The Importance of Column Names

Hugh Darwen
hd@thethirdmanifesto.com
www.thethirdmanifesto.com

For IUA[UK] - 6 October 2003

Last updated: 01 October, 2003

Synopsis

A light-hearted but **deadly serious** look at how an astonishingly badly designed but widely used computer language came about.

Illustrated by a close look at its third most severe mistake (after **nulls** and **duplicate rows**).

A plea for an implementation of Codd's Relational Model of Data (1969) to become commercially available (at last).

My criticism of SQL should not be taken as a criticism of the System R project that spawned SQL in the 1970s. Nor should it be taken as any kind of personal criticism of the System R engineers. They did a brilliant job and achieved their stated objective. The resulting acclaim was richly deserved but should not, in my opinion, have resulted in acceptance of the ad hoc language (as I would call it) that formed the user interface to System R.

The comparative severities of SQL's various logical errors is a matter of opinion, of course. I rate nulls—or rather, three-valued logic—as the worst error because, for one reason at least, nulls can't be totally avoided by judicious use of the language, whereas duplicate rows can. The dependence on column ordering can't be avoided either, but somehow this mistake seems, though unforgivable, just slightly less unforgivable than the other two, based on the severity of the problems it causes. If anybody wishes to argue that "unforgivable" is an absolute, I would accept that point.

Codd's Relational Model of Data made it quite clear that (a) every tuple in a relation has a value for every attribute (so, no nulls), (b) the body of a relation is a set (not a bag), and (c) no significance attaches to the order of attributes (where relation=table, tuple=row, attribute=column).

Further Reading

Relational Database Writings 1985-1989
by C.J.Date with a special contribution
"Adventures in Relationland"
by H.D. (as Andrew Warden)

Relational Database Writings 1989-1991
by C.J.Date with Hugh Darwen

Relational Database Writings 1991-1994
by C.J.Date

Foundation for Future Database Systems :
The Third Manifesto
by C.J. Date and Hugh Darwen

Introduction to Database Systems
(8th edition) by C.J. Date

Within these writings can be found much elaboration on SQL's various errors and, more importantly, strong indications of what a *Truly* Relational Database Management System (*T*RDBMS) looks like in contrast. *The Third Manifesto* in particular defines the necessary features, lists several notable features that must NOT appear, and makes some additional strong suggestions.

A Brief History of Data

1960: Punched cards and magnetic tapes
1965: Disks and 'direct access'
1969: E.F. Codd's great vision: "A Relational Model of Data for Large Shared Data Banks" (1970)
1970: C.J. Date starts to spread the word
1973: Stonebraker starts work on Ingres (QUEL) at UCB
1975: Relational Prototypes in IBM: PRTV (ISBL), System R
1980: First SQL products: Oracle, SQL/DS
1986: SQL an international standard
1990: OODB (didn't come to much after all)
2000: XML (?—shudder!)

The line for 1973 mentions Ingres, which was based on Codd's ALPHA. Ingres's language, QUEL, is much more faithful to the Relational Model than SQL and "for some time was a serious contender to SQL" (Date: Introduction to Database Systems, 8th edition).

The line for 1975 mentions PRTV, which perhaps few people nowadays have heard of, and System R, the famous research project that spawned SQL.

PRTV deserves wider recognition. Its language, ISBL, brilliantly answered all the questions about relational algebra-based language design that Codd's papers raised and left unanswered. And it had no nulls, no duplicate rows, and no dependence on column ordering. But PRTV wasn't a DBMS, whereas System R was. The aim of the System R project was to refute the claims of those pundits who asserted that Codd's Model was infeasible, for performance reasons.

SQL was designed by people whose primary interests and skills lay in DBMS engineering. ISBL was designed by people whose primary interests and skills lay in computer language design.

A Brief History of HD

1967 : IBM Service Bureau, Birmingham

1969 : "Terminal Business System" – putting users in direct contact with their databases.

