Why Are There No Relational DBMSs?

Hugh Darwen

If [we] could learn from history, what lessons it might teach us!
—Samuel Taylor Coleridge, *Table Talk* (1831)

L'histoire n'est que le tableau des crimes et des malheurs.
—Voltaire, *L'Ingénu* (1767)

I describe the circumstances in which I obtained, in 1978, a good answer to a burning question: how can E.F. Codd's relational model of data of 1970 [2] be properly embraced by a computer language? Considering that the answer to that question (reference [12]) was publicly available in 1975, I wonder why it all went wrong and suggest some possible reasons.

Note: Three referenced papers, [6], [8], and [9], are companion papers to this one that were originally drafted as appendixes to this paper.

"If you want something state of the art, how about developing a relational database management system?"

That was my suggestion, back in 1978, in an e-mail (sometime before that term entered our lexicon!) to my colleague Vincent Dupuis in the international software development centre for IBM's Data Centre Services (DCS), which IBM offered in most countries outside of the U.S. in those days. At that time this DCS Support Centre was split between locations in London, England, for planning and Uithoorn, The Netherlands, for development; and I was in the development department. As one of our lead planners Vincent had foreseen the need for a new service, now that cheaper computers meant that fewer businesses needed to rely on time-sharing services such as IBM's. I was about to move from development to planning (temporarily, as it turned out) and I had long nurtured a dream to produce a relational DBMS, thanks to my attendance at Chris Date's course on the subject in 1972.

Anyway, as I recall, my suggestion was immediately accepted without further questions, in spite of the fact that Vincent had been calling his dream child an "advanced end user facility". He accepted the idea that an end user interface to an "advanced" central database service would indeed be an "advanced" one.

So, when I joined the planning department I was appointed to work with my senior colleague Brian Rastrick on the technical planning for the new DBMS, which eventually acquired the name Business System 12 (BS12). Brian and I had both been previously involved with various "scripting" languages (as they would nowadays be called) for use with IBM's various time-sharing services, he as a language designer primarily, I as an implementer using IBM's System/360 and later System/370 Basic Assembler Language. Our task was to come up with a rough language definition and some guidelines for its implementation. We needed some

help in that endeavour, which we sought from various IBM research projects that had been under way during the 1970s.

We were heavily motivated by the improvements we envisaged to our old scripting languages used for queries and reports, when we realized that the input file to the report generator would be replaced by the result of a relational query, possibly using joins, unions and so on. An important objective, though, was to make sure our users would still be able to generate all the reports they were already able to generate. Note carefully the separation of function between server and client in this vision: the DBMS as server evaluates the query and presents the result to the client application; the client application formats the report as specified in a simpler script devoid of "data manipulation" instructions.

We had of course studied Codd's papers [1,2] and were happy with the structural aspects of his relational model and the notion of *physical data independence* under which the DBMS would be responsible for all aspects of physical structure and keep them "under the covers". Particularly appealing to us in this connection was the absence of a defined ordering to either the attributes or the tuples of a relation. Regarding syntax for queries we had to make a choice between something like Codd's relational calculus or something based directly on his algebra. We had significant problems with both, listed in the next section.

Problems with Codd's Algebra

1. **Calculations**

   The algebra didn't cater for various kinds of calculation that were so obviously essential as to be supported by all existing report generators—for example, computing the price of a single order item from the unit price, quantity ordered and possible discounts, and adding up those prices for the invoice bottom line. Bespoke applications did the same kind of things, of course. I later learned that this problem does not arise in the mathematical theory of relations, because a function is, after all, a relation, but that treatment obviously isn't computationally feasible as the cardinality of the relation denoted by $K = N + M$, where *K, N,* and *M* stand for values of type INTEGER (in practice a finite subset of the integers), is far too big, even for modern machines. In any case Codd didn't propose such treatment and instead proposed that support for such "calculations" should be delegated to "the host language", and that didn't make good sense to us at all,[1] as it appeared to require an application to retrieve an intermediate result of a desired query, do the calculations, and possibly send the result back to the DBMS for further processing to complete the query!

   The existing report generators at the time were general-purpose application programs using DBMS[2]-provided access methods for record-by-record retrieval from the database to process an entire script, including all required calculations and filters as well as report layout specifications. Our new vision, as I stated earlier,

[1] To be fair, [2] does include this: "Arithmetic functions may be needed in the qualification or other parts of retrieval statements. Such functions can be defined in *H* and invoked in *R*", where *H* is "the host language" and *R* is "the data sublanguage", but even if we had taken note of that we wouldn't have been much the wiser, especially as to the intended roles of client and server.

[2] Yes, we had database management systems in those days—we just didn't call them that.

was for a much simplified script to be dealt with by the application, delegating all the donkey work of data access, calculations, filters and so on to the new DBMS. That simplified script would include the following specifications, to be translated into instructions for the server—the DBMS—to handle:

- an expression (a "query") denoting the relation whose tuples providing the required data would be sent to the client;
- formatting of the attribute values in those tuples; and
- the order in which the tuples are to be presented to the client.

The remaining specifications, concerning the report layout, including required control breaks,[3] would be handled at the client.

2. **Record selection**

Existing report generators supported selection of records that satisfied a given condition, a boolean expression involving comparisons on fields, possibly allowing embedded calculation (as in X + Y > 9, for example) and possibly supporting logical connectives too. We had seen Codd's "theta-select" operator and wondered why the given condition was restricted to just one simple comparison. We realized that everything expressible using logical connectives had an equivalent expression using difference and/or intersection and/or union, but we didn't think that would be acceptable to our users (and I have to admit that performance was a big concern too, as we hadn't yet grasped the possibilities for optimization that were to become available under this new methodology). Yet, strange as it may seem now, we felt slightly uneasy about taking such a bold step as to violate what we perceived as the given prescription, by allowing the use of operators other than the usual comparison operators.

3. **Attribute names**

In reference [4] Codd wrote in the introduction to Section 2, Relational Algebra:

> For notational and expository convenience, we deal with domain-ordered relations; that is, the individual domains[4] of a given relation can be identified as the first, second, third, and so on. As pointed out in [l], however, in many practical data bases the relations are of such high degree that names rather than position numbers would be used to identify domains when actually storing or retrieving information.

