DATA MODEL VIEWS

David C. Hay
Essential Strategies, Inc.

ABOUT VIEWS AND THE CONCEPTUAL SCHEMA

This article describes a basic problem in data modeling: the lack of a method or tools for developing "views" of data models.

A data model (specifically, an "entity/relationship" model) has two purposes: first, a systems analyst uses it to confirm with prospective system users that he or she understands the nature of the business involved; second, a system designer uses it as the blueprint for the underlying structure of a new or revised system.

Data models are most often used to meet the second objective. They help data base designers visualize data base structure, and thereby to clarify their thinking.

They often fail to meet their first objective, however when they fail to account for different points of view of the company's data. They are either drawn in very concrete terms, to reflect the particular perspective of one part of the user community, or they are drawn in more abstract, general terms, to attempt to address more general issues. In either case, however, the views of some are not represented.

Originally, data base theory envisioned three distinct perspectives for the data in a data base, as shown in Figure 1. First, the *conceptual schema* represents the structure of data as they exist throughout an organization. The facts in the conceptual schema are true everywhere. Second, the *internal schema* represents the structure of data as they are stored physically on the computer. By extension, the internal schema also describes the representation of the networks and hierarchies used by some data base management systems. Finally, the *external schema* represents data as seen by each user.

This has proven a useful view of things, and it stimulated the development of relational data base management systems. With a relational data base management system, you could now describe the conceptual schema directly to the computer as a structure of tables and columns. The software would initially create the internal schema, which could then be manipulated independently of the conceptual one. Moreover, you could specify the "views" individual users have of the data. The software would keep track of the relationships between these and the conceptual schema.

