How *TTM* Got Off The Ground
or How One Thing Leads to Another

by Hugh Darwen

The 1993 presentation of the UK Open University's course on relational databases included a question in a test paper in which the students were asked to state the meaning, in real world terms, of the following SQL query:

```
SELECT NAME
FROM   CITY C1
WHERE  4 > ( SELECT COUNT(*)
             FROM   CITY C2
             WHERE  C1.POPULATION < C2.POPULATION )
```

As you can see, this is a "quota query", asking for the names of the four most populous cities (noting that if there is a tie for fourth place, then all participants in that tie will appear in the result).

I was one of the tutors whose task it was to mark these test papers. I struggled with this question and so did most of the students. My difficulty was caused mainly by the perverse way in which the two comparisons were written, causing me to translate this, word for word, like this:

> Show names of cities where four is greater than the number of other cities where the population of this one is less than the population of the other one.

By analogy it seemed like asking somebody trying to purchase alcoholic drinks if seventeen was less than their age. I tried inverting the two comparisons:

```
SELECT NAME
FROM   CITY C1
WHERE  ( SELECT COUNT(*)
         FROM   CITY C2
         WHERE  C2.POPULATION > C1.POPULATION ) < 4
```

Now my translation became

> Show names of cities where the number of other cities whose population is greater than the population of this one is less than four.

and the real meaning became clear. I thought it was a bit mean of the test paper deviser to set the question that way—after all, we're supposed to be testing the students' understanding of relational stuff, not their mental agility—and I told Chris Date about the question, asking him if he agreed that the way in which the comparisons were written made it more difficult than it ought to have been. He readily agreed but added, "I think I know why they set it that way. Many SQL implementations won't let you write it the other way—and by the way, that includes your own company's implementation, Hugh—DB2".

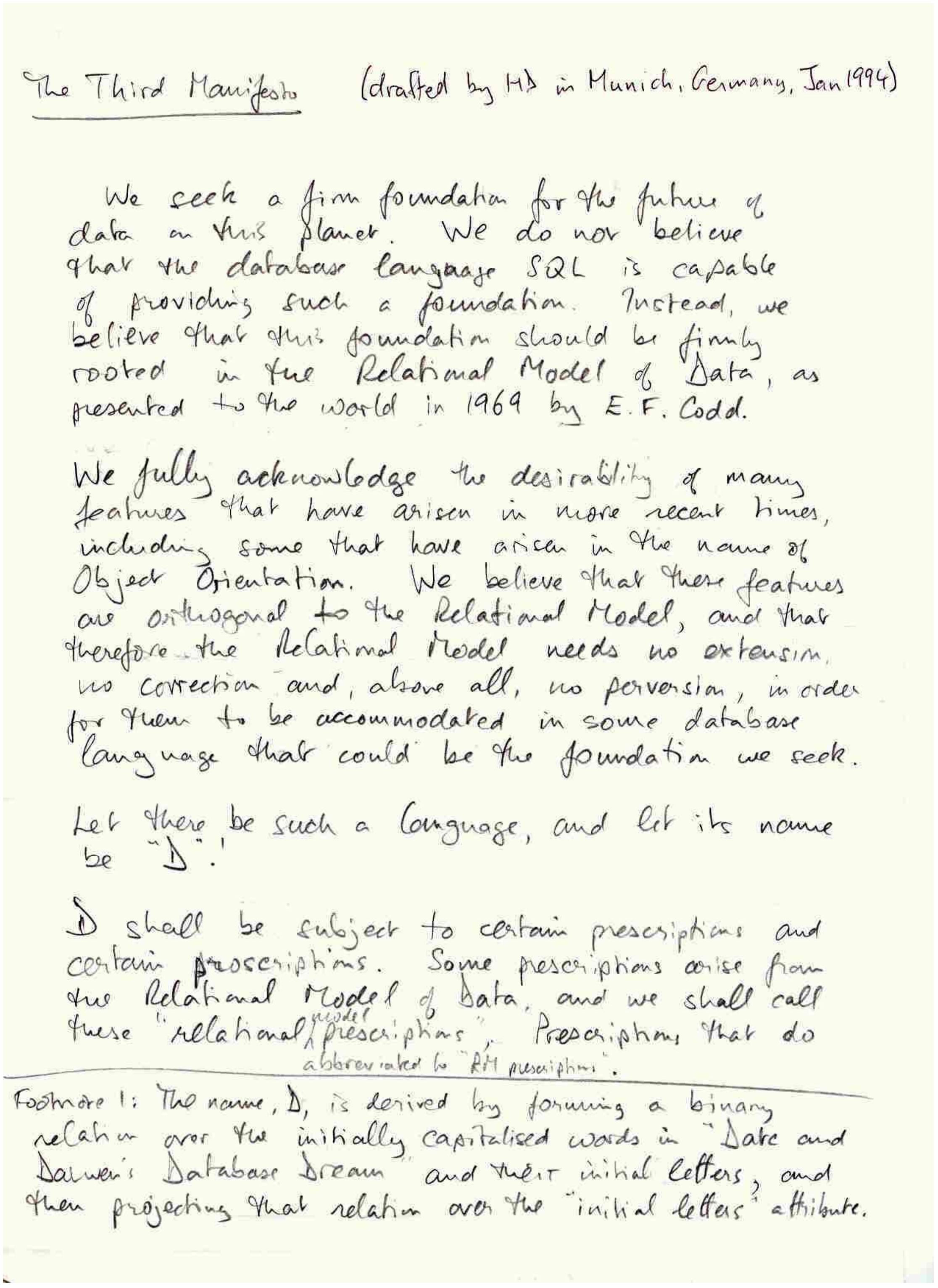
I was flabbergasted by that information. I had come across many funny quirks in programming languages during my career but never one that disallowed certain expressions from appearing on the left-hand side of a comparison while allowing them on the other side. Was there some problem, perhaps, with handling comparisons having "scalar subqueries" on both sides? I checked the new edition of the international standard that had appeared in 1992 and found that no such restriction was mentioned in it.

