

Modeling Matters: Logical Data Models, Denormalization and Conceptual Clarification: Part 1.

Dr. Tom Johnston
Mindful Data, Inc.

Originally published at DataWarehouse.com in June, 2002.
Reprinted here by permission of the senior editor at DM Review.

Some data modelers believe that only fully normalized models should be implemented in physical databases. If DBAs point out that a modest amount of denormalization can reduce response time for important queries from minutes to seconds, their response is "Fixing performance problems is what indexing, buffer management and hardware is for."

For example, Dr. John K. Sharp says that "Any structure that prevents valid data from being populated or allows invalid data to be populated must be considered an error."¹ I agree with the first part of the statement, but disagree with the second part. This article explains why.

The implementing DBA, and the data modeler, are at loggerheads because each is trying to optimize a different resource, and is willing to spend the other guy's resources in order to optimize his own. The resources that the DBA is trying to optimize are clear; they are, as the modeler noted, "indexing, buffer management and hardware". The resource that the modeler values so highly is nothing so tangible. It is whatever goodness comes from the implementation of a fully normalized relational data model.

The question is: what is that goodness? Why is implementing a fully normalized model so important? The DBA is trying to maximize the use of costly resources – disk space, main memory, I/O channels and CPU cycles. He is trying to stretch the use of the existing hardware platform as long as

¹ "De-normalized for Speed", Dr. John K. Sharp. *Journal of Conceptual Modeling*
www.inconcept.com/jcm. June 1998, Issue #3.

possible, to get the maximum value out of that investment before he has to replace it or upgrade it. What value does a fully normalized database have, that can offset the cost of these other resources?

A Database Free of Redundancy.

Many data modelers, I believe, think that the value of creating a fully normalized data model is to specify a database which is free of redundant information. The value of a redundancy-free database is that within it, it is impossible to store data which is inconsistent. The data may be wrong, of course; and no amount of data modeling magic can prevent that. But inconsistency is a serious matter, and a pervasive one.

When users retrieve inconsistent answers from a database, they begin to lose faith in that database. Users know that erroneous data can creep into databases, and that they themselves are often at least partially to blame. Instinctively, though, users seem to recognize that inconsistency is a different matter. Inconsistency is something that happens after data gets into the company's databases; it is the responsibility of *their* IT staff, of *their* DBAs.

The solution, many modelers would say, is to eliminate those inconsistencies by eliminating redundancy. Even though the cost of inconsistency is more difficult to quantify than the cost of additional memory, for example, the cost of inconsistency is unquestionably high, and worth spending some hardware dollars to eliminate.

Normalization is a way to identify data redundancies. If physically implemented, normalized data schemas are also a way to prevent those redundancies. A logical data model is a rendition of such schemas, less implementation details like DBMS-specific data types, secondary indexes and buffer utilization techniques.

But does normalization eliminate inconsistency?

Has Inconsistency Been Eliminated?

The first thing I want to point out is that even if all a company's databases were fully normalized, redundancy and inconsistency could still exist.

One way that normalized databases can contain redundant data is that normalization applies within a single database, not across multiple databases. The Acme Hardware Company may be both a supplier and a customer of our company. If it notifies us of a change of address, but does so as a supplier, the address will be changed in the Supplier database, but quite possible not changed in the Customer database. Even if both the Supplier and Customer tables, and their respective Address tables, are contained in the same database, it is possible that the address for Acme as a Customer will not be changed at the same time that Acme's address as a supplier is changed.

The source of this redundancy is a data modeling problem, but it is not a normalization problem. The Supplier and Customer tables, and the two Address tables, may well be fully normalized. The problem, instead, is a problem of *ontology* – of what types of things we choose to represent as entities in our data models. The two data models under discussion did not identify addresses as a type of thing to be modeled. Instead they identified vendor addresses and customer addresses, and modeled them as separate, unrelated entities. If the modelers had recognized the redundancy problems this would create, they could have created a single Address entity and linked vendors and customers with that single entity via a role indicator. Then Acme would have been related to a single row in the Address table twice, once as a customer and once as a vendor; and when that single row was updated, Acme would be associated with the new address under both roles.