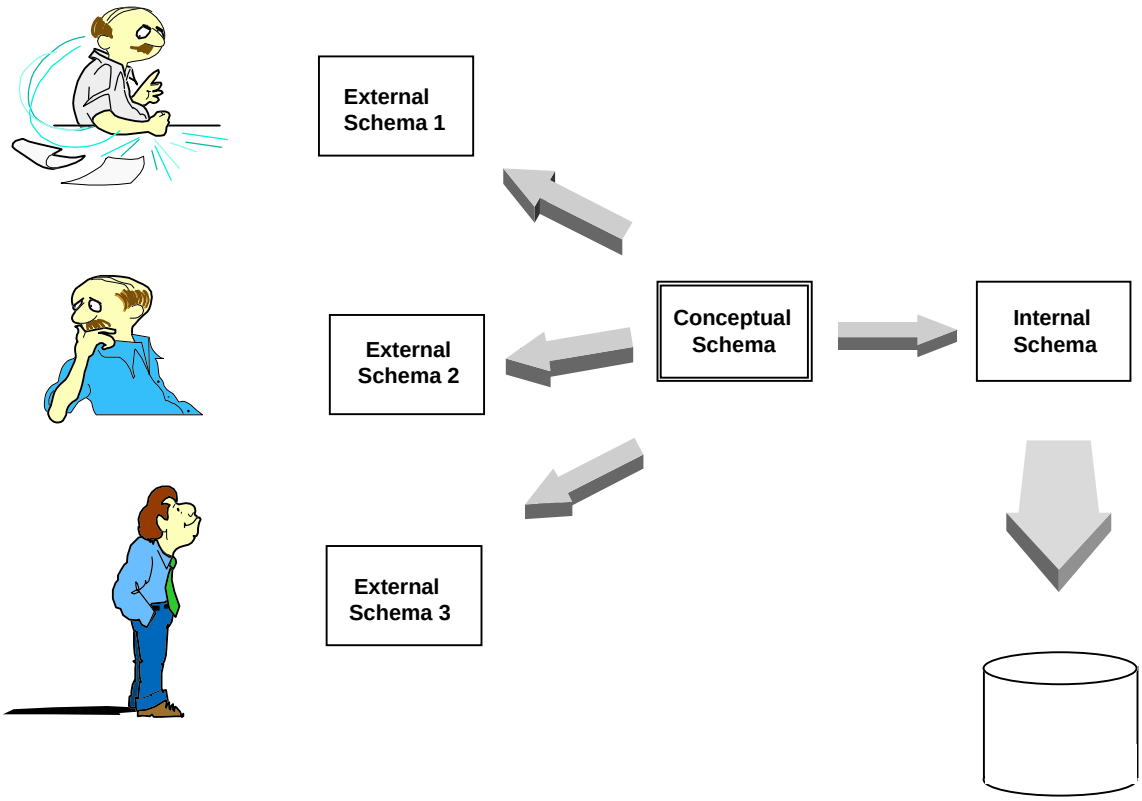

Figure 1: The Three-schema Approach

In practice, however, relational tools haven't been used this way. Early relational data base management systems were slow, even with the tools available for internal tuning, so table structures were de-normalized and departed from the conceptual model. The design of tables and columns became part of the internal schema design, just as the design of networks and hierarchies were part of the internal schema design under earlier technologies. Even though the data base management systems have gotten faster, the practice remains.

Even where they are appropriate, SQL views often have not been used, "for performance reasons." Views are still often implemented with application programs — as they always have been. Where SQL views are used, with the database no longer representing the conceptual schema, they are now linked directly to the internal schema.

The only place where the conceptual model still exists is in the data model. The data model is a drawing — intended to represent the business in its most fundamental sense — showing entities as "things of significance about which an organization wishes to hold information".

Data modeling itself, however, as practiced, is not immune to biases toward both the internal and external schemata: Some analysts will draw models of an external view, creating entities for each concrete thing seen by users, without regard for underlying similarities and principles. Others use the data model as a database design tool, simply reflecting the actual or intended database design structure.

A true conceptual model, on the other hand, will show only things which are fundamental to the business, of which most of the things people see are examples.

All of this is merely another expression of the original problem. Different people in the organization have different views of data. If a true conceptual model can be drawn, it will not necessarily be recognizable to all in the enterprise. If the model is more concrete, in deference to a particular department, it will reflect a particular external view, and people in other departments may either not recognize it or disagree with it. Even the physical database design represents but another view of the data, that is entitled to representation.

As shown in Figure 2, the three-schema architecture has gone astray.

So data modeling, our tool for developing conceptual models, has become removed from the process of specifying and developing systems. When the data model represents the conceptual schema, the external schemata are no longer connected to it, and they are not available at all in data model form. And while the conceptual schema often is the starting point for physical data base design, it is very difficult to keep the link current as requirements change.

What we need is the ability to return to Figure 1. If the facilities to support it were available, an analyst in early discussions with users would take advantage of data model graphics and sentences to build the models in the users' concrete terms. The analyst would then translate the result into generalized, corporate conceptual models.

When the designer then maps the conceptual data model to a data base design, he or she would map the external data models directly to SQL views and screens. In other words, a *default view* design could be generated along with a *default data base* design.

CASE tools, as they exist today, do not support the idea of data model views. At best, some (though by no means all) CASE tools allow the user to draw subsets of the total set of entities and relationships. This facility can be used to selectively draw entities that pertain to a particular part of the business. Where one can achieve a specific view by simply selecting entities to show, this is adequate.