1972 : Attended Date's course on database (a personal watershed)

1978 : "Business System 12"
- a relational dbms for the Bureau Service

1985 : Death of Bureau Service (and of BS12)

1987 : Returned to database after brief absence. Attended Codd & Date database conference in London

1988 : "Adventures in Relationland" by Andrew Warden. Joined SQL standardization committee.

Let me explain the 1972 line. In response to TBS customer demand, I had been struggling to put together a scripting language for a report generator that would be able to collate information from multiple files, some of whose records might be connected together by means of pointers. And I needed a Data Definition Language capable of describing such "structures", of course. My struggles got me absolutely *nowhere*. Then Chris Date taught me Codd's Relational Model, and suddenly I had the basis of a complete answer to our customers' problems. Of course, there was a small matter that would need to be attended to—the design of a language based on that Model.

It was not feasible to retrofit a relational language to TBS, but by 1978 a strong business case had arisen for a new, state-of-the-art DBMS for the Bureau Service, enabling us to start with a clean sheet. For Business System 12 we took advice from System R people regarding the DBMS engine, but we had no difficulty in rejecting SQL in favour of a language of our own devising, based on the operators of ISBL (though not using ISBL syntax).

The Importance of Column Names

Ref: “The Naming of Columns” by Andrew Warden in *Relational Database Writings, 1985-1989.*

and now:

A Sweet Disorder, by C.J. Date, available at http://www.dbdebunk.com.
Subtitled “The relational model prohibits left-to-right column ordering for sound practical reasons.”

I rather think right-to-left column ordering is prohibited too. (Joke)

Variations on a Theme: 5 SQL Queries

1. SELECT E#, SALARY + BONUS – total pay of each employee
   FROM EMP
2. SELECT D#, COUNT(*) – number of employees in each department
   FROM EMP
   GROUP BY D#
3. VALUES 0
4. SELECT W1.E#, W2.E#, W1.P# – employees working on same project
   FROM WORKS_ON W1, WORKS_ON W2
   WHERE W1.P# = W2.P#
   AND W1.E# > W2.E#
5. SELECT X, Y FROM T1
   UNION
   SELECT Y, X FROM T2

The results of all these expressions are called tables, but none of them can be the current value of a *table variable* (a.k.a. "base table")!

1. The second column has no name.
2. The second column has no name. (Exercise: what is the rule under which the first column does have a name?)
3. The only column has no name.
4. The first two columns have the same name, E#.
5. Neither column has a name (though they both would if the second operand were SELECT X, Y FROM T2!)

Do you know of any other computer language that supports expressions whose results cannot be stored in variables?

Consider how one creates a view based on such SQL expressions, and how one stores their results in the database.

How does one get SQL to present a result that is sorted on an anonymous column such as SALARY + BONUS or COUNT(*)? Didn't the original SQL language designers simply take the easy way out, with their "ORDER BY 2"? Was this perhaps because System R had, as they say, bigger fish to fry and could not be much bothered by such trifling matters (in the context of the stated aim of their project)?

Examining Example 1

1. SELECT E#, SALARY + BONUS – total pay of each employee
   FROM EMP

Suppose we want only the high earners. In the SQL of 1979:

```
SELECT E#, SALARY + BONUS
FROM EMP
WHERE SALARY + BONUS >= 5000
```

Many years later, “AS” was added, but did it help?

Try:

```
SELECT E#, SALARY + BONUS AS TOTAL_PAY
FROM EMP
WHERE TOTAL_PAY >= 5000
```

Illegal! And rightly so.

The necessity to repeat expressions such as SALARY + BONUS didn’t go away until the mid-1990s and still exists in some implementations. It seems outrageous, considering the importance that has always been attached to avoiding such repetition.

But the emphasis of System R was on performance, not on design of an industrial-strength language. One can guess (not necessarily rightly) that the SQL designers couldn’t easily find a way of avoiding that repetition, given the style and structure they had settled on, but they weren’t expecting the language itself to achieve widespread adoption. One can also surmise that had they tackled the problem head-on, they might have had second thoughts about the chosen style and structure.

