Codd's First Relational Papers:

A Critical Analysis

C. J. Date

Jakob Bernoulli's productive years coincided with Leibniz's discovery of calculus,
and [he] was one of the chief popularizers of this immensely fruitful subject.
As with any developing theory, calculus benefited from those who followed in its creator's
footsteps, scholars whose brilliance may have fallen short of Leibniz's but whose
contributions toward tidying up the subject were indispensable.
Jakob Bernoulli was one such contributor.
—William Dunham, *The Mathematical Universe* (1974)

This paper was written as a companion to, and reference source for, Hugh Darwen's paper "Why Are There No Relational DBMSs?" (available in PDF at www.thethirdmanifesto.com). It has benefited from Hugh's careful review of earlier drafts. It's intended as a contribution to the history of the field of relational database technology; until further notice, however, I respectfully request that distribution be restricted and carefully controlled. Thank you.

PRELIMINARIES

I'm hardly alone in my strong belief that relational theory is the right and proper foundation for database technology in general. That theory was originally introduced by E. F. Codd in two landmark papers:

- "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks," IBM Research Report RJ599, August 19th, 1969—referred to throughout what follows as *the 1969 paper*

- "A Relational Model of Data for Large Shared Data Banks," *CACM 13*, No. 6, June 1970—referred to throughout what follows as *the 1970 paper*[1]

And yet here we are, over 45 years later, and what do we find? Well:

- First, the teaching of relational theory, in universities in particular, seems everywhere to be in decline. What's more, what teaching does exist seems not to have caught up—at least,

[1] The 1970 paper is usually credited with being the seminal paper in the field, but this characterization is a little unfair to its 1969 predecessor. In fact the 1970 paper was, and is, essentially just a somewhat revised and extended version of the 1969 paper.

not properly—with the numerous developments in relational theory that have occurred since publication of those first two papers of Codd's.

- Second, no truly relational DBMS has ever been widely available in the marketplace.

Sadly, it seems to me that part of the blame for this depressing state of affairs has to be laid at Codd's own door. The fact is, the 1969 and 1970 papers, brilliantly innovative though they were, did suffer from a number of defects, some of which I propose to examine in what follows. And it's at least plausible to suggest that some of the sins to be observed in the present database landscape—sins of both omission and commission—can be regarded, with hindsight (always perfect, of course), as deriving from the defects in question.

Before going any further, I'd like to make it clear that the criticisms in what follows are all offered in, and should be construed in, the spirit of this paper's epigraph. I admire and am hugely grateful for the work Codd did in the late 60s and early 70s and documented in his 1969 and 1970 papers (as well as in certain subsequent papers, also published in the early 70s). Indeed, I owe my very livelihood to him and that work! But I also feel it's important not to be blinded by such feelings into uncritical acceptance of everything Codd said or wrote, either at that time or subsequently. Nor do I feel it appropriate to accept Codd as the sole authority on relational matters. Indeed, it's to his credit that Codd himself is on record as agreeing with this position. In an interview in *Data Base Newsletter 10*, No. 2 (March 1982), he stated explicitly that "I see relational theory as simply a body of theory to which many people are contributing in different ways."

Without further ado, then, I'd like to examine some of the defects—or what seem to me to be defects, at any rate—in Codd's early writings. I'll buttress my arguments with a variety of quotes from those writings, all of them taken from the 1970 paper unless otherwise indicated.

WHAT'S THE RELATIONAL MODEL?

It's a curious fact that, despite its title, the 1970 paper nowhere says exactly what the relational model consists of. (Actually the 1969 paper doesn't do so either.) Indeed, what it does say in this connection suggests rather strongly that it consists of a structural component only.[2] For example:

- "A model based on *n*-ary relations ... and the concept of a universal data sublanguage are introduced."

That "and" suggests rather strongly that "the universal data sublanguage"—i.e., the collection of operators, such as join—is distinct from the model as such. (Moreover, a later

[2] And this is a misconception that seems to be quite widespread and persists to this day, as the following quotes from a variety of well known database textbooks indicate: (a) "In this chapter, we first study the fundamentals of the relational model, which provides a very simple yet powerful way of representing data. We then describe three formal query languages [*viz., relational algebra, tuple calculus, and domain calculus*]" (from Abraham Silberschatz, Henry F. Korth, and S. Sudarshan, *Database System Concepts*, McGraw-Hill, 4th edition, 2002); (b) "Chapter 5 The Relational Data Model and Relational Database Constraints ... Chapter 6 The Relational Algebra and Relational Calculus" (from Ramez Elmasri and Shamkant B. Navathe, *Fundamentals of Database Systems*, Addison-Wesley, 4th edition, 2004); (c) "This chapter presents two formal query languages [*viz., relational algebra and relational calculus*] associated with the relational model" (Raghu Ramakrishnan and Johannes Gehrke, *Database Management Systems*, McGraw-Hill, 3rd edition, 2003). And I could cite numerous further examples—including, I'm sorry to say, somewhat similar text from the first two editions of my own book *An Introduction to Database Systems* (Addison-Wesley, 1975 and 1977).

remark in the same paper makes it quite clear that Codd isn't just talking about syntax here—he means an abstract language.)

- "[The model] provides a means of describing data with its natural structure only ... Accordingly, it provides a basis for a high level data language ... ."

These two sentences taken together again suggest rather strongly that the model and the operators are different things.

- "The adoption of a relational model of data, as described above, permits the development of a universal data sublanguage based on an applied predicate calculus."

Once again there seems to be an implication that the model is concerned with structure only.

- "The relational view (or model) of data described in Section 1 ... ."