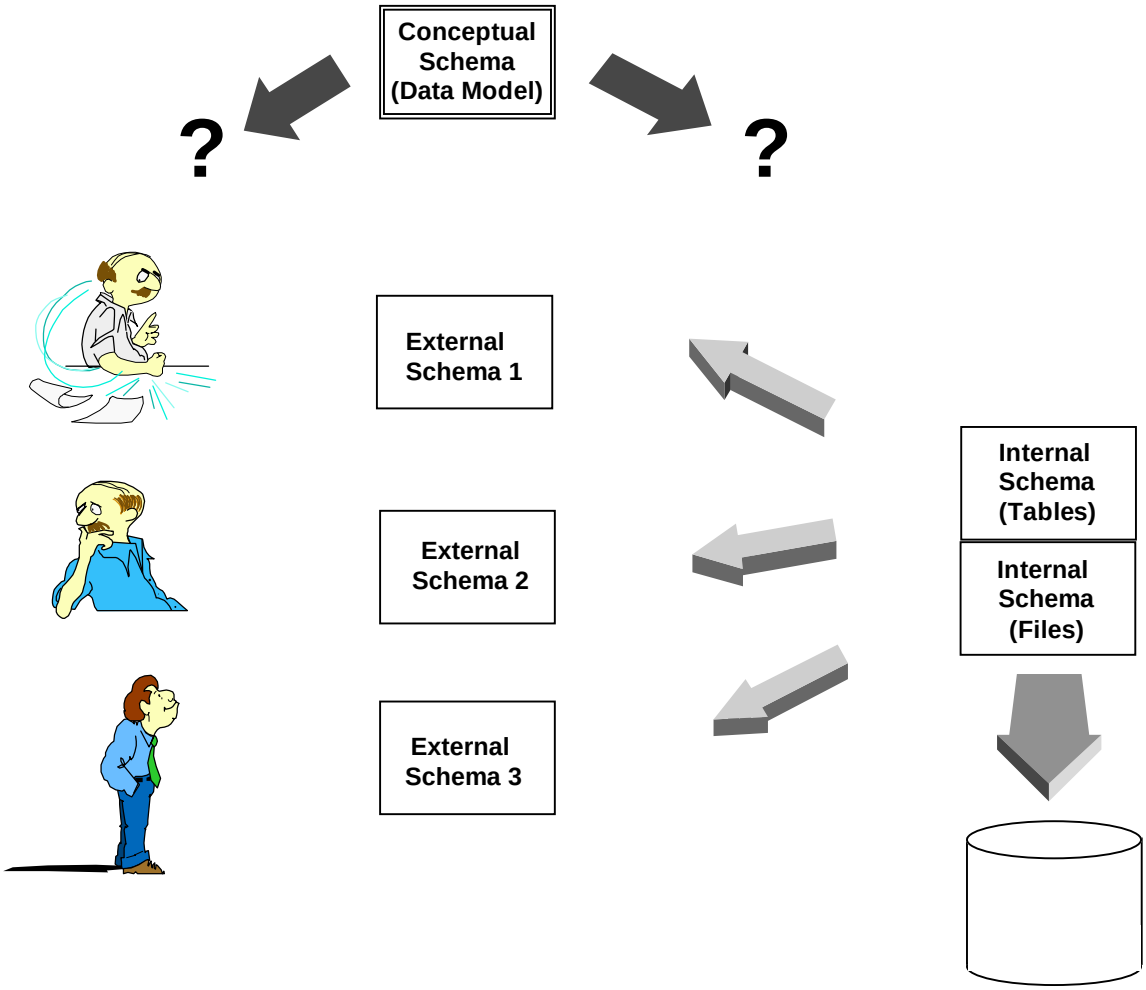

Figure 2: The Conceptual Schema Adrift

No CASE tool (in your author's experience), however, is flexible enough to allow one to represent a view of a data model where the relationships or entity names in the view are *different* from those in the underlying model. We need such a facility — along with the ability to document the relationships between this view and the underlying complete model.

The following sections present some examples of situations where this view mechanism would be useful.

EXAMPLES OF VIEWS

At least five situations give rise to the need for views of data models:

- Hierarchies
- Departmental views

- Combining entities and relationships
- Cross-departmental views
- Meta model views

Hierarchies

First, unlike data flow diagrams, data models are not inherently hierarchical, which makes it difficult to produce "summary" or "high level" data models. Upper management is often unwilling to sit through presentations of excessive detail, but there is not a convenient, standard way of showing just the most important elements.

As mentioned above, one can select entities for presentation, based on their importance to the audience, but simply failing to include a low level entity on a diagram may not be enough: If entities are left out of the presentation, relationship definitions themselves change. Figure 3 shows a relationship between two high-level entities. WORK ORDER and PERSON appear to a manager to have a simple many-to-many relationship, but at lower levels of detail the relationship is much more complex .

