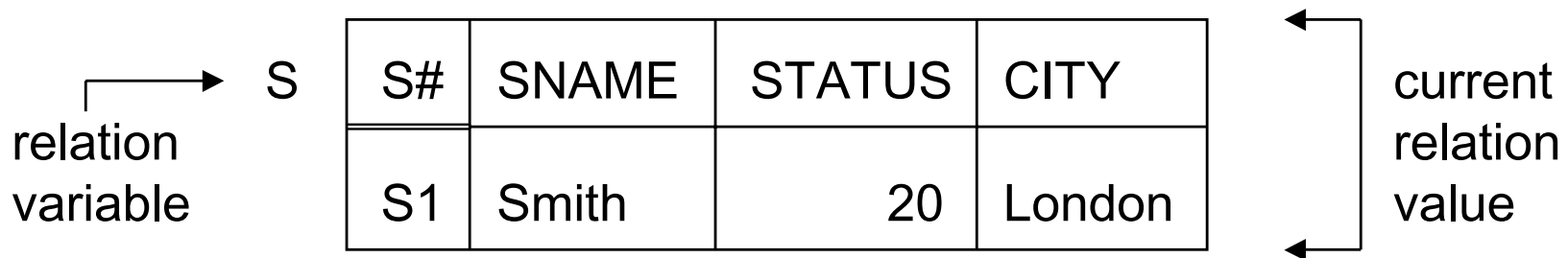
(ignoring nulls, dups, and other SQL quirks)



DELETE S WHERE CITY = 'Paris' ;

Really shorthand for:

S := S WHERE NOT ( CITY = 'Paris' ) ;



# ANOTHER GUIDING PRINCIPLE : *CONCEPTUAL INTEGRITY*

---

"... conceptual integrity is *the* most important consideration in system design. It is better to have a system omit certain anomalous features [and] to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas.

"A clean, elegant programming product must present ... a coherent mental model ... [Conceptual] integrity ... is the most important factor in ease of use ... **Today I am more convinced than ever.** Conceptual integrity *is* central to product quality."

-- from *The Mythical Man-Month*, by Fred Brooks

# BACK TO THE FUTURE :

---

- We don't think SQL is capable of providing the kind of firm foundation we need
- Instead, we believe that any such foundation should be solidly rooted in **THE RELATIONAL MODEL**
- *Aside:* In case it needs spelling out ...
  - SQL  $\neq$  the relational model !!!
- We agree that certain other features are desirable too, including aspects of **OBJECT ORIENTATION** ➡
  - I.e., *The Third Manifesto* provides a foundation for "object/relational" DBMSs in particular

# OTHER DESIRABLE FEATURES :

---

We believe that these features are TOTALLY ORTHOGONAL to the relational model

**Do NOTHING to the relational model !!!**

The relational model needs

- no *EXTENSION*
- no *CORRECTION*
- no *SUBSUMPTION*

and above all

- no *PERVERSION*

in order for some (relational) language to accommodate these orthogonal features

# RELATIONS :

---

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

The *heading* is a set of < *attrib-name:type-name* > pairs;  
the *body* is a set of tuples conforming to that heading

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

MAJOR_P#	P#	MINOR_P#	P#	QTY	QTY
P1		P2			2
P1		P3			4
P2		P3			1
P2		P4			3
P3		P5			9
P4		P5			8
P5		P6			3

---

Informally, we often ignore the *type names* (aka domain names) in the heading:

MAJOR_P#	MINOR_P#	QTY
P1	P2	2
P1	P3	4
P2	P3	1
P2	P4	3
P3	P5	9
P4	P5	8
P5	P6	3

# TYPES CAN BE ARBITRARILY COMPLEX (*very important!*) :

---

**THE QUESTION AS TO WHAT DATA TYPES ARE SUPPORTED IS ORTHOGONAL TO THE QUESTION OF SUPPORT FOR THE RELATIONAL MODEL**

*More succinctly:*

**TYPES ARE ORTHOGONAL TO TABLES**

The relational model has NEVER prescribed data types

(it's never been implemented either -- but that's another matter)