Now, at that time I was IBM's UK representative in the ISO Working Group that was (and still is) responsible for the production of database language standards, which by then had become just SQL standards. The aim of an IBM representative was to align the standard with DB2 (just as it was the aim of Oracle's representatives to align it with Oracle, and so on). Perceiving a point of non-conformance in DB2, I got on the phone to my US colleague, Nelson Mattos, who was IBM's representative on the ANSI committee and from whom IBM representatives in other national bodies took guidance. I told him about the point of non-conformance that had come to light, suggesting that he might wish to get on to the development groups responsible for DB2 on the various platforms IBM supported, to advise them to address the discrepancy.

What actually happened was not the outcome I had envisaged. It turned out that the previous edition of the standard had indeed included the restriction in question—no subqueries on the left-hand sides of comparisons. At the next meeting of that ISO Working Group, in Munich, Germany, January 1994, there appeared a change proposal from the USA, authored by Nelson Mattos, against the Technical Corrigendum for SQL:1992. ("Technical corrigendum" is ISO-speak—a euphemism for "errata".) The proposal in question was a simple one: to reinstate the restriction on the use of subqueries in comparisons! It was my duty to my employer to try to persuade my own national body colleagues that the UK should support this proposal, so I swallowed my pride and did just that, and the proposal was accepted.

Well, there's such a thing as the last straw—the one that breaks the camel's back—and this was the last straw for me. Ever since the publication in 1990 of *The Object-Oriented Database System Manifesto* and the response paper, *Third Generation Database System Manifesto*, Chris and I had been wondering if we might attempt a response to both of those manifestos, and we had been using "The Third Manifesto" as a provisional title for communication between the two of us. I was so ashamed by my participation in such skulduggery that during the interim weekend of that meeting, while many of the other delegates went off with their skis to the distant mountains of the Austrian Alps that I could see from my hotel room window, I put pencil to paper and faxed the resulting nine pages to Chris Date. What follows, with apologies for the awful scribble, is what I wrote[1] (some of the alterations might have resulted from Chris's immediate response by phone).