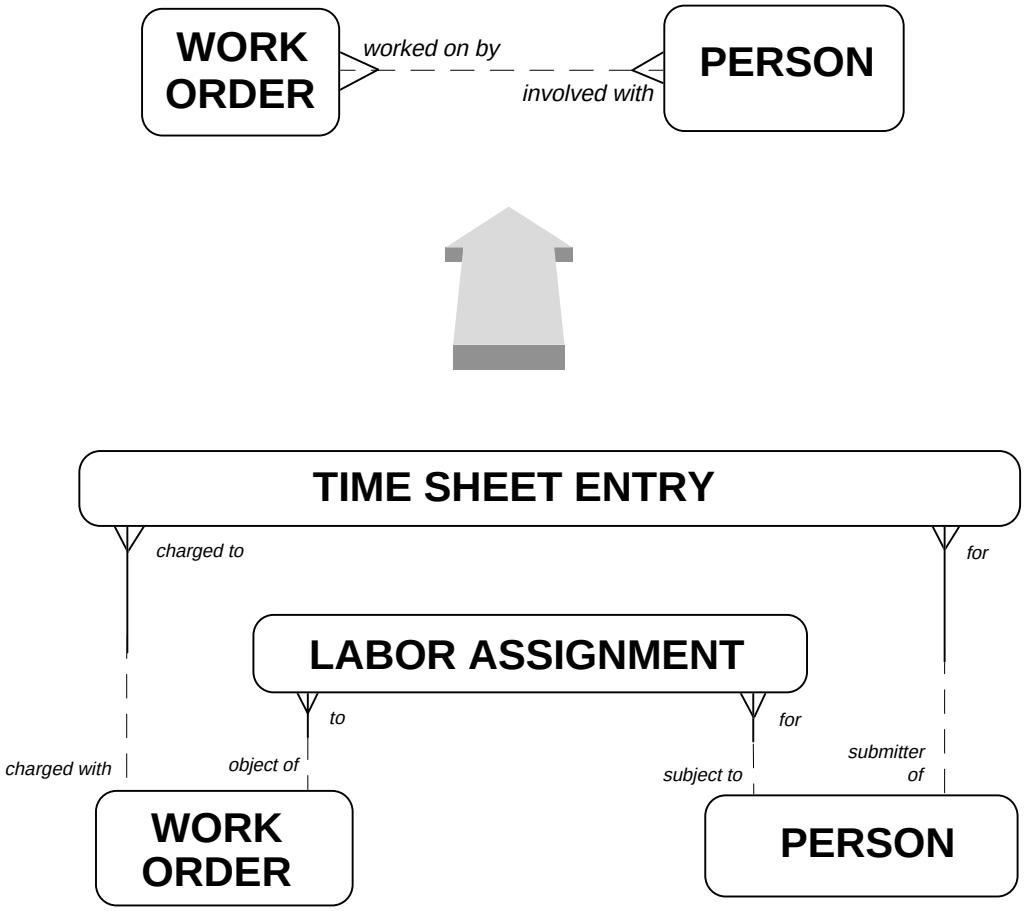

Figure 3: A Hierarchical View

(It is not always easy to judge what constitutes a "most important" entity, by the way. Some entities are clearly "high-level" or "detailed", but the significance of others depends on the manager looking at the model. In this article, however, we are discussing the ability of the *tools* to represent views — the abilities of the *analyst* to represent views *intelligently* is a separate issue.)

Note that this is not simply a matter of eliminating the TIME SHEET ENTRY and LABOR ASSIGNMENT from the summary diagram in Figure 3. The relationship pair in the summary diagram ("Each WORK ORDER may be *worked on* by one or more PEOPLE", and "Each PERSON may be *involved with* one or more WORK ORDERS") is *different* from those in the detailed diagram. While you could simply add the summary relationship pair, and show or not show the line when appropriate, we currently have no mechanism to show that these different relationships are logically equivalent.

Departmental Views

The second need for data model views comes from the fact that by building diagrams to describe a company as a whole, we often produce models that are not in terms each department can readily understand. Different departments may look at similar things in completely different terms.

