Back to **normality**

The mighty meter problem is finished off once and for all as Mark Whitehorn covers normalisation of the original table and the fastest SQL solution.

Over the last two issues, we have looked at SQL (having started with the fundamental operators which underly the language). To my delight, the Editor has decided that SQL warrants wider coverage. So, next month, we will begin a separate three-part series on the subject, as a feature elsewhere in *PCW*.

This gives us more room in the column to look at other issues, which is handy because I want to finish off the continuing "meter problem" once and for all, (see the letter from Phil Bowles, below).

Meter for measure

I will briefly recap for the benefit of non-regular readers. In the March issue I posed a problem which my colleague, Stephen, had encountered in real life. It involved a table of readings from electricity meters; see the screenshot, *Table 1*. The primary key, in this case is made from [Meter No] and [Date].

The problem was that the people who produced the data also wanted to see it as shown in the screenshot, *Table 2*. This table shows data from two records in the same row and the apparently simple question was, how can you produce a table like that shown in the screenshot *Fig 1* from *Table 1* without offending the relational model?

Stephen and I found a workable way of deriving the second table from the first, but as our solution offended the relational model, I asked in this column if anyone knew of a solution which didn't cause such offense. I was grateful to be inundated by replies, several of which were published in the May issue.

However, two further points arose. One was normalisation. One respondent (as discussed in the June issue), suggested that the table I had originally used was not normalised and that this was part of the problem. He suggested the screenshot, *Table 2*, as an alternative. The primary key in this table is Reading No.

So, in the June issue I asked readers: which one do you think is flawed in terms of the relation model, and why?; to what update and delete anomalies does the flawed one lead?; which one will be faster when queried?; and what are the implications of using each table in a real database? The other issue was one of speed. Several people wanted to know; of the SQL solutions presented, which was the fastest?

In this issue we'll look at both of these areas, since both are relevant to more than merely the original question.

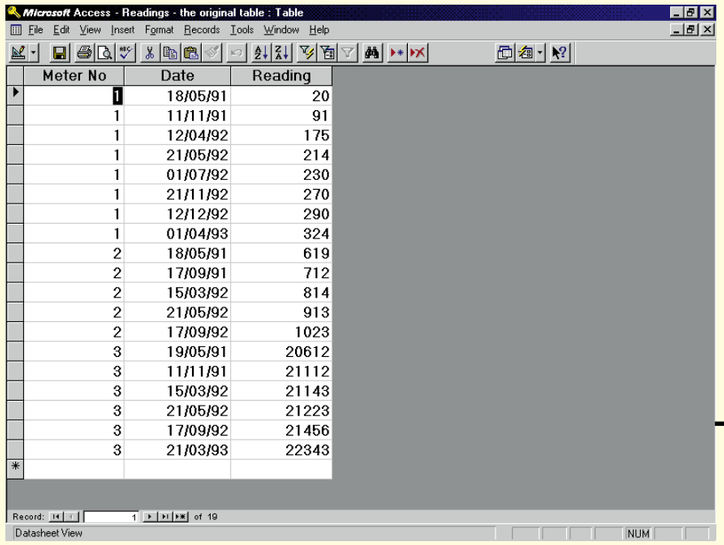


Table 1 *The primary key is compounded from Meter No. and Date*

Normalisation and data redundancy

By far the majority of readers who replied felt that Table 1 was properly normalised, as did Chris Date… yes, *the same*; the other half of "Codd and Date".

Regular readers will remember, that in the June issue a certain respondent suggested that I should "…go away and re-read Codd". Happily, I found I could do better than that because it was very

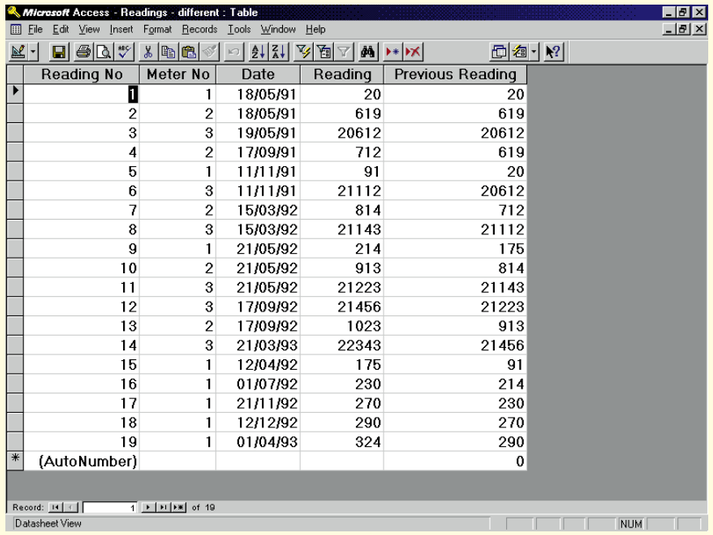


Table 2
The primary key is the "Reading No."