# *(Very important!)* WAY OF THINKING ABOUT RELATIONS :

---

Heading represents a PREDICATE (truth-valued function):  
e.g.,

*Part MAJOR\_P# contains QTY of part MINOR\_P#*

Parameters (MAJOR\_P#, MINOR\_P#, QTY in the example)  
stand for values of the relevant types

Tuples represent TRUE PROPOSITIONS ("instantiations" of  
the predicate that evaluate to *true*), obtained by substituting  
arguments for the parameters: e.g.,

*Part P1 contains 2 of part P2*

*Part P1 contains 4 of part P3 (etc.)*

In other words:

---

**TYPES are sets of things we can talk about;**  
**RELATIONS are (true) statements about those things!**

Note three very important corollaries ...

- 
1. Types and relations are both *NECESSARY*
  2. They're not the same thing
  3. They're *SUFFICIENT* (as well as necessary)

*A database (with its operators) is a logical system!*



Also, don't forget the *Closed World Assumption (CWA)*

# A NICE ANALOGY :

---

**TYPES are to RELATIONS**

**as**

**NOUNS are to SENTENCES**

-- acks to Hugh Darwen once again

# EXTERNAL vs. INTERNAL PREDICATES :

---

Predicates (and propositions) discussed so far are **external**  
-- understood by the user, not the system

- System can't know what it means for one "part" to "contain" another, etc. -- that's **interpretation**
- System can't know *a priori* whether what the user tells it is true! (can only check integrity constraints)

**Internal** predicate = system's approximation to external predicate = AND of all pertinent integrity constraints = *relvar predicate* for pertinent relvar  $R$

- Criterion for acceptability of updates on  $R$

# LOGICAL DATABASE DESIGN :

---

1. Pin down external predicates as carefully as possible (albeit informally)
2. Map the output from the first step into a set of formal relvars and corresponding internal predicates (i.e., constraints)

( 3. Iterate! )



*Note:* "E/R modeling" is almost totally incapable of dealing with constraints

*Note:* All of the above is highly relevant to what the commercial world calls *business rules*

---

Predicates and constraints are **vitaly important notions** in the database world, but they aren't very widely understood

Attempts at a classification scheme:

- Stonebraker 1975
- Date 1983, 1990, 1995, **1998, 2000 (!)**
- Codd 1990
- Ullman / Widom 1997
- Kimball 2000

See also Melton / Simon, *Understanding the New SQL: A Complete Guide* (1993), Chapter 10, "Constraints, Assertions, and Referential Integrity"

# SURVEY OF 37 DATABASE TEXTBOOKS :

---

Number of books with entire chapter on integrity: 1

- Poor treatment, even there
- Three others also had an "integrity" chapter, but one used it to mean normalization and the other two to mean locking and concurrency control -- *caveat lector!*

Most didn't even mention integrity in a chapter title, and those that did bundled it weirdly

- E.g., "Integrity, Views, Security, and Catalogs"

Number of books with good definition / explanation: 0

# ONE IMPORTANT POINT :

---

Predicates are the basis of the solution to the view updating problem!

*All views are theoretically updatable, modulo integrity constraint violations*

By the way: I'm NOT talking about (yecch) "materialized views" -- *which in fact are a contradiction in terms*

But this is not the place for details

# TYPE INHERITANCE : IS A CIRCLE AN ELLIPSE ?

---

- Let CIRCLE be a subtype of type ELLIPSE
- Let E denote an ellipse with  $a = 4$ ,  $b = 3$ . Consider:

```
THE_A ( E ) := LENGTH ( 3.0 ) ;  
/* "zap" the a-semiaxis component of variable E */
```

Does the most specific type of E become CIRCLE ???