The Principal Value of Normalization.

So if the objective of creating a normalized relational model is to build databases which cannot store the same item of data redundantly, then a fully normalized data model is no guarantee that the objective has been achieved. The modeler's argument with DBAs, then, can no longer be that fully normalized models must be implemented without modification because doing so eliminates redundancy, the source of inconsistency. Instead, the

modeler's argument is a weaker one, that implementing a fully normalized model eliminates *some* redundancy and inconsistency.

The second thing I want to point out, however, is that the elimination of data redundancy is not the main purpose of creating a normalized data model. Instead, the main purpose of normalizing a data model is to identify all the business concepts that are expressed in data components and in the structures they are assembled into, and the data constraints that implement the meaning of those concepts.

For example, in specifying the unique identifier for each entity, from the business user's point of view, we are clarifying what that entity means. Suppose a business firm has identified Contract as an entity, and defined a contract as "an agreement between our company and a customer in which we will provides services to the customer, for a specified period of time and for specified purposes, in return for specified payments". Now suppose that the subject matter experts in this company go on to tell us that the business identifier for contracts is contract-number plus contract-renewal-date.

The data modeler should reply that this sounds like the business key for a version of a contract, not for a contract. He could then suggest that some of the attributes and relationships already identified really belong to a true Contract entity, while the others will remain with this entity which should now be renamed a Contract Version entity.

It turns out, let us suppose, that legacy systems have never made that distinction, and that business users have worked with those systems for years, and perhaps for decades. The discussions to decide whether or not to accept the modeler's suggestion will probably be difficult ones, because he is introducing a new concept. In accordance with my gloss on "ontology" above, the modeler is suggesting that the users change their ontology.

No matter how we look at it, the issue is not a technical issue about data modeling. It is about the conceptual scheme of the business users, and whether or not that scheme should be changed. The suggested change, if adopted, will alter the data model, introducing the recognition of two types of things – contracts and contract versions – in place of a single type of

thing. Since the conceptual change being proposed is concerned with what types of things to recognize in the conceptual scheme and language of the business, it is proper to call it a question of the ontology used by that business.

Here is another example. Suppose that the data modeler has identified Salesperson and Customer as entities in the model. The business requirements already gathered indicate that the relationship is many-to-many, and so an associative Customer-Salesperson-Xref entity is created to represent that relationship. But notice that it is no mere modeling technicality that we are dealing with here. This business has told the modeler that its policy is to permit one salesperson to serve more than one customer, and also to permit one customer to be served by more than one salesperson.

We can easily conceive of either policy being different. Some businesses assign only one salesperson to each customer, but allow the salesperson to be assigned to several customers. Other businesses assign several salespersons to one customer, but their customers are so large that for each of those salespersons, that customer is their one and only customer. The first scenario describes a one-to-many relationship from salesperson to customer; the second describes a one-to-many relationship from customer to salesperson. This business, on the other hand, operates under the rule that there is a many-to-many relationship between customers and salespersons.

Next, the data modeler turns to the minimum cardinalities of this relationship. Can a customer exist without any salesperson assigned to him? Does the business have salespersons who are not assigned to any customers? Quite possibly, the initial set of business requirements do not answer these questions, and yet they are vital questions because they bear on what that business means by a customer and by a salesperson.

Once again, in attempting to develop a data model, the modeler is led back to concepts. What do you mean, you business experts, by a customer? Can someone or some enterprise ever be recorded as a customer in your databases before a salesperson is assigned to them? Can they remain on the database as customers after all salespersons are removed from being assigned to them? Similar questions, raised about salespersons, are similarly

questions about the conceptual scheme of the business, of what some of the basic concepts in that scheme really mean, and of what the relevant business policies and constraints are which enforce the meaning of those concepts in the data that supports them.