

Relational Theory and Data Independence: Unfinished Business.

Dr. Tom Johnston

Much has been made of the data independence that relational technology is said to provide. And indeed, much has been accomplished by this theory and these products. In particular, relational products implement two levels of data independence: physical and logical. These concepts can best be understood in terms of the ANSI/SPARC three-layered data architecture.

[insert illustration.]

Logical Data Independence and the CREATE VIEW Statement.

SQL Data Definition Language (DDL) provides a CREATE VIEW statement which does two things. First, it creates the objects in the External Layer. These objects are view schemas. Second, it provides a mapping between those objects and the objects in the “next layer down”. Those next-layer-down objects are base table schemas, and so CREATE VIEW statements define schemas for views based on schemas for tables.

This mapping is what provides *logical data independence*, which is independence of objects in the External Layer from objects in the Conceptual Layer. To a considerable extent, schemas in either layer can be changed without affecting schemas in the other layer. Also, because this mapping is done declaratively (with the CREATE VIEW statement, not with procedural code), it is easily changed. This ease of change – this mapping which is specified declaratively – makes change less expensive than if the mapping had been done with procedural code.

Physical Data Independence and the CREATE TABLE Statement.

At the next layer down, SQL DDL provides a CREATE TABLE statement which also does two things. First, it creates the objects in the Conceptual Layer – schemas for base tables. Second, it *partially* specifies objects in the

Relational Theory and Data Independence: Unfinished Business.

© Copyright 2007, Dr. Tom Johnston.

Personal, non-commercial copies only are permitted.

Last revised November, 2007. Page 1.

Physical Layer, and associates them with those tables. This association constitutes the mapping between the Conceptual and Physical Layers.

This mapping is what provides *physical data independence*, which is independence of objects in the Conceptual Layer from objects in the Physical Layer. But much of the work to create Physical Layer *persistent* physical structures (files, indexes, hash keys) and also to create Physical Layer *non-persistent* physical structures (buffers, queues, stacks) is done automatically. The DBA neither needs to nor is able to influence many of the decisions which go into setting up the Physical Layer for a database.

DBAs are always able to direct the creation of *some* Physical Layer objects, however. This varies from one DBMS to another. But all come with SQL DDL dialects that offer different clauses of the CREATE TABLE statement that do influence the construction of the Physical Layer, e.g. clauses that define unique or non-unique indexes on non-primary key columns.

Three ANSI/SPARC Layers, Two CREATE Statements, One Problem.

So in effect, CREATE TABLE statements do three things, not two things as do CREATE VIEW statements. CREATE TABLE statements:

- Create base table schemas, the objects in the Conceptual Layer;
- Create physical storage structures, the objects in the Physical layer;
and
- Provides a mapping between those two sets of objects.

This lack of a Physical Layer CREATE statement might be considered an aesthetic blemish only, doing no more harm than destroying the symmetry of three CREATE statements, one for each of the three ANSI/SPARC layers. But we can nonetheless distinguish the conceptual from the physical aspects of CREATE TABLE statements. We can do it like this.

Relational Theory and Data Independence: Unfinished Business.

CREATE TABLE As a Conceptual And As a Physical Specification.

Both the External and the Conceptual Layers are accessible to the programmer and end-user, since SQL can be written against either views or base tables.¹ So any portion of a CREATE TABLE statement that can alter what is visible to SQL is a portion that defines an object in the Conceptual Layer. On the other hand, any portion of a CREATE TABLE statement that can be altered without altering what is visible to SQL, in any way, is a portion that defines an object in the Physical Layer. Let's call these the conceptual and physical portions of CREATE TABLE statements, respectively. For example, adding a column to a table is changing the conceptual portion because it alters the table as seen by SQL. Adding an index is a physical change because it does not.

Some DBMS vendors might consider this lack a good thing because, on balance, they believe that their own developers can do a better job than end-user DBAs at defining the best static and dynamic physical structures for the database.²

But this lack of a separate CREATE PHYSICAL STRUCTURE statement is *more* than an aesthetic blemish. It is symptomatic of an unfinished revolution. For by the most important criterion of all, relational products *fail* to provide physical data independence. Relational products fail to make denormalization unnecessary! Relational theory fails to define the conceptual structures that would, when implemented, realize *full* Physical Data Independence.

¹ By "SQL" in this context, I mean the SQL that programmers and end users write to access the content of tables or views. This is more formally called "SQL Data Manipulation Language". But I follow the convention of most authors, and use the unqualified "SQL" to mean "SQL DML". CREATE VIEW and CREATE TABLE statements, on the other hand, are also SQL. But they are SQL Data Definition Language statements.

² Teradata is famous for managing its physical data structures with little input from DBAs. For example, Teradata DBAs cannot define various kinds and sizes of buffer pools. The physical keys in Teradata databases are values computed by an internal hashing algorithm. DB2 UDB, by contrast, uses as primary keys what the DBAs define as primary keys, in the CREATE TABLE statements, and also gives wide latitude to the DBA in assigning buffer pools.