It was clear to us, then, that Codd was advocating the use of a name, rather than a number, to refer to an attribute of a relation. However, his definition of the algebra was unclear concerning the attribute names of relations resulting from invocation of

---

[3] In case you're unfamiliar with the term "control break", suppose records are ordered on customer number within branch. Then a control break might be required on (customer number, branch), meaning that something special is to happen whenever that combined value changes from one record to the next, such as printing the totals for that customer at that branch. A further control break might be required on just branch.

[4] By "domains" here he meant what we now call attributes, though sometimes he seemed to use the term in place of type. For example, in the same paper we see "A simple domain is a set all of whose elements are integers, or a set all of whose elements are character strings." In this paper the term domain appearing in the context of Codd's writings it means type unless explicitly stated to the contrary.

certain dyadic operators. He appeared to be leaving that as a problem for others to solve (and "fair enough", thought I at the time).

We were familiar with the use of unique names, in existing scripting languages, to identify fields in records and we assumed that the same approach was needed with relations. But Codd's cartesian product, for example, was apparently defined for all pairs of relations *r1* and *r2*. What if some attribute in *r1* had the same name as one in *r2*? Indeed, what if *r1* and *r2* were the same relation?

When we looked at Codd's calculus notation for answers to our questions, we found none regarding calculations. Moreover, such answers as we found concerning attribute names were to some extent unclear and in any case unacceptable, as I will now explain.

Questions concerning attribute names

Assume that range variable SX is defined to range over the tuples of S, and SPX over those of SP, in Chris Date's suppliers-and-parts database.

1. Consider the expression

   ```
   (SX.S#, SPX.P#) WHERE SX.S# = SPX.S#
   ```

   What are the result's attribute names? Are they S# and P#, or SX.S# and SPX.P#?

2. If the answer to Question 1 is S# and P#, then what about this one, where SY and SX both range over S:

   ```
   (SX.S#, SY.S#) WHERE SX.SNAME = SY.SNAME
   ```

   They can't both be named S#. But if they are named SX.S# and SY.S#, then surely the answer to Question 1 has to be SX.S# and SPX.P#, raising severe questions concerning usability and, indeed, implementation.

3. Regarding union, Codd's definition of union compatibility used his "notational convenience" of assuming an ordering to the attributes, requiring the operands only to be of the same degree *n* and the *j*-th attributes of the two operands (1[sic]⩽*j*⩽*n*) to be defined on the same domain[5]. So, if the *j*-th attribute of *r1* is named A and that of *r2* is named B, what's the name of the *j*-th attribute of *r1* UNION *r2*?

Regardless of the answers to Questions 1 and 2, we couldn't accept those dot-qualified names in any case. For one thing, they would be psychologically unacceptable in the cases where they would needlessly replace the unqualified names defined for the base relations (as we then called them—we now prefer the term *base* (or *real*) *relvar*, where relvar is short for relation variable as defined in [7]). For another, prefixing an existing attribute name increased its length. Complex expressions could result in an excessive number of dots, giving intolerably long names. In those days performance considerations militated against support for very long names. Some languages at the time restricted names to a maximum of eight characters. We planned on thirty-two but thought it unacceptable for users to have to anticipate the degree of expansion that might be expected in queries, and make the attribute names for base relvars short enough to allow for

[5] Here meaning type.

that expansion. (In SQL the need for multiple dots does not arise because the qualifiers do not appear in the column names of the result of a SELECT expression, but this allows a possibly intermediate result table to have more than one column of the same name. That idea was even more unacceptable to us.)

Happily, the first port of call for Brian and me was back in our home country, England, to IBM's Scientific Centre in Peterlee, just across the water from The Netherlands, where our software development laboratory was based at the time. Researchers at this Centre had developed the "Peterlee Relational Test Vehicle" (PRTV), an implementation of the language, ISBL, that they had defined themselves. Their first prototype had been developed as early as 1971 under the name IS/1—the name change to PRTV had been occasioned by an edict from on high in IBM. The theory on which this language was founded appeared in a 1975 paper, "An Algebra of Relations for Machine Computation" by P.A.V. Hall, P. Hitchcock, and S.J.P. Todd, presented at an ACM Symposium on Principles of Programming Languages, but this paper had somehow failed to come to our attention.

Stephen Todd, who was to become our main point of contact with the Scientific Centre, was away when we visited but one of the other PRTV developers, Tony Storey, talked us through ISBL's relational operators and other features; then we were given a demonstration of the product by one of its users, an agricultural researcher investigating yields of various cereal crops in various conditions. What we learned, from that visit and from subsequent consultation with Stephen Todd, provided us with *all* the answers we needed and solved *all* of the problems that had been bothering us, as I now show.

Problem 1: Calculations

ISBL has a relational operator, #, that "extends" a given relation with one or more attributes, using open expressions to specify the attribute values. For example:

```
InvoiceItem#(Cost := Qty * Price)
```

This was remarkably similar to what our old report generator supported: SET Cost = Qty * Price.

Regarding aggregations such as invoice totals, ISBL has a relational operator, $, referred to—not very soberly!—as "glump" (later in the literature as *summarization*), which partitions a given relation according to the values of zero or more specified attributes. For example, if base relvar InvoiceItem has heading {InvoiceNo, ItemNo, CustomerNo, Description, Price, Qty},[6] the extension obtained by InvoiceItem#Cost = Qty * Price can be glumped on InvoiceNo, giving a partition for each unique value of InvoiceNo appearing in that relation. An invocation of glump also specified aggregations to be computed for each partition like this:

```
(InvoiceItem#Cost := Qty * Price)$(InvoiceNo)(Total=SUM(Cost))
```

So we had good solutions to Problem 1, but it appeared that for a traditional report showing detail lines and total lines at the end of each detail section, two or more separate queries—one for the details and one for each level of control break—would be needed and their results somehow collated. That would entail sorting each result set and some kind of nested loop to match the total lines to the correct detail section. Typically more than one level of totalling

[6] Strictly, a heading is a set of attributes, where an attribute is a name paired with a type, but we allow ourselves to omit the type names when they have no bearing on the issue at hand.

was needed. The invoice totals might need to be grouped according to CustomerNo and the overall total for each customer shown in the report. And sometimes users wanted "grand totals" to be printed at the very end of the report. Existing report generators could do all this with one sort and one pass of the sorted items.