That "relational view (or model)" is indeed described in Section 1 as claimed.[3] However, it isn't properly *defined* there. Indeed, not only is it not defined there, but all that section does in this connection is describe the structural features of the model (i.e., relations as such)—it has nothing to say about either operators or integrity constraints, except for a brief discussion of keys and foreign keys. (As an aside, I note that the fact that keys and foreign keys *are* briefly discussed in this context might be one of the reasons why, in practice, key and foreign key constraints are typically bundled in with structural definitions—as in SQL, for example—instead of being treated in the same uniform manner as integrity constraints in general.)

To pursue the point a moment longer, what *is* defined in Section 1 is the term *relation*. However, given that (a) that definition actually occurs in subsection 1.3 of that section, (b) that subsection is titled "A Relational View of Data," (c) that term *view* (as used by Codd here) has already been equated with the term *model*, and (d) that subsection 1.3 makes no mention of the operators, the clear implication once again is that the operators aren't part of the model.

Note: The first of Codd's papers to contain an actual definition of the relational model was "Extending the Database Relational Model to Capture More Meaning" (*ACM TODS 4*, No. 4), which didn't appear until December 1979. What this latter paper said was this (paraphrasing somewhat):

- "The relational model consists of (1) a collection of time-varying tabular relations, (2) the entity and referential integrity rules, and (3) the relational algebra."

This definition might be criticized on a variety of grounds—*tabular* in particular is a little odd—but at least it does make it clear that the operators are included. As for that *time-varying*, see later in the present paper.

[3] I note in passing that the text quoted actually *appears* in Section 1.

"A" Relational Model?

The 1970 paper not only fails to give a proper definition of *the* relational model, it sometimes uses the term "*a* relational model" to refer to some user's perception of some specific database (hence, perhaps, the modern and continuing confusion over the two quite distinct meanings of the term *data model*). A couple of examples:

- "There are usually many alternative ways in which a relational model may be established for a data bank."

- "To sum up, it is proposed that ... users should interact with a relational model of the data consisting of a collection of time-varying [relations]."

Incidentally, that phrase "consisting of" suggests once again that the—or a?—relational model is concerned with structure only.

WHAT'S A RELATION?

Following on from the previous section, I now observe that the 1970 paper isn't even totally clear on what it means by the term *relation*. Subsection 1.3 ("A Relational View of Data") begins thus:

- "The term *relation* here is used in its accepted mathematical sense. Given sets S_1, S_2, ..., S_n (not necessarily distinct), *R* is a relation on these *n* sets if it is a collection of *n*-tuples each of which has its first element from S_1, its second element from S_2, and so on. We shall refer to S_j as the *j*th *domain* of *R*."

Note in particular that the "domains" of any given relation—more on that topic in a few moments—are here quite explicitly considered to have an ordering ("left to right").[4] Now, the paper does subsequently go on to say that "users [should] deal, not with relations which are domain-ordered but with *relationships*, which are their domain-unordered counterparts." But almost all of the subsequent discussions in the paper (in several later writings too), as well as the title of the 1970 paper and indeed the very term *relational model*, emphasize relations as such, not those "relationships" or "domain-unordered counterparts" to relations. Accordingly, it's hard to escape the conclusion that one of the biggest flaws to be observed in SQL today—namely, that its tables have a left to right ordering to their columns—has its origin in Codd's 1970 paper.

Here are some further quotes from the 1970 paper that have to do with what exactly a relation is:

- "[See the] remarks below on domain-ordered and domain-unordered relations."

But the paper has already stated explicitly that relations are domain-ordered by definition. As far as that paper is concerned, therefore, *domain-unordered relation* is, or should be, simply a contradiction in terms.

---

[4] Note too the tacit assumption (made explicit later in other writings by Codd) that every relation has at least one domain, and hence that the important relations TABLE_DUM and TABLE_DEE are excluded.

- "Consider an example [involving a relation] called *part* which is defined on the following domains ... (5) quantity on hand, (6) quantity on order."

"Quantity on hand" and "quantity on order" would surely be distinct *attributes* defined on the *same* domain. Indeed, the 1970 paper very frequently uses *domain* when *attribute* would clearly be correct, or at least much more appropriate. Unfortunately, however, it never mentions the term *attribute* at all—at least, not as a component of a relation.[5] (As a matter of fact, it never mentions the term *tuple* as an abbreviation for *n*-tuple, either. Taken together, these omissions are more than a little surprising, since relations in the relational model are today universally understood to consist specifically of attributes and tuples.)

- "... simple domains—domains whose elements are atomic (nondecomposable) values."

The 1970 paper fails to define the term *atomic value* adequately.[6] This failure led to a massive misunderstanding in the database community at large as to what exactly it means to say a relation is in first normal form—a misunderstanding that persists, widely, to the present day. Indeed, it's not unreasonable to suggest that Codd himself might have been a little confused over this issue, to judge not only by the 1970 paper but also by certain of his later writings.

- "Nonatomic values can be discussed within the relational framework. Thus, some domains may have relations as elements."

Two questions here. First, what about other "nonatomic" values, such as lists or sets? Are they permitted? Second, are relations allowed to contain such "nonatomic" values or aren't they? The 1970 paper never really gives a clear, straightforward answer to this question.[7]

- "For expository reasons, we shall frequently make use of an array representation of relations, but it must be remembered that this particular representation is not an essential part of the relational view being expounded."

First of all, it's interesting to see that Codd suggests representing relations as arrays, not tables (in fact, it's a little surprising to find that the term *table* doesn't occur in the paper at all). Second, to suggest that relations might be thought of in terms of arrays is really rather strange—not to say actively misleading—given that it's a sine qua non of arrays that they have both an ordering to their rows and an ordering to their columns, and relations have nothing analogous to either. As a consequence, Codd's caveat, to the effect that the array representation "is not an essential part of the relational view being expounded," seems to me much too weak. What *is* an

---

[5] Oddly enough, the 1969 paper does mention the domain vs. attribute distinction.