shortly afterwards that I met up with Chris Date. I couldn't resist showing him the tables and asking his opinion. In fact, we discussed three tables, the third being one where each record contains a pointer to the "previous" reading; as in the screenshot, *Table 3*.

Chris gave the following opinion: "We can ask ourselves 'what is the effect of normalisation?' Well, basically, it's to reduce redundancy but in order to consider that question carefully we have to have a careful definition of what redundancy is, and without getting into such a refined definition (because I don't think I could give you one), I will simply point out that normalisation *per se* does not, in general, eliminate all redundancy.

"Here's a classical example" (he indicated Table 2). "This is in third normal form and yet there is clear redundancy. What normalisation does (normalisation to the ultimate normal form) is to get you to a position that guarantees that you will not have any update anomalies that can be removed by taking projections. It doesn't say it'll get rid of all anomalies, it just gets rid of those anomalies which can be removed by taking projections.

So yes, you can have redundancy, and normalisation doesn't help. In fact, normalisation is the one tiny piece of science we have but it is not enough — there are all kinds of other questions — is this" (Table 2) "a good design or a bad design? — I don't know because it is subjective, there is no science there. The only sort of working definition of redundancy you can have is if, somehow, you can make something smaller: then you have redundancy. My gut feeling is that it's a bad design, but I can't quantify or qualify that really."

I think the information Chris gives here is well worth stressing, if only because several books that I have on database design get this wrong. Normalisation doesn't guarantee to remove all redundancy,it only removes that which can be removed by projection. Therefore, you can normalise a set of tables and still have redundancy and, hence, update anomalies lurking in the tables.

So, to answer my own questions: which is flawed in terms of the relation model? All three are in third normal form, but Table 2 contains redundant data, and both Tables 2 & 3 can suffer from update anomalies (see below). To what update and delete anomalies does the flawed one lead? Tables 2 & 3 have potential update and delete anomalies.

For example, consider Table 3. Suppose that we discover that meter no.1 was also read on 01/02/93 and yielded a reading of 300. We can add a record like this:

Reading no.*	Meter no.	Date	Reading	Prevs readg no.
15	1	12/04/92	175	5
16	1	01/07/92	230	9
17	1	21/11/92	270	16
18	1	12/12/92	290	17
19	1	01/04/93	324	18
20	1	01/02/93	300	18

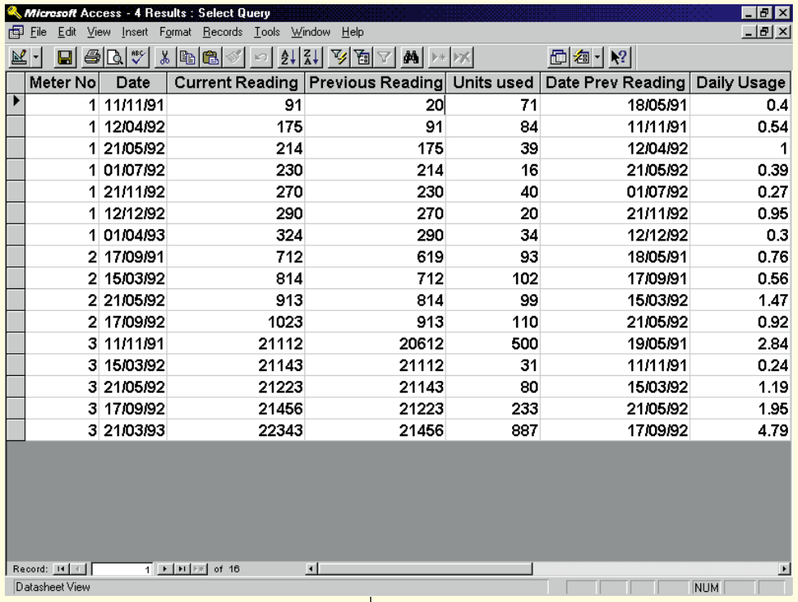


Fig 1 *The way in which the users want to see the data*

The fact that the row is "out of sequence" (at least, in terms of dates) is of no consequence. However, the addition of record 20 has rendered the pointer in record 19 incorrect (it now points to the wrong record). So unless we locate the errant record and correct it, the table now has an internal inconsistency.

