PASCAL

Pointing Backwards

In an article titled *Using Oracle Nested Tables*, Donald Burleson writes: "Using the Oracle nested table structure, subordinate data items can be directly linked to the base table by using Oracle's newest construct, the object ID (called an 'OID'). One of the remarkable extensions of the Oracle database is the ability to reference Oracle objects directly by using pointers as opposed to relational table joins. Proponents of the object-oriented database approach often criticize standard relational databases because of the requirement to reassemble an object every time it is referenced."

There are two claims here: that (a) object IDs (essentially, pointers) and (b) nested tables, are "remarkable" Oracle extensions; there is also that annoying nonsense about "reassembling objects". We have dispelled this stuff many, many times, see for example, Don't Mix Pointers with Relations in C.J. Date's RELATIONAL DATABASE WRITINGS 1994-1997, and PRACTICAL DATABASE FOUNDATIONS papers #1, *What First Normal Form Really Means*, and #2, *What First Normal Form Means Not*. But those who know and care only about products, are ignorant of fundamentals, yet feel competent to write or teach, keep repeating them.
So let us restate our arguments in brief. In this article we discuss the drawbacks of pointers (unless otherwise stated, in what follows quotes are from the Date article, to which the reader is referred for more in-depth treatment). We will tackle nested tables in a future column.
Here's Date: "The idea that databases should be allowed to include pointers to data as well as data per se has been around for a long time. Certainly it was a sine qua non of the old pre-relational (IMS and CODASYL) world, and – in the shape of 'object IDs' – it permeates the object world as well. And despite the fact that *Codd very deliberately excluded pointers from the relational model* when he first defined it, the same idea rears its head from time to time in the [SQL] world ... [e.g.] current 'SQL3' attempts to extend the SQL standard to include support for objects ... Note: Given that there are no pointers in the relational model, it follows immediately that *'relational'* databases that include pointers are – by definition – not relational!"
Aside from the ignorance of fundamentals and the history of the database field, he assigns the origins of the renewed interest in pointers to one of the confusions in the object world: "So what exactly is an object? Is it a value or a variable? Or both? Or something else entirely? Note: In fact, it's precisely this object world confusion over values vs. variables that seems to be one of the principal sources of the 'mixing pointers and relations' idea..."

He then proceeds to explain why mixing pointers with relations is a very bad idea.

Complexity and error-proneness
Pointers lead to pointer chasing, and pointer chasing is notoriously error-prone.
Pointers require support of address types, values, and literals, and referencing and dereferencing operators for them, which are confusing in terminology and use.
(Note: In the case of SQL, pointer support exacerbates another problem, language redundancy: many different ways of doing the same thing, which makes optimization difficult – see SQL Redundancy and DBMS Performance).

Pointers require support of row variables, another violation of the relational model: Because values don't have addresses, only variables do, values in table columns of address type must be addresses of row variables, not of row values. But: "The relational model deals with relational values, which are (loosely speaking) sets of row values, which are in turn (again, loosely speaking) sets of scalar values. It also deals with relation variables, which are variables whose values are relations ... However, it does not deal with row variables (which are variables whose values are rows) ... the introduction of row variables dictate that we'd have to define a whole new query language for rows ... (a 'row algebra'?), analogous to the ... [relational algebra]. We'd also have to define row-level update operators, analogous to the existing relational ones. We'd have to be able to define row-level integrity and security constraints, and row-level views. The catalog would have to describe row-variables as well as relation variables ... We'd need a row-level design theory ... We'd also need guidelines as to when to use row variables and when relation variables."

Consequently:
Adding a new kind of variable adds complexity, but not power;
User interfaces will be more complex;
Applications will be more difficult to implement and maintain, and more vulnerable to changes in database structure.

Pointers pertain to base, but not derived relations:
When people talk of "pointers to rows in relations," it is quite clear that what they mean is pointers in *base* relations specifically ... In other words, they forget about *derived* relations! ... this is a mistake of the highest order, because the question as to which relations are base, and which derived is, in a very important sense, arbitrary.

Violation of Codd's core relational requirement, the Information Principle:
... [a] result of [an] address invocation ... [is a scalar value] that cannot be derived from the scalar values in the database. And so there's apparently some "information in the database" that's *not* "cast explicitly in terms of values in relations," and the principle is thereby violated.

Redundancy:
Adding pointers to the relational model is ... unnecessary! The relational model has managed perfectly well without them for over a quarter of a century; thus, any "relational" pointer mechanism – *if* it could be made to work in such a way as to overcome all of the objections already articulated ... – would still be 100 percent redundant.

The "reassembling objects" argument is flawed for at least two reasons. First, object orientation, which is fuzzy and does not have a scientific foundation, lacks well-defined, precise design guidelines; what is an object, and which objects to define are in the eye of the beholder. Second, because the approach originates in programming, its proponents tend to have a biased perspective, usually toward a specific application, ignoring the needs of other applications, including future ones. The normalization principles in the relational approach avoid both these flaws.

Burleson is oblivious to all this. Like so many cookbook practitioners, he offers a simplistic example:
"... a nested table is used to represent a repeating group for previous addresses. Whereas a person is likely to have a small number of previous employers, most people have a larger number of previous addresses, and the nested tables allows repeating groups to be linked to the employee with pointers."
And a snippet of data definition code and, voilá:
"That's all there is to it ... The locator enables Oracle to use the pointer structures to dereference pointers to the location of the nested rows. A pointer dereference happens when you take a pointer to an object and ask the program to display the data the pointer is pointing to."

Remarkable, isn't it? Isn't ignorance bliss? For how 'remarkable' Oracle's nested tables are, stay tuned.

Fabian Pascal is onafhankelijk IT-analist, consultant en auteur gespecialiseerd in data management.
Zie ook zijn website www.dbdebunk.com