[6] The closest it gets to such a definition is in the following loose characterization of the distinction between what it calls"simple" and "nonsimple" domains: "The terms attribute and repeating group in present database terminology are roughly analogous to simple domain and nonsimple domain, respectively." That being said, however, it seems virtually certain that what Codd meant by "atomic values" was nothing more nor less than what in programming language circles are referred to as *scalar* values. But as I've had occasion to remark elsewhere—see, e.g., my book *SQL and Relational Theory: How to Write Accurate SQL Code* (O'Reilly, 2nd edition, 2012)—"scalarness" isn't an absolute. It isn't even formally defined (indeed, the very same value might be regarded as scalar in some contexts and nonscalar in others). Now, it would surely be unwise to require the formal relational model to rely on such a fuzzy notion in any formal way; thus, if this understanding of what Codd meant by "atomic values" is in fact correct, then I think it must be rejected.

[7] It's relevant to note that the 1969 paper explicitly does allow relations to contain such values.

essential part of the "view being expounded" is that the row and column ordering inherent in the suggested array representation must be explicitly ignored.

Unfortunately, the 1970 paper then goes on to make matters worse by frequently and repeatedly talking in terms of arrays and columns when what it really means is relations and attributes. For example:

- "A binary relation has an array representation with two columns. Interchanging these columns yields the converse relation."

No, it doesn't—it yields *an array representation* of "the converse relation" (not to mention that the very term *converse relation* is meaningless in the context of the relational model anyway!).[8] This is just one of several places where the paper itself falls into the very trap it explicitly warns us against, of confusing a relation with its array representation. No wonder so many people continue to make the same mistake to this very day.

- "A relation whose domains are all simple can be represented in storage by a two-dimensional column-homogeneous array of the kind discussed above."

Represented in storage (specifically, in the form of an array) is unfortunate, suggesting as it does a "direct image" style of implementation (which is, sadly, the style found in all mainstream SQL products today, to a first approximation). *Two-dimensional* is unfortunate too—to this day, far too many people seem to think that "two-dimensional relations" are incapable of representing "*n*-dimensional data" (or indeed most other data that occurs in practice, come to that).

- "Some more complicated [storage representation] is necessary for a relation with one or more nonsimple domains.[9] For this reason ... the possibility of eliminating nonsimple domains appears worth investigating."

The stated reason is a very bad one!—suggesting as it does that representation in storage is the major concern. Also, does "appears worth investigating" mean that "eliminating nonsimple domains" *must* be done (assuming it's possible), or are we merely talking about something that might be desirable but isn't absolutely required?

- "The simplicity of the array representation ... is not only an advantage for storage purposes ... but also for communication of bulk data between systems ... The communication form ... would have the following advantages ... :"

Again the phrase *for storage purposes* is unfortunate. (So too is the phrase *communication of bulk data between systems*, come to that.) Anyway, the first of the advantages the paper goes on to mention is: "It would be devoid of pointers." Of course, this state of affairs clearly implies that the arrays seen by the user are also devoid of pointers, but this latter fact, strangely enough, is *not*

---

[8] Note, incidentally, that Codd is quite definitely talking about "domain-ordered relations" here, not their "domain-unordered counterparts."

[9] The paper uses the term *data structure*, not *storage representation*, but it's clear from the context that storage representation is what's meant.

cited as an advantage. In other words, the crucially important fact that database relations per se are supposed to "be devoid of pointers" is nowhere spelled out explicitly.[10]

- "A first-order predicate calculus suffices if [every relation in the] the collection of relations is in [first] normal form."

There seems to be a suggestion here that second-order logic is necessary otherwise. It's not clear to me whether such is in fact the case, but I do think this remark is the source of some further confusion. And while I'm on the subject of logic, let me also point out what seems to be a fairly major omission: namely, the omission, from both the 1970 paper and its 1969 predecessor, of any mention of the crucial connection between relations and predicates.[11]

WHAT'S A DOMAIN?

I've already mentioned the 1970 paper's lack of clarity over domains vs. attributes. In fact, however, there's a further point of confusion that arises from the notion of domains as described in Codd's writings. To elaborate: As far as I and many other writers are concerned, domains in the relational world are indistinguishable from *types* in the programming languages world.[12] But the 1970 paper doesn't discuss this equivalence at all; in fact, it doesn't even mention it. Au contraire, in fact: In later writings Codd went to great lengths to argue the opposite point of view—viz., that domains and types are different things. For example, in his paper "Domains, Keys, and Referential Integrity in Relational Databases" (*InfoDB 3*, No. 1, Spring 1988), he draws a distinction between what he calls "basic data types" and "extended data types," and goes on to say that "extended data types" are domains but "basic data types" aren't. Here's a quote from that paper:

- "Each domain is declared as an extended data type (not as a mere basic data type) ... The distinction between extended ... and basic data types is NOT that the first is user-defined and the second is built into the system."

The following table from that same paper purports to summarize the differences between the two—the differences as seen by Codd, that is—though as far as I'm concerned all it manages to do is raise a host of further questions:

---

[10] Indeed, this fact doesn't seem to have been spelled out explicitly in any of Codd's writings until 1979, when the following text appeared in his paper "Extending the Database Relational Model to Capture More Meaning" (*ACM TODS 4*, No. 4, December 1979): "Between ... relations there are no structural links such as pointers. Associations between relations are represented solely by values." However, a couple of remarks can be found in his 1971 paper "Normalized Data Base Structure: A Brief Tutorial," Proc. ACM SIGFIDET Workshop on Data Description, Access, and Control (San Diego, Calif., November 11th-12th, 1971), that do at least strongly hint at such a state of affairs.

[11] The first of Codd's writings to spell out this connection explicitly seems to have been the paper I've mentioned a couple of times already, "Extending the Database Relational Model to Capture More Meaning" (*ACM TODS 4*, No. 4, December 1979).