In a nutshell this is what is wrong with this type of table design (in my view). Simple changes — updates, and by the same token, deletions — to, or of, one record can cause anomalies in other records. To ensure internal data integrity, some or all of the table has to be checked for integrity after every update. This is clearly not impossible to do but does makes extra work for the developer and may well slow down the database, particularly in a multi-user environment.

In addition, even if the developer's work is perfect, later maintenance work on the database may unknowingly circumvent the checks and lead to a loss of integrity.

Incidentally, the same generic problem exists with Table 2:

Reading No.*	Meter No.	Date	Reading	Prevs. Readg.
15	1	12/04/92	175	91
16	1	01/07/92	230	214
17	1	21/11/92	270	230
18	1	12/12/92	290	270
19	1	01/04/93	324	290
20	1	01/02/93	300	290

Which one will be faster when queried? It is clear that Table 2 will be the fastest for queries like the one desired by the people who wanted the original table. However, if the necessary checking is performed after updates, it will be slower to update.

What are the implications of using each table in a real database? To summarise, Table 1 makes the maintenance of data integrity much easier: queries run against Tables 2 and 3 should run queries more rapidly.

My feeling is that I would rarely implement a base table like 2 or 3, and most readers agreed, although many, like correspondent Phil Bowles, gave an impressively balanced view:

"It may well be that with all those things considered, his choice of solution is a shooting offence. Who can tell? Personally, I try to adhere to a clean design at the outset as experience shows me that it prevents major problems in the future and I'm clever enough to cope with complicated SQL — so I'd go with you. But then again, if I had a team of trainee programmers who were SQL-illiterate, I might consciously make the compromise, adulterate the design and simplify the SQL to reduce development problems.

Its all so complicated, isn't it? These are the reasons that I am no longer an IT professional — at the end of the day, who cares? This type of 'holy war' is one of the reasons why I left IT to become a police officer. If there's going to be a row, let it be over something that matters."

Be warned, he closed his email with *"So my final answer is: I don't care who is right or wrong but if you don't stop arguing right now, someone's going to get nicked."*

…OK, guv', fair enough.

The truth is, of course, that both of these table designs have advantages. The good news is that with a bit of extra work we should be able to have our speed *and* our data integrity.

Suppose we store the data in Table 1, and use that table for all data entry, updates and deletes. Suppose also that, every night, we run a background process that generates Table 2 from Table 1 and writes it to disk. We can then run the user queries against Table 2, and they will run like greased lightning.

Certainly there are disadvantages. The queries that we run under this regime have the potential to yield answers which are a maximum of one day out of date. So we might run the update of Table 2 twice a day, or three times a day — the important point is to discuss it with the users and discover what are acceptable limits. We can even offer them two alternatives: fast queries (run against Table 2) where the data might be slightly out of date; or slower ones (which run against Table 1) which are guaranteed to be up to date. In essence, what I am suggesting is a very simple form of data warehousing. It combines the best of both worlds, which is why it has become so popular recently. Trendy isn't necessarily bad.

Speed

It is worth stressing that in the original question all those months ago, I didn't ask for a *rapid* solution. I never mentioned speed at all, I asked for elegance, academic purity, relational correctness, whatever you want to call it, but not speed. So, while some of the original replies turn out to be slow, this in no way reflects badly upon either the "worth" of the original reply or the worth of the people who supplied them.

However, you asked for the relative speeds so here they are. I looked at three solutions. The first (labelled 4-Stage SQL) is the original one that Stephen and I concocted. It's crude, it's messy, it offends the relational model, but it works. The second is a two stage solution which used two SQL statements and the third is a single SQL solution. (Both of these solutions were featured in the May issue).

Times to complete the queries are given in seconds.

Number of Records	Four-stage SQL	Two-stage SQL	Single SQL
100	2	3	18
200	3	6	79
400	4	18	395
1000	5	117	*
2000	8	553	
20000	58		

*(After 2,700 seconds, the completion bar for the query was showing one percent. I know those bars are notoriously inaccurate but even so I thought the point had been made that it was very slow. I stopped the test because I needed the PC for something else).