We eventually found ways to mitigate these problems, as I describe later. By mitigation, rather than total solution, I mean that we discovered how the DBMS could provide all the details and totals in a single relation and provide the results to the application in the required order. The necessity for extra passes remained but was at least handled by the DBMS, not the application—it was important to minimize traffic on the communication line between client and server (even though these ran on the same physical machine at the time).

Problem 2: Selection

ISBL's selection operator, *r*:*c*, allowed the condition *c* to be an open boolean expression of arbitrary complexity, supporting the usual logical connectives, thus providing the comfort we had nervously sought. (*Aside:* When ISBL was implemented as PRTV, the operator name ":" was accidentally changed to ";"!)

Problem 3: Attribute Names

The solutions regarding calculations and selections were pretty obvious, of course, merely showing that what we were already familiar with in our old report generator could easily be adapted to become relational operators. We had not thought of such solutions ourselves simply because we did not feel qualified to set about extending the algebra that had already received wide attention and enthusiasm. The real eye-openers for us were ISBL's solutions to the problems that had been bothering us concerning attribute names. These solutions were based on what I later in whimsical fashion called *The Marriage Principle*. Essentially, this Principle can be stated as follows:

(a) The attributes of a heading (i.e., of a tuple or a relation) have no ordering to them—something that Codd had appeared to advocate but had not fully embraced. Instead each attribute of a heading *h* has a name that is unique within *h* and a type (Codd's "domain"). Headings *h1* and *h2* are equal if and only if they are the very same heading, implying that every attribute in *h1* is an attribute of that name and type in *h2*, and vice versa.

(b) Tuples *t1* and *t2* are equal if and only if they are the very same tuple, implying that they have the same heading and the value for each attribute *a* in *t1* is also the value for *a* in *t2*.

(c) The join of tuples *t1* and *t2*, denoted $t1 \bowtie t2$, is defined only for the case where the union $t1 \cup t2$ is a tuple, in which case it is just that union. Note that wherever *a* is an attribute of both *t1* and *t2*, *a* must have the same value in both tuples, for if it didn't, then the union would contain two elements with the same name but different values, violating condition (a).

ISBL showed us how *The Marriage Principle* affected each of the operators of its relational algebra, starting with join.

ISBL defined the join, *r1*r2,* of relations *r1* and *r2*, as the relation consisting of every tuple that results from the join of some tuple in *r1* with some tuple in *r2* under *The Marriage Principle.* Note that some tuples of *r1* might not join with any tuples of *r2* even when *r2* is not

actually empty (and vice versa). Note also that attributes that are common to the headings of *r1* and *r2* thus appear only once in the result and thus condition (a) of *The Marriage Principle* is preserved. This definition of join prompted several questions from Brian and me, which I've tried to capture in the following question and answer session, representing our discussions to the best of my memory, thirty-six years on.

Q1. What about cartesian product?—considering that Codd seemed to require this operator to be defined for all pairs of relations.

Answer: It is defined only in the case where the headings of *r1* and *r2* have no attribute names in common.

Q2. Codd writes about "theta-joins", such as "less than" joins, where the joining condition, so to speak, is based on something other than equality of common attribute values. Such a join can be expressed, he shows, by applying a selection condition to the cartesian product, but the user might want to do that even when the relation operands do have common attributes. In that case the result of a cartesian product would have two attributes of the same name. What do you do about that?

Answer: We have an "attribute rename" operator, %%. This takes a single relation *r* and returns the relation *s* that differs only in the names of one or more attributes. For example, consider a relation *r1* of heading {S#, STATUS}, giving the status of each supplier S#. Then *r1*%%X := STATUS specifies a relation that is identical to *r1* except that the attribute name X is used in place of STATUS. The resulting relation *s1* thus has heading {S#, X}. Now *s1* can be joined with relation *r2* of heading {J#, STATUS}, where *r2* shows for each project J# the minimum status required for suppliers to that project. Because there are no common attributes that join yields the cartesian product of *s1* and *r2* with heading {S#, X, J#, STATUS}. The selection condition X >= STATUS can be applied to that, giving a result showing each case of supplier S# having status X at least that required for suppliers to project J#.

Q3. So how do you write that "join" of *s1* with *r2,* given that it's actually a cartesian product?

Answer: *s1*r2!* Or *r2*s1*—join is commutative (also associative), thanks to condition (a) of *The Marriage Principle*.[7] Note also that every tuple of *s1* joins with every tuple of *r2* under condition (c) of *The Marriage Principle*, so cartesian product is just a special case of join and doesn't need a name of its own. And by the way, if you want to tell the system that you really mean this join to be a cartesian product, you can specify that there are no common attributes thus: *s1*//r2*. The slashes are enclosing a list of attribute names where you can tell the system what you think are the common attributes—in this case the list is empty. The operation is rejected with a type error if "what you think" turns out to be wrong.

We were deeply impressed.

---

[7] Whether Codd's joins were commutative is a moot point, considering that his definitions depend on an ordering to the attributes. He defined a "natural join" in [2], denoted *r1*r2* as (later) in ISBL. He noted that this operator is associative but did not say anything about commutativity.

Q4. What about union, difference, and intersection? We understand that the operands have to be of the same degree, that a one-to-one correspondence between their attributes must be defined, and that corresponding attributes must be based on the same domain. What do you do about that, considering that two or more attributes of a relation might be based on the same domain?

Answer: Corresponding attributes must have the same name as well as being based on the same domain. By the way, we relax that "same degree" requirement in the case of difference. Our version of difference, *r1*-*r2*, gives the tuples of *r1* that have no matching tuples on the common attributes in *r2*, making Codd's difference just a special case. Intersection is just the special case of join where all attributes of both operands are common, but we do have special syntax for it: *r1*.*r2*, which is useful when you wish to be sure that all attributes of both operands are common.

We were even more deeply impressed. Regarding the generalization of difference, we remarked that its converse might also be handy, yielding the tuples of *r1* that *do* have matching tuples in *r2*—a sort of halfway house between intersection and join. ISBL didn't have that one but we included it in BS12 under the unfortunate name INTERSECT. In 1982, when I demonstrated BS12 to Ted Codd, I must have done a bad job of explaining INTERSECT to him, for he suggested the addition of an operator he called semijoin, which turned out to be exactly our INTERSECT. The semijoin of *r1* and *r2* is their join projected on the attributes of *r1* (so the operator is not commutative).