In their defence, it should be noted that Codd himself, whom they did consult, paid little attention to “calculated columns”. His Relational Algebra did not include the EXTEND and SUMMARIZE operators that were added later by others.

The "Right" Solution

(Well, one that works.)

```
SELECT DISTINCT E#, TOTAL_PAY
FROM ( SELECT E#, SALARY + BONUS AS TOTAL_PAY
       FROM EMP ) AS TEETH_GNASHER
WHERE TOTAL_PAY >= 5000
```

(Aside: awful language design continues to plague us!)

"Derived tables in the FROM clause" eventually made it to the SQL international standard in 1992, but it still took several years for the major SQL vendors to provide it and some products are still without it.

SQL is **relationally incomplete** without this feature. What does that say for the original language design?

When we look at the "right" solution that did eventually come along, we can begin to see why I wonder if the SQL designers might have had second thoughts about the SELECT-FROM-WHERE structure, had they addressed the repetition problem up front.

A word about relational incompleteness. Completeness is a useful concept only when we have a clear understanding of what it takes to be complete in some sense, and what kinds of problems can be solved given such completeness. Like computational completeness, relational completeness is such a concept. The real problem with not having relational completeness is not so much that there are certain problems that you can't solve but would be able to solve with if you did have completeness. Rather, it is not being able easily to identify the problems that can't be solved. In other words, you can scratch your head for hours in pursuit of a nonexistent solution.

TEETH_GNASHER? Yes, sorry about that. It makes me *really mad* that the SQL syntax *requires* a name to be given here, considering that it is not required always to give a name in places where a name really ought to be required. Grrr!

The Other "Right" Solution

(But significantly less "right" than the first)

```
SELECT DISTINCT E#, TOTAL_PAY
FROM ( SELECT DISTINCT E#, SALARY + BONUS
         FROM EMP ) AS TEETH_GNASHER ( E#, TOTAL_PAY )
WHERE TOTAL_PAY >= 5000
```

In Codd's Relational Model, **no significance at all** is attached to any order in which the attributes of a relation might be defined or perceived to appear.

This was one of Codd's **great insights.** Mathematicians deal with ordered n-tuples, and a Cartesian product operator that is non-commutative. Codd saw an improvement on those concepts for database purposes.

Here's my guess as to how this redundancy—utterly needless, surely?—might have come about.

CREATE VIEW was part of original SQL. It was decided that the things called views should look as much as possible like the things called base tables. Now, SQL does require every column in a base table to have a unique name. Views are defined on query expressions, but some query expressions define tables that do not have that required property. How to assign unique column names to the columns of the query on which the view is defined? The solution they came up with was CREATE VIEW <view name> (<column-name-commalist)—i.e., an optional list of column names in parens after the view name, the correspondence with the columns of the query being determined by ordinal position.

No doubt, the existence of that column-order-dependent device in CREATE VIEW inspired the addition of a similar device for use in nested FROM clauses. And no doubt the ability to assign column names in the SELECT clause came later, when people complained about the inconvenience of column ordering!??

The Tutorial D Solution

```
( ( EXTEND EMP ADD SALARY+BONUS AS TOTAL_PAY )
  WHERE TOTAL_PAY >= 5000 ) { E#, TOTAL_PAY }
```

Or, if you prefer the steps to be more clearly visible:

```
WITH ( EXTEND EMP ADD SALARY+BONUS AS TOTAL_PAY ) AS T1,
        ( T1 WHERE TOTAL_PAY > 5000 ) AS T2 :
        T2 { E#, TOTAL_PAY }
```

- No need for two projections.
- No need to think about whether DISTINCT is needed.
- And look! – the operations are performed in the order in which they are specified, in the conventional manner.
- "AS" used for two purposes, but in a consistent way. The expression is always on the left, the name on the right (cf. SQL).

The notation used in Tutorial D was designed with pedagogic considerations in mind. We do not particularly advocate it for industrial strength DBMSs.

In this example, we first operate on the relation variable ("base table") EMP, extending it to produce a relation (table) with an additional, derived attribute (column) called TOTAL_PAY. Call that result T1. Then we apply a restriction (WHERE) to T1, discarding the tuples (rows) we are not interested in. Call that result T2. Finally, we project T2 over the required attributes (columns).