[12] This observation does *not* apply to what SQL calls domains, which aren't domains in the relational sense at all. Rather, they're merely a kind of factored out "common column definition," with a number of rather strange properties that have nothing to do with the relational model as such and hence are beyond the scope of this paper.

#	Basic Data Type	Extended Data Type
1	property-oriented name	object-oriented name
2	a property of an object	an object
3	not independently declarable	independently declarable
4	range of values NOT specifiable	range of values specifiable
5	applicability of >, < not specifiable	applicability of >, < specifiable
6	two database values with the same basic data type need not have the same extended data type	

A detailed discussion of these alleged differences would be out of place in the present paper; suffice it to say that all that's happening here is that Codd is confusing types and representations (or so it seems to me, at any rate).[13] Further evidence in support of this claim is provided by several further remarks in that same *InfoDB* paper. I'll content myself with quoting just one of those remarks here:

- "With special authorization ... a user may employ the DOMAIN CHECK OVERRIDE qualifier in his [*sic*] command, if a special need arises to 'compare apples with oranges'."

The "apples and oranges comparisons" Codd is referring to here are comparisons involving domains that are (by definition) logically distinct but share the same physical representation—for example, a comparison to see whether the physical representation of a certain supplier number is the same as that of a certain part number. *Note:* In other writings, Codd (rather unfortunately, in my opinion) used the term "semantic override" in place of "domain check override." The term "command" is unfortunate too—"expression" would be much better.

Actually, there's at least one remark in the 1970 paper that does touch on these matters:

- "It is a remarkable fact that several existing information systems ... fail to provide data representations for relations which have two or more identical domains."

This remark is clearly incorrect on its face; a relation with, e.g., two attributes both defined on the domain of integers surely can't give rise to any special difficulties of implementation. From the context, however, it seems that what Codd was really getting at here is that systems often provide special support for one to many relationships between "entity types" but fail to provide special support for many to many relationships between such "entity types." This criticism might apply to XML, for example, though of course XML didn't exist in 1970. Incidentally, it's interesting to note that, by contrast, it doesn't apply to IMS (which did exist at the time), despite the fact that IMS is usually perceived as being a hierarchic system.[14]

[13] The significance of this important logical difference (i.e., between types and representations) is nicely captured in the following quote from "On Understanding Types, Data Abstraction, and Polymorphism" (*ACM Comp. Surv. 17*, No. 4, December 1985), by Luca Cardelli and Peter Wegner: "A major purpose of type systems is to avoid embarrassing questions about representations, and to forbid situations in which these questions might come up."

[14] And despite the fact also that the very next sentence in the 1970 paper, immediately following the sentence already quoted, is: "The present version of [IMS] is an example of such a system"!

WHAT'S A TIME-VARYING RELATION?

Ever since the early 1990s, Hugh Darwen and I have been calling attention to the fact that there's a logical difference between relations as such (meaning relation values), on the one hand, and relation variables (which we abbreviate to just *relvars*) on the other. Codd's 1970 paper uses the qualifier *time-varying* in an attempt to get at the same distinction. For example:

- "The totality of data in a data bank may be viewed as a collection of time-varying relations."

However, it's my belief that the phrase *time-varying relation* has been the source of a very great deal of confusion. Since relations simply don't vary with time (as indeed the definition of the term *relation* in the first paragraph of subsection 1.3 of the 1970 paper makes quite clear), it follows that a "time-varying relation," whatever else it might be, certainly isn't a relation. In particular, therefore, given that the operators—join, projection, and so on—defined elsewhere in the paper all apply to relations as such, what does it mean to apply them to a "time-varying relation"? Note that Codd certainly does assume they do apply, as is evidenced by several remarks in Section 2.2 ("Redundancy") of the 1970 paper and elsewhere.

- "As time progresses, each *n*-ary relation may be subject to insertion of additional *n*-tuples, deletion of existing ones, and alteration of components of any of its existing *n*-tuples."

"Relation" here ought really to be "time-varying relation" or (better) "relation variable" (variables can be updated, values can't). Also, the 1970 paper nowhere mentions the fundamental operation of relational assignment, an oversight that perhaps led to the omission of that operator from SQL, as well as from most other proposed relational or would-be relational languages. As I've had occasion to remark elsewhere, INSERT, DELETE, and UPDATE are convenient shorthands, but they're all, in the final analysis, defined in terms of relational assignment. And a model as such—which after all is what the 1970 paper is supposed to be all about—ought surely to be concerned with fundamentals, not with mere shorthands that might happen to be convenient for the user.

Note: While I'm on the subject of updates, the 1970 paper also has this to say:

- "Deletions which are effective for the community (as opposed to the individual user or subcommunities) take the form of removing elements from declared relations."

It's frankly not clear what this remark is supposed to mean (especially given the qualification in parentheses)—it looks a little mysterious to me. As far as I know, however, it didn't lead to anything, so perhaps we can simply ignore it.

WHAT'S A KEY?

The 1970 paper is quite muddled over the concept of keys in general and primary keys in particular. Consider the following:

- "Normally, one domain (or combination of domains) of a given relation has values which uniquely identify each element (*n*-tuple) of that relation. Such a domain (or combination) is called a *primary key* ... A primary key is *nonredundant* if it is either a simple domain (not a combination) or a combination such that none of the participating simple domains is superfluous in uniquely identifying each element. A relation may possess more than one nonredundant primary key ... Whenever a relation has two or more nonredundant primary keys, one of them is arbitrarily selected and called *the* primary key of that relation."

  I have several comments on this extract.

1. First of all, keys of any kind represent certain integrity constraints, and integrity constraints apply to variables, not values (since they constrain the effects of updates, and updates apply to variables, not values). Thus, "relation" should be "time-varying relation" or (better) "relation variable" throughout.