Relational Theory and Data Independence: Unfinished Business.

Relational theory, and relational products, fail to make denormalization unnecessary, and thus fail to support full Physical Data Independence.

Figure 1. How Relational Theory and Products Fail to Support Physical Data Independence.

Data modelers and DBAs know this for a fact. We are frequently faced with the need to improve the performance of on-line queries. And frequently, the best or only way to do so is to eliminate one or more of the joins in those queries. And how do we do that?

The only way to eliminate a join, against otherwise normalized tables, is to denormalize! Specifically, it is to introduce a second normal form violation, a partial dependency, or else to introduce a third normal form violation, a transitive dependency.

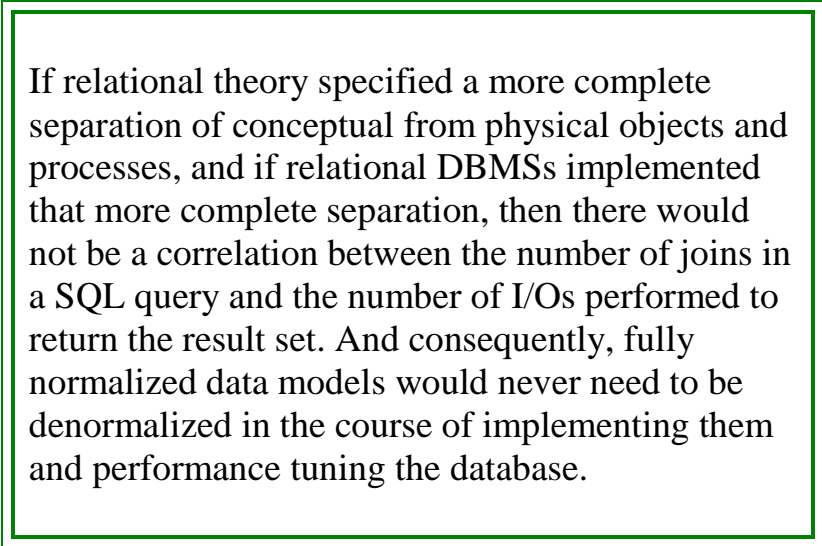
For example, illustrating a second normal form violation, to eliminate a join that gets the salesperson name on a Salesperson By Customer associative table, we add the salesperson name to that associative table, introducing a partial dependency. Since the fact that a specific salesperson identifier is associated with a specific salesperson name can now be recorded in the database as many times as that salesperson identifier appears as part of the primary key of the associative table, the database is denormalized, and duplicate facts are allowed into the database.

To illustrate a third normal form violation, we may choose to eliminate a join which gets the department name for an employee from the Department table by adding department name to the Employee table. Since department number (the primary key of the Department table) was already part of the Employee table, this introduces a transitive dependency into the Employee table.

Relational Theory and Data Independence: Unfinished Business.

A more direct way to see this failure to achieve full physical data independence is to focus on the join operation itself. For what is a join? It is, in actuality, a specification of both a logical and a physical operation. The logical operation is a mathematical operation on logical objects. This is how the join operation is defined.

But unfortunately, a join, as it is provided by relational DBMSs, is also a physical operation. This operation always involves accessing separate and distinct physical structures. The reason that extra joins slow down a retrieval is that the separate structures accessed by the join are usually so separate that they cannot both be retrieved in the same I/O operation. It is the extra I/Os that slow down joins.



If relational theory specified a more complete separation of conceptual from physical objects and processes, and if relational DBMSs implemented that more complete separation, then there would not be a correlation between the number of joins in a SQL query and the number of I/Os performed to return the result set. And consequently, fully normalized data models would never need to be denormalized in the course of implementing them and performance tuning the database.

Figure 2. In a Perfect World

This, in turn, would guarantee that a logically correct image of the database was always presented to the programmer and end user. It would be presented as a fully normalized set of schemas, in the Conceptual Layer, and a set of views derived from them in the External Layer.

Since data modelers and DBAs must work with the DBMSs and SQL dialects given to them by their vendors, the question is how best to cope with these unfortunate and unnecessary limitations. Current best practice is

Relational Theory and Data Independence: Unfinished Business.

simply to denormalize only when absolutely necessary, when required performance cannot be achieved by any other means.

What Can We Do?

[Notes only. I'll provide a more extensive write-up later.]

1. Don't wait for standards committees or vendors. This issue doesn't seem to be even on their horizon.
2. The bottom-line solution is to provide a purely syntactical, semantically neutral, set of physical schemas for data. This is what RDF attempts to do.
3. I made my own attempt, in 2004, in an eight-part series at (the now defunct). DataWarehouse.com, entitled *Stability and Flexibility in Databases*. (Links to these articles will appear in the DMReview.com archives sometime soon.)
4. I don't yet know enough about RDF to be able to compare it with my own solution.