Examining Example 2

2. SELECT D#, COUNT(*) – number of employees in each department
   FROM EMP
   GROUP BY D#

Suppose we want only the big departments. In the SQL of 1979:

```
SELECT E#, COUNT(*)
FROM EMP
GROUP BY D#
WHERE COUNT(*) >= 50 – wrong! (why?)
HAVING COUNT(*) >= 50 – "right"!
```

Semantically, HAVING = WHERE (= ON in JOIN).
Why have more than one operator with same meaning, with only one of them the "right" one to use in any given context?

They couldn't use WHERE for this purpose, because (a) WHERE can appear in the same query, *before* the GROUP BY clause, to denote a restriction on the result of the FROM clause, and (b) the GROUP BY clause can be omitted, in which case the existence of an aggregate operation in the SELECT clause implies GROUP BY ( ). (GROUP BY ( ) wasn't allowed to be written explicitly until the late 1990s.) In certain circumstances, then, it would be difficult to know whether a WHERE restriction was to be applied before aggregation or after it.

When you get into a fix like this in the design of a computer language, it might be a good idea to reconsider the assumptions that might have got you into it in the first place. Unless for some reason it's too late to do that, of course.

SQL:1992 Made HAVING Redundant

```
SELECT D#, COUNT(*)
FROM EMP
GROUP BY D#
HAVING COUNT(*) >= 50
```

In SQL:1992 this is equivalent to

```
SELECT *
FROM ( SELECT D#, COUNT(*) AS NUMBER_OF_EMPS
         FROM EMP
         GROUP BY D# ) AS TEETH_GNASHER
WHERE NUMBER_OF_EMPS >= 5000
```

Why wasn't the problem recognised in 1979?

If it had been, would they ever have bothered with HAVING at all?

And might they then have had second thoughts about the enforced SELECT-FROM-WHERE structure?

The Tutorial D Solution

```
( SUMMARIZE EMP
    BY { D# }
    ADD COUNT ( ) AS NUMBER_OF_EMPS )
  WHERE NUMBER_OF_EMPS >= 50
```

By the way, Tutorial D also supports “densifying”. E.g., for number of employees in each department, including the empty departments:

```
SUMMARIZE EMP
    PER DEPT { D# }
    ADD COUNT ( ) AS NUMBER_OF_EMPS
```

So BY { D# } is shorthand for PER EMP { D# }
This is a *planned* shorthand. Helps user *and* system.
So much of SQL’s redundancy is *unplanned*, caused by bad mistakes in original design.

“Densification” has been a known requirement on SQL, from the Data Warehouse community especially, for some time. An extension to support it is currently being considered by the committee that develops the SQL international standard.

Examining Example 3

3. VALUES (0)

How to name the columns of a VALUES expression?

```
SELECT DISTINCT *
FROM ( VALUES 0 ) AS TEETH_GNASHER ( ZERO )
```

Note the necessity to add SELECT and FROM!

In Tutorial D:

```
RELATION { TUPLE { ZERO 0 } }
```

Sorry about that DISTINCT. Utterly unnecessary of course, but sometimes I find the urge to write it just too irresistible.

Examining Example 4

```
4.  SELECT W1.E#, W2.E#, W1.P# – employees working on same project
    FROM WORKS_ON W1, WORKS_ON W2
    WHERE W1.P# = W2.P#
        AND W1.E# > W2.E#
```

Nowadays, the “right” solution is

```
SELECT *
FROM ( SELECT E# AS E1, P#
         FROM WORKS_ON ) AS TEETH_GNASHER
         NATURAL JOIN – if you can get it
       ( SELECT E# AS E2, P#
         FROM WORKS_ON ) AS TEETH_GNASHER_AGAIN
WHERE E1 > E2
```

In Tutorial D:

```
 ( ( WORKS_ON RENAME E# AS E1 ) JOIN ( WORKS_ON RENAME E#
AS E2 ) ) WHERE E1 > E2
```

I have a further observation in connection with duplicate column names. The following expression is legal in SQL:

SELECT C1 AS X, C2 AS X FROM T

Aren’t those duplicate names likely to be a mistake, at least in a more realistic example? Isn’t it usual in computer languages for the compiler to outlaw such clashes? SQL outlaws it only if you subsequently attempt to reference column X. But why? Answer: because if the duplicate names specified via AS had been outlawed, then duplicate column names in general would have had to have been outlawed. And that would have led to an incompatibility.