2. "Domain" should be "attribute" throughout.

3. The term *simple domain* doesn't mean here what it means elsewhere in the paper (viz., a domain whose values are "atomic").

4. The "combination" consisting of no attributes at all should be allowed. *Note:* Of course, this possibility isn't explicitly excluded by the text quoted. But the term "combination" (used here and elsewhere in the paper), rather than the term "set," does tend to suggest that the intended interpretation is "one or more," not "zero or more."[15]

5. That opening "Normally" is quite puzzling—it suggests that what the paper calls a primary key is in fact optional.[16] Worse, it actually suggests that "a given relation" might not have a "domain (or combination of domains) which uniquely identify each [tuple]" at all, and hence that duplicate tuples might be legitimate! Of course, I'm quite certain this interpretation isn't what Codd intended, but it would surely have been better to have used more careful wording. As it is, he lays himself open to attack by critics of the deconstructionist persuasion.[17]

6. Note that the extract allows a "relation" to have any number of primary keys, and moreover that such keys are allowed to be "redundant" (better: *reducible*). In other words, what the paper calls a primary key is what later (and better) became known as a *superkey*, and what the paper calls a nonredundant (better: *irreducible*) primary key is what later became known as a *candidate* key or (better) just a *key*.

---

[15] Further evidence that this interpretation is the one intended can be found in various later writings by Codd. For example, in his book *The Relational Model for Database Management Version 2* (Addison-Wesley, 1990), we find this: "A primary key may consist of a simple column or a combination of columns. When it consists of a combination of columns, the key is said to be *composite*." It's hard to believe that Codd intended that a key of degree zero should be thought of as composite.

[16] But the paper does later say "Each [relation] declaration ... identifies the primary key for that relation," which suggests that primary keys are mandatory after all.

[17] Deconstruction in general is a useful and effective technique of literary criticism. This isn't the place for a detailed explanation, but what it boils down to is this: You can judge a writer's intent only by what he or she has actually said, not by what you might possibly think he or she might possibly have wanted to have possibly said, but didn't.

7. Finally, "*the* primary key" as defined in the final sentence of the extract quoted is indeed what the term *primary key* later came to denote. But I reject that concept anyway. I don't believe the kind of arbitrariness involved (i.e., in selecting the primary key) has any place in a formal system such as the relational model is supposed to be.

Here's another quote on primary keys (though actually what it says is surely intended to apply to keys in general, not just to primary keys in particular):

- "No primary key has a component domain which is nonsimple."

And the paper goes on to say "The writer knows of no application" that would require this condition to be relaxed. Well, we could perhaps argue over the precise meaning of the term *require* here; however, there are certainly applications for which keys defined on "nonsimple domains" do seem to be the most natural design.[18]

So much for keys as such; now I turn to foreign keys. I have just one quote on this subject:

- "We shall call a domain (or domain combination) of relation *R* a *foreign key* if it is not the primary key of *R* but its elements are values of the primary key of some relation *S* ... ."

Foreign keys were invented by Codd, but his definition of the concept changed several times over the years.[19] The first definition, in the 1970 paper, was the one just quoted. That definition includes the strange and clearly unnecessary restriction that a foreign key not be the primary key of its containing relation (or relation variable, rather). Codd later and silently dropped that restriction, but he never dropped the restriction that the target of a foreign key had to be a primary key specifically.[20]

WHAT ABOUT THE OPERATORS?

Although (as I've tried to show) the 1970 paper is at best unclear as to whether the relational operators are part of the model—not to mention whether they apply to "time-varying relations"—it does have a lot to say about such operators. With hindsight, however, it seems to me that some of the things it does say are a little strange. For example:

- "Since relations are sets, all of the usual set operations are applicable to them. Nevertheless, the result may not be a relation; for example, the union of a binary relation and a ternary relation is not a relation."

These remarks are undoubtedly true, but they constitute the sole mention of "the usual set operations" (union in particular) in the entire paper. The clear implication is that (a) union and the rest are included in the proposed set of operators, and (b) there's no requirement for what

---

[18] Examples of such applications can be found in the paper "What First Normal Form Really Means," in my book *Date on Database: Writings 2000-2006* (Apress, 2006).

[19] An annotated history of those changes can be found in the paper "Inclusion Dependencies and Foreign Keys," in the book *Database Explorations: Essays on The Third Manifesto and Related Topics* (Trafford, 2010), by Hugh Darwen and myself.

[20] To its credit, SQL never abided by either of these restrictions.

Codd referred to in later papers as *union compatibility*.[21] A further implication is that there's no requirement for what later came to be called *closure* (relational closure, that is, meaning that the result of every relational operation is itself a relation). Indeed, this latter notion—which later came to be regarded as crucial—isn't mentioned in the paper at all. (It might be germane to mention here that SQL as originally defined in fact failed to abide by the closure requirement, a shortcoming that wasn't corrected, at least in the standard version of that language, until 1992.)

- "*Join*. Suppose we are given two binary relations, which have some domain in common. Under what circumstances can we combine these relations to form a ternary relation which preserves all of the information in the given relations?"

The join concept is in many ways one of Codd's most important contributions. It's unfortunate, therefore, that—in my opinion, at any rate—the discussion of that topic in the 1970 paper is a little confusing:[22] so much so, in fact, that it can be quite hard in places to see the forest for the trees. For example, it might not be immediately obvious to a modern reader—though the foregoing quote, which opens the discussion, does suggest as much—that the paper is primarily, albeit not exclusively, concerned with what we would now call *nonloss* joins.[23] Nonloss joins are crucially important in what's now called normalization theory, but that theory isn't part of the relational model; rather, it's a separate theory that's built on top of the relational model.[24] To put the point another way: Join as such is just an operator. Whether some particular join is nonloss is a separate question, one that's significant in certain important contexts but is, or should be, irrelevant as far as the relational model itself is concerned.