Now see *Fig 2*; the obvious implication from these results is that the original, rather offensive, solution, happens to be very fast. The single stage SQL, while interesting, is very slow, even compared to the two-stage one. However there is a more fundamental difference. It is clear (if we invert the figures) that the efficiency of the "pure" solutions, in terms of the number of records processed per second, drops as the size of the table increases.

Number of records processed per second

Number of Records	Four-stage SQL	Two-stage SQL	Single SQL
100	50	33	6
200	67	33	3
400	100	22	1
1000	200	8	*
2000	250	4	
20000	345		

By contrast, the efficiency of the original method measured in these terms, actually increases. I must admit, that this answer surprised me initially, because I expected the set operations to be inherently faster. However, a little thought suggested an answer.

The original method still uses set operations — they are only "impure" in terms of the relational model. The great advantage for this method is that it simply has to manipulate tables of the same size as the test data. The problem for both of the "pure" solutions is that they involve self-joins. These in turn are generating huge intermediate tables which are presumably increasing in size by something like the square of the number of records. I suspect it is this that is gluing up the processing.

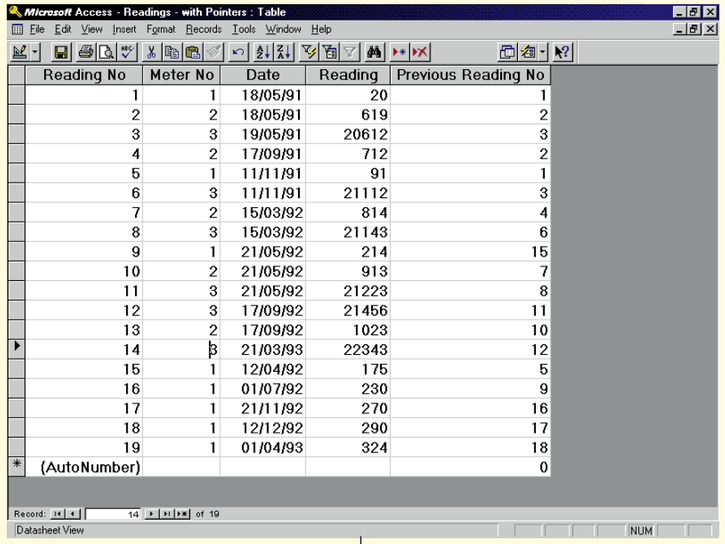
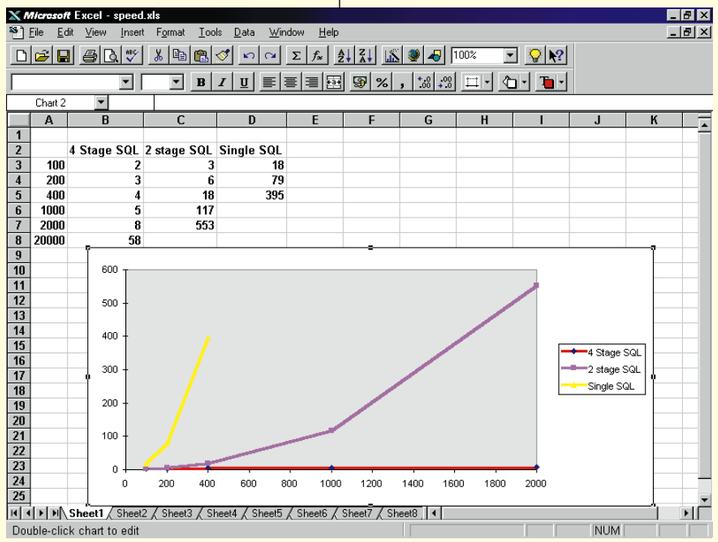


Table 3 *An alternative to Table 2, which essentially uses pointers rather than the data from the previous record*

Fig 2 *The results of the speed tests*



For those who still maintain an interest, and wish to speed test their own solutions, I have included an MDB file, on our cover-mounted CD-ROM, which is the testing database that I used. It is crude and essentially undocumented, because I developed it for my own use. Nevertheless, I suspect that readers who are competent enough to try their own SQL solutions will be able to drive it.

There is a form, with associated code, which will generate the test data for you. If you can find a really fast, yet still pure, solution let me know — but please, only if it is significantly faster!

● *For more about SQL, see the new feature series starting in next month's issue.*

PCW Contacts

Mark Whitehorn welcomes readers' correspondence and ideas for the Databases column. He's on **m.whitehorn@dundee.ac.uk**