Q5. Talking about domains, we aren't 100% clear as to what Codd exactly means by that term. What's your take?

Answer: We think a domain is just a data type. Some are system-defined, such as those for integers, reals and character strings and we started with just those. Support for user-defined domains was a slightly later addition to PRTV. With it one could for example define domains for geometrical constructs such as points and polygons, and associated operators such as "union" of polygons and testing a point for appearing inside a given polygon.

Q6. What about the relational "divide" operator?

Answer: As it can be defined in terms of join, difference and projection, it isn't really needed.

Aside: Many years later Stephen Todd described an attempted generalization of Codd's DIVIDE, defined for all pairs of relations for which join is defined, based on *The Marriage Principle*. He gave as an illustrative example—though not a very realistic one!—PARENT_OF{*Parent, Child*} DIVIDEBY ATTENDS{*Child, School*}, whose result gives *Parent/School* pairs such that all the children of parent *Parent* attend the same school *School*, noting that the converse, ATTENDS{*Child, School*} DIVIDEBY PARENT_OF{*Parent, Child*} gives *Parent/School* pairs such that all the children attending school *School* have the same parent *Parent*. We didn't learn about this in time for BS12, in which Codd's division was shipped as a user-defined generic relational operator that all users were authorized to use. In any case, on close examination Codd's

operator was shown much later (by Chris Date) not to be a special case of Todd's after all as the semantics differed in certain boundary cases. *End of aside.*

Q7. Well, we now seem to have covered all the relational operators we've previously learned about, apart from projection, but that's pretty straightforward, isn't it?

Answer: Yes, our projection operator, %, just takes a relation *r* and a list of attribute names, each of which must refer to an attribute of *r*. For example, *r*%A,B. We allow attribute renaming to be done at the same time, so to speak: *r*%A,X := B, giving a relation of heading {A,X}.

My memory told me that we were also told that ISBL supported projection by exclusion too, but Stephen Todd tells me that it was actually our idea, resulting in a later addition to ISBL: *r*%^A,B, being projection on all the attributes of *r* apart from A and B. In this connection, shortly after Stephen Todd's visit to our development laboratory in The Netherlands for further consultation, he sent me an e-mail telling me that, in case I was wondering, it's okay if a projection results in a relation with no attributes at all! (He gives the credit for that observation to Frank Palermo, who was at IBM's research lab in San Jose, California, known for his paper [16], on which Stephen Todd collaborated. This paper was one of the earliest on relational optimization and is regarded as a classic.)

There are only two relations of degree zero, the one that contains a single tuple, the 0-tuple, and the one that's empty. So in BS12 we allowed a projection to specify no attributes, yielding one of the "twins" that later became known as TABLE_DEE and TABLE_DUM. The predicate for such a relation clearly has no free variables and therefore takes the form of a proposition, *p*. TABLE_DEE indicates that *p* is true, TABLE_DUM that it is false. Projection on no attributes is used to obtain answers to questions such as "Is supplier S1 located in Paris?" or those of the form "Are there any …?"—questions to which the answer is either "yes" or "no".

Q8. Thanks a million for all that! There's just one thing still bothering us … (and we returned to the problem I mentioned before concerning reports showing details and totals).

Answer: Why not just take the join of the "details" relation and all the "totals" relations (one for each level of control break)? In the case of your simple example involving invoices and just one control break it would be

```
(InvoiceItem#Cost = Qty * Price)$InvoiceNo#Total=SUM(Cost) *
(InvoiceItem#Cost = Qty * Price)
```

and this is a good time to show you the neat trick we have for avoiding the need to repeat that expression giving the cost of each item:

```
t1 := N!InvoiceItem#Cost := Qty * Price
t2 := N!t1$InvoiceNo#Total=SUM(Cost)
```

Those statements behave like definitions, not assignments—the N! in front of the names InvoiceItem and t1 indicates that evaluation of those right-hand sides takes place only when subsequent explicit requests for evaluation of expressions using the names t1 and t2 are given. We call this mechanism "delayed evaluation". In this example, you would give LIST(t1*t2) to obtain the required join. And if InvoiceItem is later updated, the same

request would reevaluate the join, using the new value for InvoiceItem, and give the appropriately updated result. When the DBMS is asked to evaluate t1*t2, it actually reconstructs the original expression, replacing the appearances of t1 and t2 by the right-hand sides of their definitions. The names t1 and t2 remain in scope for the whole session and can be used for various purposes, such as displaying or printing the result in tabular form, assignment to a variable, or via a SAVE command, saving the definition as a permanently named view. The step-by-step approach also allows each step to be tested in isolation, simply by displaying its result.

The answer to Q8 helped us enormously. We realized that the DBMS would still very likely be having to perform several passes over the same input but on the other hand the application, seeing the relevant totals in every tuple, would have more flexibility regarding the report format. For example, the totals could appear at the beginning of a section, if desired, instead of at the end. But our performance worries led us to probe the experts on matters of implementation and we learned of several important techniques that we brought to BS12:

- **Pipelining:**[8] A relation expression is represented internally as a tree whose root node represents the final result, intermediate nodes represent invocations of system-defined relational operators, and leaf nodes represent relations from the database or relation literals. Evaluation is done, in principle and loosely speaking, as follows. Tuples are requested, one at a time, from the root node. For example, in the tree representing (SP*(P:COLOR = 'red'))%PNO, CITY, the root node passes on each request to its immediately inferior node for the projection on PNO and CITY and the projection node passes it on to the join node, which, join being a dyadic operation, has two immediately inferior nodes: a leaf node for SP and a select node for P:COLOR = 'red'. The join node requests tuples from both of its inferior nodes, looking for pairs that match under *The Marriage Principle,* passing tuples that satisfy the match up to the requesting projection node. The select node obeys a request by requesting tuples from its leaf node, representing the relvar P, until it gets one whose COLOR value is 'red', which is passed up to the join node. When a node has returned its last tuple, it responds to the next request with a "no more" return code, and evaluation is complete when this return code is given to the root node.