[1] Thanks to Lindsay Darwen for scanning in these pages for me, nearly 18 years after they were written.



The Third Manifesto (drafted by HD in Munich, Germany, Jan 1994)

We seek a firm foundation for the future of data on this planet. We do not believe that the database language SQL is capable of providing such a foundation. Instead, we believe that this foundation should be firmly rooted in the Relational Model of Data, as presented to the world in 1969 by E.F. Codd.

We fully acknowledge the desirability of many features that have arisen in more recent times, including some that have arisen in the name of Object Orientation. We believe that these features are orthogonal to the Relational Model, and that therefore the Relational Model needs no extension, no correction and, above all, no perversion, in order for them to be accommodated in some database language that could be the foundation we seek.

Let there be such a language, and let its name be "D".[1]

D shall be subject to certain prescriptions and certain proscriptions. Some prescriptions arise from the Relational Model of Data, and we shall call these "relational model prescriptions", Prescriptions that do
abbreviated to "RM prescriptions".

Footnote 1: The name, D, is derived by forming a binary relation over the initially capitalised words in "Date and Darwen's Database Dream" and their initial letters, and then projecting that relation over the "initial letters" attribute.

not arise from the Relational Model we call "other orthogonal prescriptions", abbreviated to "OO prescriptions". We similarly categorise D's proscriptions. We now proceed to itemise D's prescriptions and proscriptions:

<u>RM Prescriptions</u>

1. A "**database**" is a persistent, named variable bound to set of persistent, named, relation-valued variables. (A variable is relation-valued iff it is bound to some relation) We shall refer to such variables as "base tables".
2. A <u>relation</u>, R, is a set, H, of zero or more <u>attributes</u>, paired with a set, B, of n-tuples over H. A n-tuple in B is a set containing <u>exactly one</u> value corresponding to each attribute in H (this is the property commonly known as First Normal Form, or 1NF). The set of permissible values ~~values that are permitted~~ for some attribute, <u>a</u>, is the <u>domain</u> of <u>a</u>.

3. D shall make no prescriptions, and no proscriptions, concerning the domains over which attributes of relations may be defined. However, D shall include some collection of primitive constructs to support the definition of domains, and associated operators, of arbitrary complexity. (An operator that returns a value in some domain is termed a "scalar operator", to distinguish such operators from "relational operators".)

4 + criterion of identity!

4. D shall include primitive constructs to support definition of a database and the subsequent modification of such a definition by the addition or removal of base tables.

5. D shall provide a notation, REL, in which relation-valued expressions of arbitrary complexity can be expressed. The name of a base table, say BT, denoting the relation-value to which BT is bound, shall be a valid expression in REL.
+relational literal! The Relational Algebra defined in [ref] shall be expressible in REL.
~~(What about recursion?)~~

6. D shall permit a REL expression to be assigned to a base table.
(Note: Of course, this does not prohibit the additional provision of convenient shorthands similar to the INSERT, UPDATE and DELETE commands of SQL.)

7. D shall provide a notation, CL, in which truth-valued expressions of arbitrary complexity can be written for the purpose of defining database constraints. CL shall include an operator that returns true or false according to whether or not the value of some specified REL expression is empty.

RM Proscriptions

The observant reader will note that many of the proscriptions documented in this section are logical consequences of the RM Prescriptions. In view of the unfortunate mistakes that have been made in the database language SQL, we feel it necessary, nevertheless, to write down these consequences by way of clarification:

1. For every relation, R, expressible in REL, the attributes of R shall be distinguishable by name. (I.e., no more "anonymous columns" like SQL's "SELECT X+Y FROM T", and no more "duplicate column names as in SELECT X,X FROM T and SELECT T1.X, T2.X FROM T1,T2.)