- "A binary relation *R* is *joinable* with a binary relation *S* if there exists a ternary relation *U* such that [the projection of *U* on its first and second attributes is equal to *R* and the projection of *U* on its second and third attributes is equal to *S*].[25] Any such ternary relation is called a join of *R* with *S*."

Observe that, quite apart from the fact that it assumes that relations have a left to right ordering to their attributes,[26] this definition implies the following: If *X* is the second attribute of relation *R* and *Y* is the first attribute of relation *S* (and if those two attributes are defined on the same domain, of course), then *R* and *S* can be joined *only if the set of X values in R is the same as the set of Y values in S* (equivalently, only if the projections of *R* and *S* on *X* and *Y*, respectively,

---

[21] In fact I reject "union compatibility" as such anyway. In our work on *The Third Manifesto*—see our book *Databases, Types, and the Relational Model: The Third Manifesto* (Addison-Wesley, 3rd edition, 2007)—Hugh Darwen and I replaced, and subsumed, that notion by a carefully thought out notion of *relation type*.

[22] The discussion in the 1969 paper is essentially identical.

[23] Actually the term *nonloss join* has at least two different meanings. In the present paper I use it in the following sense (and the following sense only): The join *j* of relations *r1* and *r2* is nonloss if and only if the projection of *j* on the attributes of *r1* is equal to *r1* and the projection of *j* on the attributes of *r2* is equal to *r2*.

[24] It is, however, only fair to mention that it was Codd himself who established the foundations of normalization theory (now more commonly known as dependency theory). See his paper "Further Normalization of the Data Base Relational Model," in Randall J. Rustin (ed.), *Data Base Systems: Courant Computer Science Symposia Series 6* (Prentice-Hall, 1972).

[25] Our modern understanding of the term is rather different; today we would say that two relations are joinable if and only if attributes with the same name are of the same type.

[26] And quite apart also from the fact that it implies that *R* and *S* might be joinable while *S* and *R* aren't—and even if they are, that the join of *R* and *S* and the join of *S* and *R* will, in general, be distinct (as indeed the 1970 subsequently and explicitly admits).

are equal).[27] Oddly, however, the paper then goes on to give a definition of the natural join of *R* and *S* that doesn't require *R* and *S* to be "joinable" in the foregoing sense!—indeed, that definition is close to the one we use today.

The paper then gets into a fairly extensive discussion of what it calls *cyclic* joins. But that discussion is back to front, in a sense. What it really has to do with is not so much join as such but what we now know as *join dependencies*, and in particular with the possibility that a given relation might be nonloss decomposable into (say) three projections but not into two. Indeed, the very fact that no one at the time seemed to realize that such matters were the real topic lends weight to my claim that the discussion overall wasn't as clear as it might have been. Be that as it may, the paper gives an example, in Figs. 8 and 9, to illustrate the foregoing possibility (the possibility, that is, that a relation might be nonloss decomposable into three projections but not into two). Unfortunately, however, Fig. 8 contains at least one mistake, which makes it difficult to understand exactly what's going on. At the very least, the fourth row of T in Fig. 8 should either be changed (but how?) or be deleted.[28]

- "[It] appears that an adequate collection [of operators is] projection, natural join, tie, and restriction." *Note:* Codd's restriction operator here isn't quite the same as the operator of that name as now understood, but the differences aren't important for present purposes. As for the tie operator, it can be defined as follows: Given a relation *r* with attributes *A1*, *A2*, ..., *An*, the *tie* of *r* returns the relation containing just those tuples of *r* for which *A1* = *An*. (Incidentally, note the reliance on left to right attribute ordering in this definition.)

Here Codd is touching on what would later come to be called *relational completeness*. But that "adequate" list of operators should certainly include (relational) union and difference. (By contrast, it's hard to see why tie is included, or even what purpose that operator serves at all.)

- "Arithmetic functions may be needed in the qualification or other parts of retrieval statements. Such functions can be defined in [the host language] *H* and invoked in [the data sublanguage] *R*."

It's good to see that the 1970 paper tacitly endorses the idea of supporting relational operators such as EXTEND and SUMMARIZE. Unfortunately, the paper fails to give further guidance as to how those "arithmetic functions" might actually be "invoked in *R*" (the relational operators defined in the paper don't seem to have room for any such invocations).

By the way, the terms *arithmetic*, *retrieval*, and *statements* in the foregoing quote are all somewhat misleading, in my view. For my part, I would greatly prefer to replace the first sentence by something along the following lines: "Relational expressions might need to contain invocations of computational functions" (and that qualifier *computational* might not be necessary, either).

---

[27] This condition is sufficient to guarantee that the join is nonloss, in the sense in which I'm using the term *nonloss join* in this paper.

[28] Oddly enough, the counterpart to Fig. 8 in the 1969 paper does appear to be correct (to be specific, it omits that fourth row of T). On the other hand, at least one republished version of the 1970 paper (viz., the one in Phillip LaPlante, *Great Papers in Computer Science*, West Publishing, 1996) manages to introduce an additional mistake of its own. *Caveat lector*.

- "It is well known that ... it is unnecessary to [be able to] express every formula of the selected predicate calculus [in the data sublanguage]. For example, just those in prenex normal form are adequate."

  Actually prenex normal form is *not* adequate, as I've shown elsewhere.[29]

- "The network model ... has spawned a number of confusions, not the least of which is mistaking the derivation of connections for the derivation of relations." [*Later:*] "A lack of understanding of relational composition has led several system designers into what may be called the *connection trap*.[30] This trap may be described in terms of the following example. Suppose each supplier description is linked by pointers to the descriptions of each part supplied by that supplier, and each part description is similarly linked to the descriptions of each project which uses that part. A conclusion is now drawn which is, in general, erroneous: namely that, if all possible paths are followed from a given supplier via the parts he [*sic*] supplies to the projects using those parts, one will obtain a valid set of all projects supplied by that supplier."