- **Duplicate elimination:** I had been extremely worried about this issue, imagining that each individual operation in a complex query would entail the expensive overhead of some method, such as sorting or hashing, to test for the appearance of duplicate tuples (and eliminating them, of course!). We were assured that no operation in PRTV could ever produce a "relation" with duplicate tuples, and we were given some extremely helpful advice, as follows.

  Point 1: Assuming that leaf nodes are duplicate free (which we already knew how to achieve at little cost), the only nodes where duplicates might arise in the absence of an elimination mechanism are those representing (a) projection, and (b) union. What's more, if keys are defined for base relvars, then projection is a problem only when it fails to preserve the attributes of at least one key. And if the operands of

[8] Though we learned about this technique from the PRTV implementers, a reviewer of the present paper has pointed out that Codd himself explicitly referred to pipelining in his paper on ALPHA ([3]).

union can be shown to be disjoint, then again the duplicate elimination process is not needed. In BS12 we used the term "rogue" for a node at which duplicates might arise.

Point 2: Many of the nodes representing relational operators come to no harm if duplicates pass through them. In fact the only nodes that can give incorrect results if presented with the same tuple more than once are (a) those involving aggregation (i.e., the summarization operator) and (b) the root node, whose result must be a true relation. In BS12 we used the term "fussy node" for one that cannot tolerate the appearance of duplicates.

Point 3: In the light of Points 1 and 2, duplicate elimination does not need to be built into the implementations of projection and union. Instead, the responsibility can be borne by the system's optimizer. For example, if a projection node appears somewhere in the line below a summarization node, with several non-rogue nodes in between, a special node for duplicate elimination can be inserted at any point between the rogue and the fussy node and the optimizer can decide which point is likely to give the best overall performance.

That concludes my account of our education by the people we met at IBM's Scientific Centre in Peterlee. It had enormous influence (see Appendix B) on the commercial DBMS we subsequently developed over the following four years and again, two decades after that 1978 meeting, on the language **Tutorial D** [10] devised by Chris Date and myself for purposes of illustration and education and implemented by Dave Voorhis as *Rel* [18] for those same purposes. Back in 1978 I fully expected it to have a similar influence on the future of relational DBMSs in general. After all, when we looked at SQL, still only a research project at the time, we saw that it had many defects and completely failed to properly address our problem number 3, **Attribute names**. It also had no solution, at the time,[9] to the problem of obtaining details and several levels of totals in the same query. In short, we thought, it would "never catch on", and those were the words I used when quizzed by the IBM executive who would authorize the funding for BS12 as to why we had rejected the language they were "all talking about" on the other side of the Atlantic.

But ISBL clearly didn't have that influence, and SQL clearly did "catch on". So, what went wrong? My own attempt to answer that question now follows, but for further information see references [14] and [9], critiques of Codd's 1969 and 1970 papers by David McGoveran and Chris Date.

Why Are There No Relational DBMSs?

That's the provocative question posed in this paper's title. Of course I really mean what some have called *truly* relational DBMSs. A truly relational DBMS, of course, is one that implements a true relational language. But we don't really want a plethora of different languages, in the main commercial field at least, so perhaps a more appropriate question to ask is: Why didn't the ISO subcommittee for Database Languages produce a standard for a true relational database language? According to what I was told shortly after I joined the ISO working group developing SQL standards, that idea had indeed been discussed in the ANSI committee, X3H2, that developed and

[9] It needed the ability to specify a SELECT expression in the FROM clause, which was added to the international standard for SQL in 1992.

published the first SQL standard. This is corroborated by X3H2 member Michael Gorman in reference [11]:

> X3H2 was initially chartered to develop a relational database standard by [parent committee] X3 in 1982. X3H2 determined that it is more expedient to develop a standard based on an existing product, rather than create a DBMS standard from scratch. The X3H2 member from IBM, Phil Shaw, created an SQL technical specification document at the request of the committee.

and that American national standard was soon taken up by ISO and published as an international standard in 1986. Around the end of the decade Michael Stonebraker was to observe, famously and perhaps a little wryly,[10] that SQL had become "intergalactic dataspeak".

That situation[11] therefore prompts us to ask, why isn't SQL truly relational? And, given that it isn't, why did the industry at large so readily embrace it? Well, we can easily forgive the industry at large, I suppose, because in spite of all its defects, SQL was a hugely significant advance on what we had before—its implementations did offer a fairly intuitive declarative query language without the need for loops or pointer chasing, and they did offer a modicum of physical data independence. But why didn't its inventors come up with a truly relational language? Why did they neglect the obvious need for each of a table's columns to have a name? Why did they allow a table to have two or more columns with the same name? Why did they decide to allow more than one copy of the same row to appear in a table? Why did they invent NULL and introduce that third truth value, UNKNOWN?

Names for Columns

I was astounded when I first saw, in 1978, that SQL allowed a nameless column to be specified in its SELECT clause. I had come away from Chris Date's course in 1972 with a vision of our existing report generator operating on a single query result—in other words a single relation—in place of the single file, uniform in record format, that it was required to operate on in the environment of our existing DBMS at that time. It would be simplified by delegating much of its existing functionality—record selection, calculations and totals—to the query language, its task being reduced to just layout specification, including when and where to print the totals. The existing one needed a unique name for each field in the input file and for each calculated field (such as price times quantity). I had also come away from Chris's course with the firm idea that every relation's attributes had unique names, and I didn't realize until many years after BS12 had been developed that Codd's original papers had been a little confusing on that particular requirement. Here is what Codd had to say on the matter in [2], having previously defined a relation's "domains" (really, attributes) to be ordered from left to right:

> In many commercial, governmental, and scientific data banks, however, some of the relations are of quite high degree (a degree of 30 is not at all uncommon). Users should not normally be burdened with remembering the domain ordering of any relation (for example, the ordering supplier, then part, then project,

---

[10] In the early 1980s another language, QUEL, implemented initially in Stonebraker's own product, Ingres, and "used for a short time in most products based on the freely available Ingres source code" (Wikipedia). According to David McGoveran (commenting on an earlier draft of this paper) Ingres's QUEL "did more to swell the early relational market than any other product and language". That QUEL is a more faithful implementation of the relational model and an altogether better designed language than SQL is a generally agreed fact, though it does suffer from some of SQL's defects. Originally it had no concept like NULL but this was added later when QUEL became required to coexist with SQL in Ingres.