2. For every relation, R, expressible in REL, if t1 and t2 are both n-tuples in R, then there shall be some attribute, a, of R such that the value for a in t1 differs from the value of a in t2.
(I.e. "duplicate rows" are outlawed!)

3. For every relation, R, expressible in REL, for every attribute, a, of R, for every n-tuple, t, in R, a shall be bound to some value in the domain of a.
(I.e. "NO MORE NULLS, EVER!")

4. D shall not forget that the relation over zero attributes is a respectable (and interesting) one.

~~5. If an assignment to a relation-valued function, FR, (commonly called "updating through views") is prohibited~~

5. Let T1 be some point in time at which the relation R1 is assigned to S, and let T2 be some point in time after T1 at which the relation R2 is assigned to S, and let there be no other assignments to S between T1 and T2. Then, at all times between T1 and T2, the RCR expression "S" shall return R1.
(I.e. no more updating views WITHOUT CHECK OPTION!).

6. All arbitrary restrictions documented in [ref], and all other arbitrary restrictions of a similar nature, are absolutely prohibited

7. D shall not be SQL.

8. D shall be constructed according to those well-established principles of Good Language Design that are so lamentably violated in SQL.

OO Prescriptions

1. Inheritance shall be multiple, if at all. secretaries

2. Representation shall be secret, and not necessarily be inherited

3. Every operator defined for supertype shall also be defined for subtype (defined 'defined for') Mult functions
(the implementation for the subtype may differ from that for the supertype)

4. An instance of subtype may appear wherever an instance of supertype expected (type test?)

5. Transactions shall be explicitly started and ended.

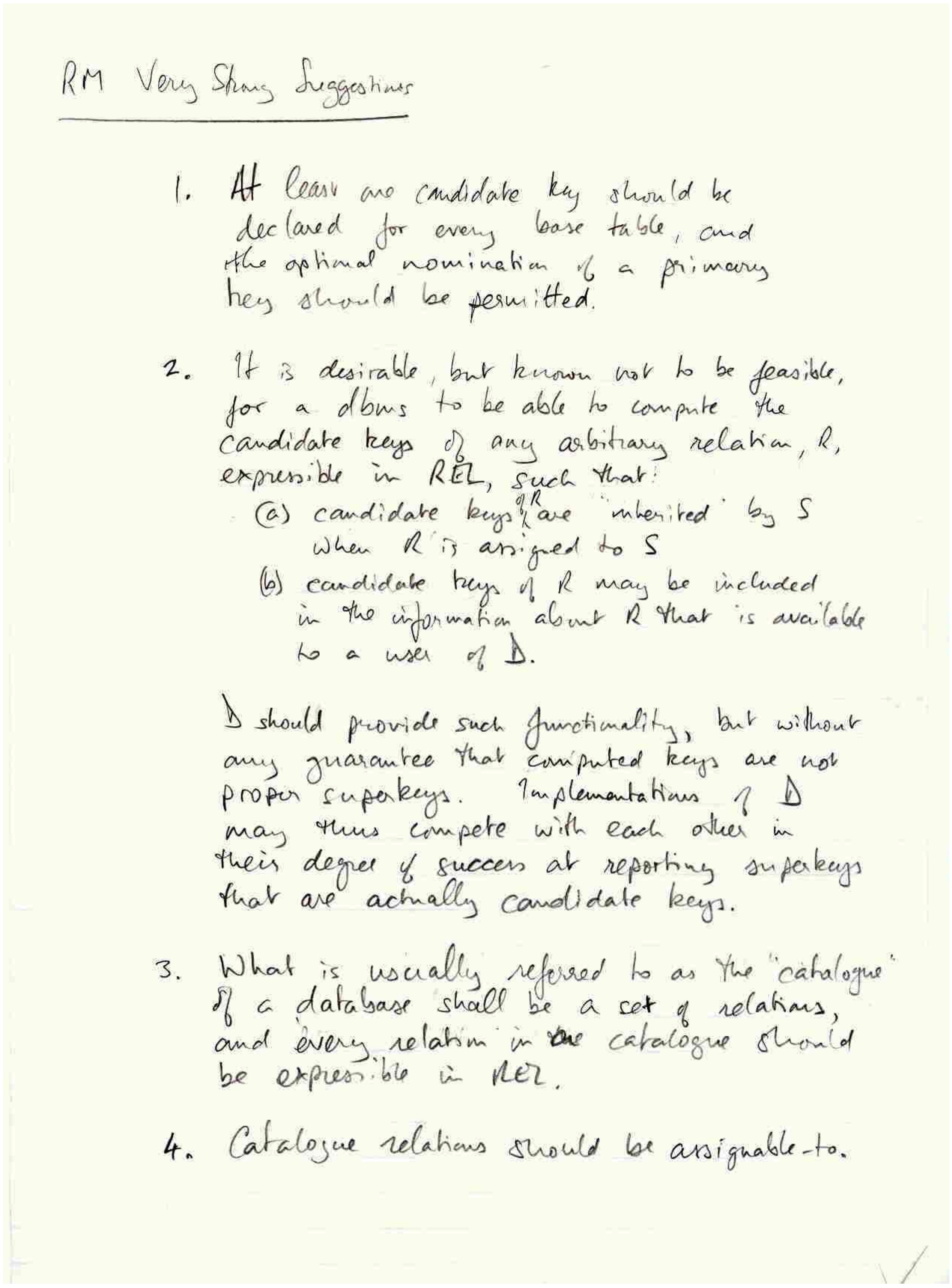
6. It shall be possible to nest transactions