- If it does, we have **specialization by constraint** ("S by C")

# WELL ...

---

In the real world, an ellipse with equal semiaxes is certainly a circle

A model that does not understand this fact can hardly be said to be "a good model of reality"

**Our model of type inheritance DOES support S by C -- though originally it did not**

- Many advantages (details omitted here)

*Why did we reject it originally ???*

- Because just about all other attempts at an inheritance model did so, too (not a very good reason!). E.g.:

---

"Is SQUARE a subclass of RECTANGLE? ... Stretching the x-dimension of a rectangle is a perfectly reasonable thing to do. But if you do it to a square, then the object is no longer a square. This is not necessarily a bad thing conceptually. When you stretch a square you *do* get a rectangle ... But ... most object-oriented languages do not want objects to change class ... All of this suggests [a] design principle for classification systems: *A subclass should not be defined by constraining a superclass.*"

James Rumbaugh: "A Matter of Intent: How to Define Subclasses", *Journal of Object-Oriented Programming* (September 1996)

---

Note the rationale: OO languages "do not want objects to change class" ... we'd rather get the *model* right first before worrying about implementations!

Though Rumbaugh does also say:

"It would be computationally infeasible to support a rule-based, intensional definition of class membership, because you would have to check the rules after each operation that affects an object."

But we reject this claim (i.e., we believe we know how to implement S by C efficiently)

# ANOTHER QUOTE :

---

Bjarne Stroustrup: *The C++ Programming Language*  
(3rd edition), Addison-Wesley (1997)

[In] mathematics a circle is a kind of an ellipse, but in most programs a circle should not be derived from an ellipse or an ellipse derived from a circle. The often-heard arguments 'because that's the way it is in mathematics' and 'because the representation of a circle is a subset of that of an ellipse' are not conclusive and most often wrong. This is because for most programs, the key property of a circle is that it has a center and a fixed distance to its perimeter. All behavior of a circle (all operations) must maintain this property (invariant ...).

(cont.)

---

On the other hand, an ellipse is characterized by two focal points that in many programs can be changed independently of each other. If those focal points coincide, the ellipse looks like a circle, but it is not a circle because its operations do not preserve the circle invariant. In most systems, this difference will be reflected by having a circle and an ellipse provide sets of operations that are not subsets of each other.

# GENERALIZATION BY CONSTRAINT :

---

Support for S by C implies support for "*generalization by constraint*" (G by C) as well ... E.g.:

```
E := ELLIPSE ( LENGTH ( 4.0 ), LENGTH ( 3.0 ), ... );  
/* MST(E) = ELLIPSE */
```

```
THE_A ( E ) := LENGTH ( 3.0 );  
/* MST(E) now CIRCLE, thanks to S by C */
```

```
THE_B ( E ) := LENGTH ( 2.0 );  
/* MST(E) now ELLIPSE again, thanks to G by C */
```

# AN INTERESTING OBSERVATION :

---

Let  $O$  be a language that supports object IDs, or in other words *pointers* (prohibited in *The Third Manifesto*)

Variables in  $O$  typically contain such pointers instead of regular values -- e.g., variables  $XE$  and  $XC$  contain *pointers to ellipses and circles*, not ellipses and circles as such (i.e., pointers to ellipse and circle *variables* or "objects")

- *Referencing*: Given a variable  $V$ , returns a pointer to  $V$
- *Dereferencing*: Given a variable  $P$  of type pointer, returns the variable  $P$  currently points to

---

```
VAR E  ELLIPSE ;
VAR XE PTR_TO_ELLIPSE ;          /* PTR_TO_ is a type generator; */
VAR XC PTR_TO_CIRCLE ;          /* PTR_TO_CIRCLE is a proper */
                                /* subtype of PTR_TO_ELLIPSE */
```

```
E := CIRCLE ( LENGTH ( 5.0 ) , POINT ( 1.0, 1.0 ) );
/* OK because of substitutability: MST(E) now CIRCLE */
```

```
XE := PTR_TO ( E );              /* PTR_TO is the referencing op */
```

```
XC := TREAT_DOWN_AS_PTR_TO_CIRCLE ( XE );
/* XC and XE contain same pointer value */
```

```
/* now -- the crucial assignment: */
THE_A ( Deref ( XE ) ) := LENGTH ( 6.0 );
```