It is sometimes impossible to recover from mistakes in language design, because of what I call “the shackle of compatibility”.

Note that in Tutorial D, the only “join” operator is called JOIN, and it means “natural join”. Consider the sentences “*e1* works on project *p*” and “*e2* works on project *p*”. When we connect those two sentences together by a conjunction, we obtain a single sentence with two references to project *p.* For example, “*e1* works on project *p* and *e2* works on project *p*”. In such a sentence each appearance of the symbol *p* stands for the same thing! That is the simple principle on which natural join is based. There should be no other kind of join.

Examining Example 5

5. SELECT X, Y FROM T1
   UNION
   SELECT Y, X FROM T2

Nowadays, the “right” solution is

```
SELECT X, Y FROM T1
UNION CORRESPONDING
SELECT X AS Y, Y AS X FROM T2
```

<*screaming aside*> **Why is there no INSERT CORRESPONDING?**</*screaming aside*>

If UNION CORRESPONDING had been accepted as correct in 1979, wouldn’t it have been spelled just UNION?

In Tutorial D:

```
( T1 { X, Y } ) UNION ( T2 RENAME ( Y AS X, X AS Y ) )
```

Actually, not many SQL implementations seem to support CORRESPONDING, but it’s been in the international standard since 1992.

Few people have had the experience of using a proper relational language. Of those who have, I strongly suspect that none of them ever complained about some perceived inconvenience in pairing columns according to their names.

The Growth of Redundancy

The following SQL features are among those that became redundant as former mistakes were recognised and “corrected”:

- subqueries in the WHERE clause
- range variables (a.k.a. “correlation names”)
- doing joins in longhand
- the HAVING clause
- the GROUP BY clause
- UNION/INTERSECT/EXCEPT based on column order.

Which would you retain if we could start again with a clean sheet?

But in any case, wouldn’t you really like to be able to try out a *T*RDBMS? (*T* = *Truly*)

Of course that bullet list is very incomplete.

The Relationlander's Promise

I solemnly promise …

… *never* to use the word "relational" when I mean SQL, …

… cross my heart and hope to die.

I made this promise, privately, in 1980. I have kept it ever since. Fortunately for me, the SQL international standard deliberate avoids the word "relation" and its derivatives.

The Dream Database Language

D: “Date and Darwen’s Database Dream”

A language whose name includes “D” conforms to *The Third Manifesto* and faithfully implements The Relational Model of Data, with

- *NO EXTENSION*
- *NO PERVERSIONS*
- *NO SUBSUMPTION under something else*

A Guiding Light

"All logical differences are big differences"
(Wittgenstein)

All logical mistakes are big mistakes
(Darwen's corollary)

All non-logical (psychological) differences are small differences
(Darwen's conjecture)

Conceptual Integrity

"Conceptual integrity is *the* most important property of a software product"

(Fred Brooks, 1975)

Of course, you must *have* concepts before you can be true to any. These had better be:

a. few
b. agreeable to those invited to share them

The citation is from "The Mythical Man-Month".

Reims Cathedral

Fred Brooks uses this cathedral to illustrate what he means by conceptual integrity.

Conceptual Integrity

Principle #7 (bis)

"This above all: to thine own self be true,
And it must follow, as the night the day,
Thou canst not then be false to any user."

(from Polonius's advice to D, by WS with HD)

From Hamlet: Polonius's advice to his son, Laertes, on the latter's departure from Denmark to France. Except that the last word was "man" in WS's original. I imagine Polonius wagging his finger at a would-be D.