[11] Chis Date offers an explanation for how this state of affairs came about in reference [15].

> then quantity in the relation supply). Accordingly, we propose that users deal, not with relations which are domain-ordered, but with relationships which are their domain-unordered counterparts. To accomplish this, domains must be uniquely identifiable at least within any given relation, without using position. Thus, where there are two or more identical domains, we require in each case that the domain name be qualified by a distinctive role name, which serves to identify the role played by that domain in the given relation. For example, in the relation component of Figure 2 [a relation with two "domains", both named "part"], the first domain part might be qualified by the role name sub, and the second by super, so that users could deal with the relationship component and its domains-sub.part super.part, quantity-without regard to any ordering between these domains.
>
> To sum up, it is proposed that most users should interact with a relational model of the data consisting of a collection of time-varying relationships (rather than relations). Each user need not know more about any relationship than its name together with the names of its domains (role qualified whenever necessary).

Unfortunately Codd himself never amplified on that "proposal". Perhaps we can charitably assume that the range variables used in the notation, ALPHA, that he invented himself ([3]) were intended as introduced role names, but in that case the qualified names would have to become real column names in SQL.

After I had discovered ISBL's treatment, I realized that unique attribute names were even more important, for original SQL didn't even have a solution to the problem of combining details and totals in a single relation, described in the answer to Question Q8 in the sectioned headed **Problem 3: Attribute Names**. For one thing, the expression to calculate the totals wasn't allowed to appear as an element in the FROM clause and for another, even if it had been allowed, the columns containing the totals couldn't be specified in the SELECT clause because they didn't have names! (Those defects were eventually addressed in the 1992 edition of the standard, though the language still allows a column to have no name, and two or more columns in the same table to have the same name, owing to what I have called The Shackle of Compatibility—mistakes in language design live forever and that's why it's so important to minimize them by adopting Chris Date's Principle of Cautious Design.)

My best guess as to why column names had been so neglected in SQL is that, as Don Chamberlin once told me, the language had been shown, with Codd's own ratification, to be "relationally complete"[12], meaning that every relation expressible in relational algebra is expressible in SQL. But the relational algebra in question—Codd's, of course—didn't include extension or summarization. So these tables with anonymous columns and/or duplicate column names would only ever been seen as the *final* result of a query—expressions denoting them couldn't appear in a FROM clause. Why burden the user with the need to specify a unique name for each column? As I said, that's my best guess, but if it turns out to be correct, then I have to wonder why it was thought reasonable to (a) require general purpose utilities like ISQL to have to invent names to put in column headings in tabular displays, and (b) forget about column names altogether and depend instead on a column ordering in operations such as UNION, INSERT, and assignment of column names in CREATE VIEW.

Duplicate Rows

I know that Raymond Boyce and Don Chamberlin, the designers of the language SEQUEL that was SQL's precursor in 1974, were worried about the performance implications of having to eliminate redundant duplicates from every query result, and rightly so. That's why SEQUEL

---

[12] Codd's definition did not require support for extension and aggregation, of course.

included special syntax whereby the user could explicitly specify that duplicates were to be retained. However questionable we might think that approach, the decision in SQL to make SELECT without DISTINCT the default is surely much more questionable, as it lures users into the traps that can arise when duplicates arise unexpectedly.[13]

In BS12 we worked out and implemented further mitigation by requiring every base relation to have a declared key and using some heuristics based on functional dependency theory to determine keys for derived relations. A projection that retains a key of its input relation cannot be one that could give rise to duplicates.

It seems that the System R team didn't investigate ways of mitigating the performance problem but, rather, decided to pass it on to users. That decision does seem questionable when one considers that the main aim of their project was to disprove the claims of those pundits at the time who were dismissing Codd's proposals as being infeasible for use with the very "large shared data banks" that the title of his 1970 paper declared they were intended for. Although the team was acclaimed for apparently having achieved that aim, had they really?

NULL and 3-Valued Logic

This proposal, to address the perceived problem of how to represent, in a relation, the fact that a certain piece of information is missing came, alas, from Codd himself (though his first extensive description of nulls and 3VL didn't appear until 1979). Although the proposal was controversial from the outset, not many other approaches had been advanced at the time and those that had, such as "default values", were also problematical. Given the respect that Codd was held in, it is perhaps not all that surprising that the System R team decided to adopt his proposal. It is of some interest that Codd himself later realized it was at least inadequate, though the revision he subsequently proposed was even more complex, involving two different kinds of null and a fourth truth value.

As a matter of fact, we had initially been thinking of supporting some kind of null in BS12, without having given much thought to its ramifications. We changed our minds rapidly when the need (?) for comparisons to support a third possible result in addition to the usual TRUE and FALSE was drawn to our attention (by Stephen Todd, as it happens).

Why Did ISBL Get Overlooked?

While I was drafting the present paper I managed to get hold of a copy of *A Relational Algebra for Machine Computation* [12], by Patrick Hall, Peter Hitchcock, and Stephen Todd. This paper, which I refer to henceforth as HHT1975, was presented at an ACM symposium held in 1975 on principles of programming languages. It lays down the theory that formed the basis for ISBL's operators. I describe this paper as the one that "should have been the game changer", by which I mean that it should have been the basis for *every* relational database language. It changed the game by showing, for the first time, how a notation suitable for *machine computation* could honour Codd's dictum to the effect that there is no ordering to the attributes of a relation. In the cold light of day in 2015 I can see why this paper perhaps didn't get the attention it needed if it were to influence the industry in the way it should have:

[13] The later language addition allowing the key word ALL to be used where DISTINCT is not required didn't help much—The Shackle of Compatibility required ALL to be the default.