Far be it from me to defend "the network model," but this criticism really has nothing to do with the network model as such—it applies equally to the relational model, mutatis mutandis.[31] "Derivation of connections" *is* "derivation of relations"! The mistake consists in misinterpreting the relations so derived. It's the quoted text that's confused.

And What About Relational Comparisons?

The algebra of sets is usually thought of as including a partial ordering operator called set inclusion, denoted by the symbol "⊆". Here's the definition: The expression *s1* ⊆ *s2*, where *s1* and *s2* are sets, evaluates to TRUE if and only if every element of *s1* is also an element of *s2* (i.e., if and only if *s1* is a subset of *s2*). So if, as seems likely, Codd meant to pattern his algebra of relations after the algebra of sets, it would have been reasonable to define an analogous relational inclusion operator. More generally, it would have been reasonable, and indeed useful, to define a full array of relational comparison operators: equality, inclusion, proper inclusion, and so on. Sadly, however, neither the 1970 paper nor its 1969 predecessor mentioned any such operators (at least, not explicitly).[32]

Note: Actually the foregoing omission is a trifle odd, inasmuch as the discussion of redundancy in both papers certainly assumed the ability to compare two relations for equality if

---

[29] See the paper "A Remark on Prenex Normal Form," in the book *Database Explorations: Essays on The Third Manifesto and Related Topics* (Trafford, 2010), by Hugh Darwen and myself.

[30] Relational composition is a generalization of conventional functional composition. In its simplest form it can be defined as follows: Let relation *r1* have attributes *A* and *B* and let relation *r2* have attributes *B* and *C*, and let attributes *r1.B* and *r2.B* be of the same type (i.e., let *r1* and *r2* be joinable, either in Codd's sense or as that term is now understood). Then the composition of *r1* and *r2* is the join of *r1* and *r2*, projected on *A* and *C*.

[31] By the way, Codd really shouldn't have used the phrase "the network model" here if, as he later argued, no abstract network model even existed at the time. See his remarks on such matters in his paper "Data Models in Database Management" (*ACM SIGMOD Record 11*, No. 2, February 1981).

[32] Codd did subsequently require such operators to be supported in what he called Version 2 of the model—see his book *The Relational Model for Database Management Version 2* (Addison-Wesley, 1990)—but he didn't discuss them in detail or even define them. (The pertinent text, on page 365 of that book, reads in its entirety thus: "[The relational language] also includes set comparators such as SET INCLUSION.")

nothing else. What's more, the definitions Codd himself gave for his relational division operator in various later publications all explicitly invoked the operation of relational inclusion! Be that as it may, the fact that relational comparisons are still to this day omitted from nearly all database textbooks (and from the SQL language as well, come to that) can, I think, fairly be traced back to the lack of any mention of such operators in Codd's first two papers.

WHAT ABOUT DATA INDEPENDENCE?

Here's the opening sentence from the abstract to the 1970 paper:

- "Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation)."

I agree strongly with this position, implying as it does that there must be a rigid distinction between model and implementation. But several later remarks in the paper blur that distinction considerably. For example, subsection 1.6 is titled "Expressible, Named, and Stored Relations." Clearly, any notion of some given relation being stored—i.e., *physically* stored, in some kind of "direct image" form, with each stored record corresponding to a single tuple from the relation in question[33]—is counter to the objective of users being "protected from having to know how the data is organized in the machine." In fact, subsection 1.6 nowhere discusses "stored relations" anyway! The title is presumably a hangover from the 1969 paper, where such a notion *was* discussed. (Here's a quote from that earlier paper: "The large, integrated data banks of the future will contain many relations of various degrees *in stored form*"—emphasis added.) Indeed, the whole idea of some relations being "stored" ones was a mistake in that earlier paper, a mistake partially but not totally excised from the 1970 paper. For example, here are some more quotes from this latter:

- "[The host language] permits declarations which indicate ... how these relations are represented in storage."

- "Once a user is aware that a certain relation is stored ..."

- "... [then that user will] expect to be able to exploit it using any combination of its arguments as 'knowns' and the remaining arguments as 'unknowns' ... This is a system feature ... which we shall call ... *symmetric exploitation* of relations. Naturally, symmetry of performance is not to be expected."

Actually there's no a priori reason why symmetry of performance shouldn't be expected, unless—as the extract quoted here does tacitly seem to assume—we're dealing with a "direct image" style of implementation (which is indeed, as noted earlier, what we do find in most if not

[33] One reviewer objected here that Codd's talk of stored relations didn't necessarily imply a "direct image" style of implementation (especially since the very next bullet item—another quote from the 1970 paper—in the present paper explicitly refers to "declarations which indicate how relations are represented in storage," suggesting some degree of choice and variability in physical representation). But other remarks of Codd's, quoted elsewhere in the present paper, do strongly suggest a direct image style. And in any case (and more to the point), that very phrase *stored relation* implies—at least to me—that there's some kind of one to one mapping between relations and storage structures, and in my view the assumption of such a mapping has already prejudiced the debate. In my book *Go Faster! The TransRelational™ Approach to DBMS Implementation* (2002, 2011, *http://bookboon.com*), I argue strongly that it's a good idea that there not necessarily be any such one to one mapping.

all mainstream SQL products today). Thus, once again we find the 1970 paper tacitly endorsing the notion that certain relations will be physically stored as such (or, rather, the notion that their array representation will be physically stored as such). *Note:* By the way, "arguments" in the foregoing quote ought really to be *parameters* or (perhaps better) *attributes*.