7. No implicit conversions.)

OO Proscriptions

1. Special typing and behaviour shall be confined to <u>domains</u>, and not applicable to relations.

2. No function invocation by sending messages to objects



RM Very Strong Suggestions

1. At least one candidate key should be declared for every base table, and the optional nomination of a primary key should be permitted.

2. It is desirable, but known not to be feasible, for a dbms to be able to compute the candidate keys of any arbitrary relation, R, expressible in REL, such that:
   (a) candidate keys of R are "inherited" by S when R is assigned to S
   (b) candidate keys of R may be included in the information about R that is available to a user of D.

   D should provide such functionality, but without any guarantee that computed keys are not proper superkeys. Implementations of D may thus compete with each other in their degree of success at reporting superkeys that are actually candidate keys.

3. What is usually referred to as the "catalogue" of a database shall be a set of relations, and every relation in the catalogue should be expressible in REL.

4. Catalogue relations should be assignable-to.

5. REL should include at least one special operator for the treatment of recursive relations, such that the transitive closure of such a relation can be conveniently expressed.

6. Universal quantification shall not be significantly more difficult to express than existential quantification

For example, if REL includes a specific operator for relational projection, then it should also include one for the general form of relational division noted in [ref].

7. D should support, but not require, invocation from so-called "host programs" written in languages other than D.

Similarly, D should support, but not require, the use of other programming languages for implementation of (user-defined) scalar functions.

8. SQL should be expressible in D. So that a painless migration route is available for current SQL users.

✓

OO Very Strong Suggestions

1. Inheritance should be supported, and multifunctions for resolving function invocation based on all arguments.

2. If some notation is available by which "attributes" appear to be visible at the interface of a type, there should be no correspondence between such attributes and those which appear in the encapsulated representation. What is ~~are~~ commonly known as "public attributes" ~~should be~~ notation should be merely syntactic shorthand for certain special function invocations.

3. "Collection" types, such as LIST, ARRAY and SET, are commonly found in languages supporting rich type systems. If any such types are available in D, then a RELATION type should be included so that, for example, the domain of some attribute of some relation may itself be relation-valued.

4. If a collection type C other than RELATION, ~~are~~ is supported, ~~can~~ then a conversion function, say C2R, is available for converting values of type C to values of type R, and an inverse conversion function, R2C, is also available. C2R(R2C(r)) = r, for every relation, r, expressible in REL, and R2C(C2R(c)) = c for every expressible value in type C.