- It's a hard read, very terse and dense, lacking in illustrative examples. I know I saw this paper in 1978 but I'm equally certain it would have been impenetrable to me at that time, when I was certainly not well versed in matters of logic and set theory. Luckily for me, I didn't need to understand it at that time because, as I have described, I had been shown ISBL, via PRTV, with lots of illustrative examples, and I did understand those very well.
- It has a long digression on an issue that, though an extremely important one, was really a side issue to the main point of the paper. The issue in question was the one mentioned in my section **Problems with Codd's Algebra**, point **1. Calculations**. The text goes to great lengths in considerable theoretical depth to explain something that is intuitively fairly obvious. It observes, correctly, that a mathematical function, such as addition for example, is a mapping that could be represented by a relation under Codd's definition of that term were it not for the fact that in many cases the relation is infinite in cardinality. It refers to such relations as *computed relations*. It then investigates the circumstances in which a finite relation *r* can be joined with a computed relation *cr* without giving rise to an infinite result, namely, when the common attributes for the join provide in each tuple of *r* a value for the domain[14] of the function represented by *cr*. In that case each tuple of *r* matches exactly one (or at most one) in *cr*, so the cardinality of the result cannot exceed that of *r*. At that time, in 1975, the ISBL team hadn't defined the operators based on this observation that we now know as extension, restriction, and summarization. Perhaps this lengthy treatise would have been more suitably dealt with in a separate follow-on paper.
- The paper was produced by a team working in a comparatively small outpost of IBM in the U.K. Scientific Centre, lacking the superior cachet of its U.S. research labs. Indeed, research of this kind was not really part of the official mission of national Scientific Centres, whose main purpose was to support the scientific community in their own country.
- The conference at which the paper was presented was not one devoted to database issues. I am reliably informed that it did not exactly light up the proceedings, and no doubt the audience's interests were spread among many other topics to do with principles of programming languages.

The upshot is that the details of ISBL's semantics didn't come to the attention of those who might have been in a position to influence the future direction. Also, there was a lack of awareness of the problems to do with attribute naming that I have described as arising from Codd's papers—problems that Brian Rastrick and I, by contrast, were very aware of for the reasons I have given.

We know that Codd was aware of ISBL and had looked at it. He also looked at BS12 but that was in 1982, by which time SQL was the focus of relational attention within IBM. It is interesting to note that neither ISBL nor BS12 had any influence at all on his 1990 publication [5], and nor did my articles that had appeared, at his and Chris Date's invitation, in their company's journal two years earlier.

---

[14] Not "domain" in the sense of either of Codd's uses of this term!

Stephen Todd recalls an IBM meeting in the 1970s to discuss the way forward regarding development and delivery of a relational database product. It took place at the IBM research laboratory in San Jose, California, where System R was developed. ISBL people, including Todd himself, were present at that meeting but were heavily outnumbered by members of the System R team and were apparently unable to steer the debate in a right direction. Chris Date, who was a member of the programming languages department at that time, has no recollection of that meeting and is sure he wasn't invited. If the convenors had thought of involving the languages people, then surely Date would have been the obvious choice.

Todd has little recollection now of how the discussion went but he thinks that SQL won the day because its implementation in System R included suitable update operators (INSERT, DELETE, UPDATE) whereas PRTV, implementing ISBL, had only direct assignment (:=). Moreover, System R supported transactions and concurrent users, so its code implementing SQL was perhaps significantly closer to being "ready to go" as a commercial DBMS. Even so, if Todd is right about the features that might have won the day for SQL—features that have nothing to do with the notation for expressing queries—surely some consideration could have been given to the semantics of ISBL's relational operators and the problems solved by giving every attribute of a relation a unique name. Perhaps ISBL's dense mathematical syntactic style, compared with SQL's verbosity and use of key words, was also an issue, considering that Codd's own syntax in [2] had led to many criticisms along the lines of "The relational model is just too mathematical for ordinary mortals".

Why Did BS12 Get Overlooked?

First of all, there was no culture of publication in IBM's DCS Support Centre where BS12 was planned and developed. Rather, details of our implementations and our source code were always classified as "IBM Confidential" and no patent applications ever emerged from our work—not surprising, considering that in those days software patents were virtually unknown. In any case, there was nothing original concerning the semantics of our relational operators—we got them from ISBL. The only aspects of our work that might have been worth publishing as original ideas were (a) our mitigation of the "duplicate removal" performance problem and (b) our database storage structure, involving a certain amount of "vertical" decomposition and hash tables for the primary keys.

Two members of the System R team—Bruce Lindsay and the late Jim Gray—were among our consultants, Lindsay having been appointed by our management to vet our design. And from the same research laboratory we benefitted hugely from a paper on dynamic extendible hashing, from Ron Fagin. So, although BS12 was supposed to be (and was) the beneficiary of useful advice from the other party in these consultations, the opportunity was there for SQL to benefit from our work too, if only they had noticed—and cared, about the duplicate row problem. If only …

If only …

If only quite a few of the circumstances I have described had been different, then I suggest the true relational model might have seen the light of day instead of—or even as well as—what we actually got. Here, then, is a list of "what ifs", for what they may be worth.

1. What if Codd's 1969 and 1970 papers [1,2] had been more clearly written and/or easier to understand? Some of the things SQL got away with could be put down to a simple lack of

understanding of the relational model on the part of all kinds of people, and that lack of understanding might be partly due to the nature of those early papers. As Chris Date points out in his companion paper to this present piece [9], those papers were in fact quite muddled in lots of ways. (Added to that, it didn't help that Codd also had a habit of changing his positions and definitions in subsequent writings, often doing so silently.) The 1969 paper has also been critically reviewed by David McGoveran [14].

I have expressed, here and in the final section of reference [8], what might be taken as strong criticism of some of Codd's work, but I really don't mean to diminish the value of his original contribution, nor even complain about its deficiencies. It was good enough to inspire others to complete the work, and that's how things should be—see Chris Date's epigraph in reference [9]. And that's how things actually were, in view of the ISBL work, except that nobody noticed—not even Codd himself—apart from Brian Rastrick and me. That we did notice was because of the task we were confronted with—we *had* to sort out the details in order that a language could be devised that we knew how to implement.