**What happens?**

# FIRST POSSIBILITY :

---

## Run-time type error:

- Because  $MST(DEREF(XE))$  is CIRCLE and assignment to THE\_A not supported for variables of current  $MST$  CIRCLE (as in our original model)
  
- **Model is bad because:**
  - a. No G by C (and hence no S by C, and hence model not "good model of reality")
  
  - b. Run-time errors undesirable anyway

# SECOND POSSIBILITY :

---

Assignment "succeeds" (update done) but G by C does not occur:

→ **Model is bad** because:

- a. No G by C (and hence no S by C, and hence model not "a good model of reality")
- b. XC now points to a noncircular circle
- c. *No type constraint support!*

**Note:** This case is SQL:1999!

# THIRD POSSIBILITY :

---

**Assignment "succeeds" (update done) and G by C occurs:**

→ **Model is bad** because:

- a. XC ( $DT = PTR\_TO\_CIRCLE$ ) now points to a variable of current *MST ELLIPSE* ... XC is broken!
- b. *No type constraint support!*

# CONCLUSION :

---

- **Model is bad ...**
- Culprit is concept of shared variables ...
- It's OIDs that allow shared variables in the first place ...
- *OIDs and a good model of type inheritance are incompatible!*

---

Perhaps this is why there haven't been any good inheritance models in the past (so far as we know) ??? ... Prior work all seems to have been done in an OO context ... hence, OIDs were a given

But OIDs imply that S by C and G by C can't be supported! ... and so model must be bad!

Since support for objects implies support for OIDs, *objects and a good model of inheritance are incompatible!*

# TEMPORAL DATA :

---

C. J. Date / Hugh Darwen / Nikos A. Lorentzos

*Temporal Data and the Relational Model*

(Morgan Kaufmann, 2003)



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

# TEMPORAL SUPPORT :

---

**Do NOTHING to the relational model !!!**

The relational model needs

- no *EXTENSION*
- no *CORRECTION*
- no *SUBSUMPTION*

and above all

- no *PERVERSION*

in order for some (relational) language to accommodate appropriate temporal features

# INTERVAL TYPE GENERATOR : AN EXAMPLE

---

SP\_DURING

S#	P#	DURING
S2	P1	[d02:d04]
S2	P1	[d08:d10]
S2	P2	[d03:d03]
S2	P2	[d09:d10]
S4	P2	[d06:d09]
S4	P4	[d04:d08]
S4	P5	[d05:d10]

← type = INTERVAL\_DATE

(d02 = "day 2" etc.)

Plus operators on intervals (OVERLAPS, MEETS, INCLUDES, etc.)

# UNPACK AND PACK OPERATORS (shorthand!)

---

*r*

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

UNPACK *r*  
ON DURING

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

PACK *r*  
ON DURING

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

# U\_OPERATORS (shorthand!) :

---

Important for raising the level of abstraction (also for implementation efficiency, probably)

USING ( ACL ) ◀ R1 MINUS R2 ▶

≡ PACK

( ( UNPACK R1 ON ( ACL ) )  
MINUS  
( UNPACK R2 ON ( ACL ) ) )  
ON ( ACL )

Similarly for all other relational operators, also relational comparisons

**Regular operators just a special case**

# ADDITIONAL "TEMPORAL" CONCEPTS AND CONSTRUCTS :

---

- New database design methodology
- More database design science (6NF)
- New "key-like" constraints (shorthand)
  - *Regular key constraints a special case*
- New general constraints (shorthand)
- New query techniques (shorthand)

- 
- New update operators (shorthand)
    - *Regular operators a special case*
  - New update techniques (shorthand)
  - Stated times and logged times (shorthands)
  - Cyclic types etc.

By the way: *Type inheritance and S by C turn out to be important in all this!*

# CONCLUDING REMARKS :

---

## **Foundation matters!**

The relational model is deeper than you might think

- Of course, we've only scratched the surface in this presentation ...

Still plenty of novel work to be done

I wish you success with your own DB investigations!