As an aside, I note that the paper goes on to say "To support symmetric exploitation ... [for] a relation of degree *n*, the number of [access] paths to be named and controlled is *n* factorial." It's a little hard to argue with this claim since the term *access path* isn't defined, but shouldn't the number of paths be 2^n, not *n* factorial? (There are 2^n combinations of attributes that might be used as "knowns.")

WHAT ABOUT INTEGRITY?

The 1970 paper has quite a lot to say about integrity. However, it mostly doesn't use that term, it uses the term *consistency* instead—not unreasonably, because the only integrity constraints it considers in any detail[34] are ones having to do with data redundancy specifically, as the following quote (from subsection 2.3 of the paper, "Consistency") makes clear:

- "Whenever the named set of relations is redundant ... we shall associate with that set a collection of statements which define all of the redundancies which hold independent of time between the member relations[35] ... Given a collection of *C* of time-varying relations, an associated set *Z* of constraint statements, and an instantaneous value *V* for *C*, we shall call the state (*C*, *Z*, *V*) *consistent* or *inconsistent* according as *V* does or does not satisfy *Z*."

There's just one issue—a very important one!—arising in connection with this topic that I'd like to comment on here. Here first is the relevant extract from the paper:

- "There are, of course, several possible ways in which a system can detect inconsistencies and respond to them. In one approach the system checks for possible inconsistency whenever an insertion, deletion, or key update occurs.[36] Naturally, such checking will slow these operations down. If an inconsistency has been generated ... and if it is not remedied within some reasonable time interval, either the user or someone responsible for the ... integrity of the data is notified. Another approach is to conduct consistency checking as a batch operation once a day or less frequently."

I regard these remarks as unfortunate in the extreme. Indeed, I regard them as the source of a serious mistake to be observed in SQL and elsewhere: namely, the decision to allow the checking of certain integrity constraints to be "deferred," typically to the end of the transaction (i.e., "COMMIT time"). In strong contrast, it's my very firm position that all integrity checking needs to be "immediate" (i.e., done whenever an update is requested that might cause the

---

[34] Apart from key and foreign key constraints, discussed earlier in both this paper and Codd's 1970 paper.

[35] It's a sad comment on the state of the database industry that, to this day, no mainstream product actually provides this functionality.

[36] I don't know why Codd singles out key updates specifically here; surely integrity checking should be performed on *all* updates (all pertinent updates, at any rate)?

constraint in question to be violated, at the time the request in question is made). Let me elaborate:

- First of all, to say the database is consistent merely means, formally speaking, that it conforms to all declared integrity constraints. But it's crucially important that databases *always* be consistent in this sense; indeed, a database that's not consistent in this sense, at some particular time, is like a logical system that contains a contradiction. And in a logical system with a contradiction, you can prove anything; for example, you can prove that 1 = 0. What this means in database terms is that if the database is inconsistent in the foregoing sense, you can never trust the answers you get to queries (they may be false, they may be true, and you have no way in general of knowing which they are); all bets are off. As far as declared constraints are concerned, in other words, the system simply *must* do the checking whenever a pertinent update occurs; there's no alternative, because (to say it again) not to do that checking is to risk having a database for which all bets are off. In other words, immediate integrity checking is logically required.

- What's more, I don't agree that immediate checking will necessarily slow the system down. If the user has bothered to declare the constraint, presumably he or she wants it enforced—for otherwise there's no point in declaring it in the first place. And if the user wants the constraint enforced, and if the system isn't going to do it (by which I mean do it properly, by which I mean doing immediate checking on all pertinent updates), then the user is going to have to do it instead. Either way, the checking does have to be done. And I would hope that the system could do that checking more efficiently than the user! Thus, I think that, far from the operations being slowed down, they should be speeded up, just so long as the system does the right thing and shoulders its responsibility properly.

Now, to be charitable, what I think Codd might have been getting at by his suggestion that integrity checking might be deferred was what has since come to be known as "eventual consistency." But if so, then I think he was confusing consistency in the formal sense and consistency as conventionally (and informally) understood—meaning consistency as understood in conventional real world terms, outside the world of databases. Suppose there are two items *A* and *B* in the database that, in the real world, we believe should have the same value. They might, for example, both be the selling price for some given commodity, stored twice because stored data replication is being used to improve availability. If *A* and *B* in fact have different values at some given time, we might certainly say, informally, that there's an inconsistency in the data as stored at that time. But that "inconsistency" is an inconsistency as far as the system is concerned only if the system has been told that *A* and *B* are supposed to be equal—i.e., only if "*A* = *B*" has been stated as a formal integrity constraint. If it hasn't, then (a) the fact that $A \neq B$ at some time doesn't in and of itself constitute a consistency violation as far as the system is concerned, and (b) importantly, the system will nowhere rely on an assumption that *A* and *B* are equal. In other words, if all we want is for *A* and *B* to be equal "eventually"—i.e., if we're content for that requirement to be handled in the application layer—all we have to do as far as the database system is concerned is omit any declaration of "*A* = *B*" as a formal constraint. No problem, and in particular no violation of the relational model.

AND WHAT ABOUT NULLS?

I've deliberately saved until last one of the biggest mistakes of all: *nulls*. One of my reasons for doing so is that the 1969 and 1970 papers actually have nothing at all to say about the issue—quite correctly, in my view, since nulls (at least in the sense in which they're usually understood, involving a foundation in *n*-valued logic for some $n > 2$) have no part to play in a formal system like the relational model, which is firmly based on conventional two-valued logic. Indeed, Codd never discussed nulls in any detail until the 1979 paper I've mentioned several times already: viz., "Extending the Database Relational Model to Capture More Meaning" (*ACM TODS 4*, No. 4). In other words, Codd's relational model did perfectly well without nulls for some ten years. I would prefer to keep it that way.

——— ♦♦♦♦♦ ———