2. What if it had been realized early on that lack of any ordering to the attributes of a relations was essential to achievement of his aim of data independence? And then, what if Codd had thought through the implications of requiring every attribute of a relation to have a unique name, as he suggested in the text from [2] cited earlier in the section headed **Names for Columns**?
3. What if Codd's proposal to devolve "calculations" to "the host language" had been properly fleshed out so that we all could understand what he meant?
4. What if Codd, post 1970, having been shown ISBL, had seen it as a significant improvement on his own algebra and had given it due recognition in his lectures and subsequent writings?
5. What if Codd had never floated the idea that led to SQL's NULL and its associated three-valued logic?
6. What if the ISBL team had been able to follow up their 1975 paper [12] with one describing ISBL itself, and present it at some conference devoted to databases, where it would be bound to attract significant interest? (Actually, at least one such paper [17] did appear in 1976, but that one didn't mention the ISBL operators for extension, selection, and summarization either.)
7. What if ISBL and SQL had been equally represented at the IBM meeting in San Jose, California (which I mentioned earlier as recalled by Stephen Todd) to decide which notation to go with? And what if IBM's language specialists in various parts of its empire had been consulted, including the department devoted to that subject in the nearby Santa Teresa Laboratory? And what if the IBM Scientific Advisory Board had been consulted (a group of external experts which at one time counted John Backus among its members, for example)?
8. What if Codd hadn't distanced himself from the programming languages people in IBM, following their refusal to add support for relations to PL/I?
9. What if System R had been developed in the U.K. Scientific Centre and ISBL in the San Jose Research Laboratory?

10. What if PRTV had covered database "motherhood": concurrency, transactions, insert/delete/update, etc.? As already noted, System R did include these essential features and thus was surely perceived by decision makers as significantly more "ready to go".

11. What if Larry Ellison hadn't taken the preemptive action of bringing SQL to the market, and aggressively so, in 1979? IBM followed suit with SQL/DS in 1981, effectively cementing SQL's place in the industry even though it took a few more years to achieve dominance.

12. To make the list up to a number often associated with Codd and also, for a different reason, somewhat dear to my own heart (Business System 12), I finish with a wild one that, for personal reasons as you will see, fills me with retrospective terror. In the 1960s IBM Corporation in the U.S. was forced under litigation to hand its wholly-owned subsidiary Service Bureau Corporation (SBC) to Control Data Corporation (CDC). At that time the product that became Terminal Business System, including the report generator I have mentioned, was under development in SBC, based in San Jose, California—possibly in the very building where the System R team was also housed. Under the agreement, IBM retained that project for eventual use in its remaining service bureaux outside the U.S.A., and the development was transferred to a newly formed team in IBM U.K. (which I joined in 1969, shortly after its inception). What if that dispute between IBM and CDC had never happened? Might the SBC team in San Jose have raised the requirements that motivated Brian Rastrick and me, and might they have taken those requirements to the System R team? (And—shudder!—where would my subsequent career have taken me instead?)

Acknowledgments

Stephen Todd has given invaluable help in connection with the IBM U.K. Scientific Centre's work in the 1970s. Both he and Chris Date saw early drafts and suggested several improvements. David McGoveran saw an early, incomplete draft, encouraged me to complete it, and then, when I did so, gave me some very useful comments causing me to make some corrections to my historical account.

References

[1] E.F. Codd: "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks". IBM Research Report RJ599 (August 19th, 1969).

[2] E.F. Codd: "A Relational Model of Data for Large Shared Data Banks". *CACM* 13, No. 6 (June 1970).

A copy of this paper is available in PDF at http://dl.acm.org/citation.cfm?id=362685 in the ACM's Digital Library.

[3] E.F. Codd: "A Data Base Sublanguage Founded on the Relational Calculus", Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control, San Diego, Calif. (November 1971).

[4] E.F. Codd: "Relational Completeness of Data Base Sublanguages". IBM Research Report RJ 987, San Jose, California (1972).

[5] E.F. Codd: *The Relational Model for Database Management, Version 2.* Boston, Mass.: Addison-Wesley (1990).

[6] Hugh Darwen: "ISBL's Relational Operators", available in PDF at www.thethirdmanifesto.com.

This shows the extent to which Business System 12 in 1980 and **Tutorial D** in 1998 were in turn influenced by ISBL.

[7] Hugh Darwen and C.J. Date: *The Third Manifesto*, available in PDF at www.thethirdmanifesto.com.

[8] Hugh Darwen and C.J. Date: "Textbook Treatments of Relational Algebra", available in PDF at www.thethirdmanifesto.com.

This was originally drafted as an appendix to the present paper.

[9] C.J. Date: "Codd's First Relational Papers: A Critical Analysis". To appear.

A review draft of this paper in PDF is available from Hugh Darwen on request by email. It carries the following epigraph:

> *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)

[10] C.J. Date and Hugh Darwen: Chapter 11, "**Tutorial D"**, in *Database Explorations: Essays on The Third Manifesto and Related Topics.* Trafford Publishing (2010).

This book is available in PDF s a free download at www.thethirdmanifesto.com, as is a standalone document giving a slightly revised language definition. Previous versions of this language appear in earlier books by the same authors.

[11] Michael M. Gorman: *Database Management Systems: Understanding and Applying Database Technology.* Wellesley, Mass.: QED Information Sciences Inc. (1991).

[12] Patrick Hall, Peter Hitchcock, and Stephen Todd: "An Algebra of Relations for Machine Computation", Conf. record of the 2nd ACM Symposium on Principles of Programming Languages, Palo Alto, California (January 1975).

A copy of this paper, with annotations by Hugh Darwen in 2014, is available in PDF at www.thethirdmanifesto.com.

[14] David McGoveran: "Observations on the 1969 Relational Operations" http://www.alternativetech.com/ATpubs_dir.html#1969

[15] Richard Morris: "Chris Date and the Relational Model", an interview with Chris Date, August 2014. https://www.simple-talk.com/opinion/opinion-pieces/chris-date-and-the-relational-model.

See in particular Date's answer to Morris's question: "What was key to SQL becoming the standard language for relational databases in the mid- 1980s? Was it all down to good marketing?".

[16] Frank Palermo: "A Data Base Search Problem", in Julius T. Tou (ed.), *Information Systems: COINS IV,* New York, N.Y.: Plenum Press (1974).

[17] Stephen Todd: "The Peterlee Relational Test Vehicle—A System Overview", *IBM Systems Journal* **15**, No. 4 (1976).

[18] Dave Voorhis: *Rel: An Implementation of Date and Darwen's **Tutorial D** Database Language.* http://dbappbuilder.sourceforge.net/Rel.html.