Regardless of the management level concerned, all people work in an environment of concrete things: Users know about "parts", "instruments", "sub-assemblies", etc. They don't know about the more general "items" or even (except for the accountants) "assets" that would appear on a more enterprise-wide model. At the very least, any entity on a model could list examples, but better yet would be to make the entity names themselves reflect the concrete world of the user. Certainly one must portray relationships as the department sees them.

Where different departments have different names for the same things, each should be able to see a model with its own names on it. In Figure 4, PRODUCTION FACILITY and MEASURING DEVICE are two broad categories of hardware (ITEM TYPE) in a plant. Individual departments, however, deal with subsets of these things. The maintenance department deals with PIECES OF EQUIPMENT, the LABORATORY deals with INSTRUMENTS, and the process control department deals with TAGS.

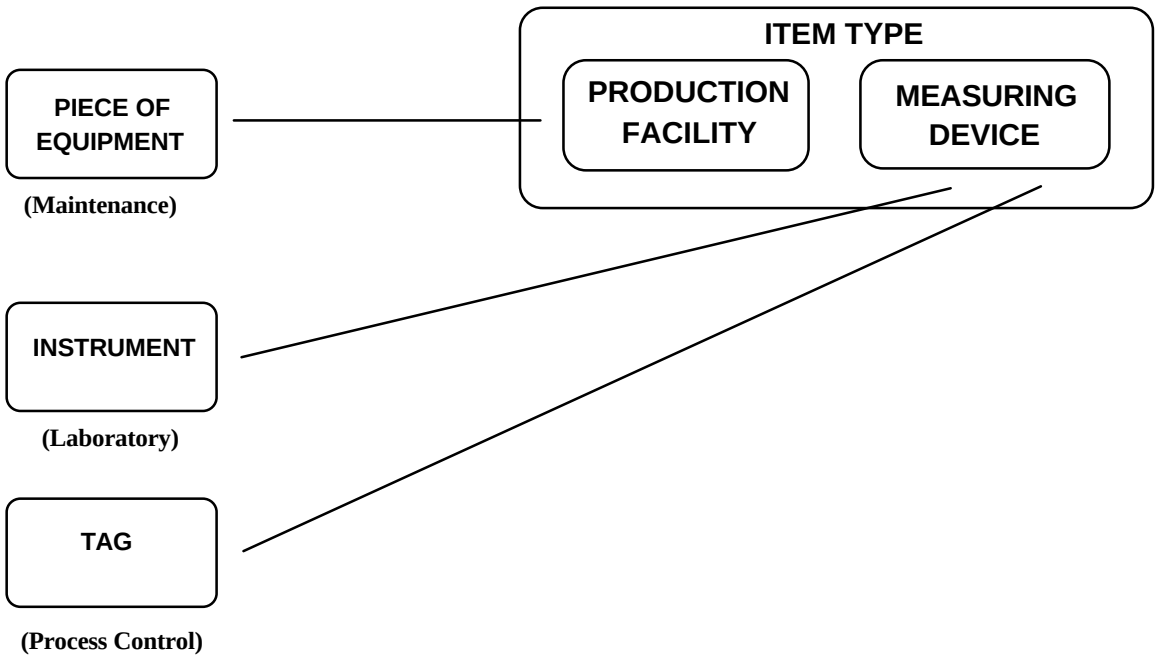

Figure 4: Item Types

What is needed here is the ability to specify that a particular department's entities and relationships exist — showing these in a data model — and to specify also the links between these and a conceptual model whose entities and relationships may look quite different. The view entities and relationships are manipulated in exactly the same way as real entities, but the data dictionary understands how they are different.

Another example is shown in Figure 5: A maintenance engineer may view a "ROTATING PART" as an entity of significance. In the conceptual model, this is simply another kind of EQUIPMENT. A data model view language would permit definition of the following:

ROTATING PART <= "EQUIPMENT,
(each of which must be *of* EQUIPMENT TYPE)
WHERE
EQUIPMENT TYPE.DESCRIPTION
= 'Rotating part'" ;

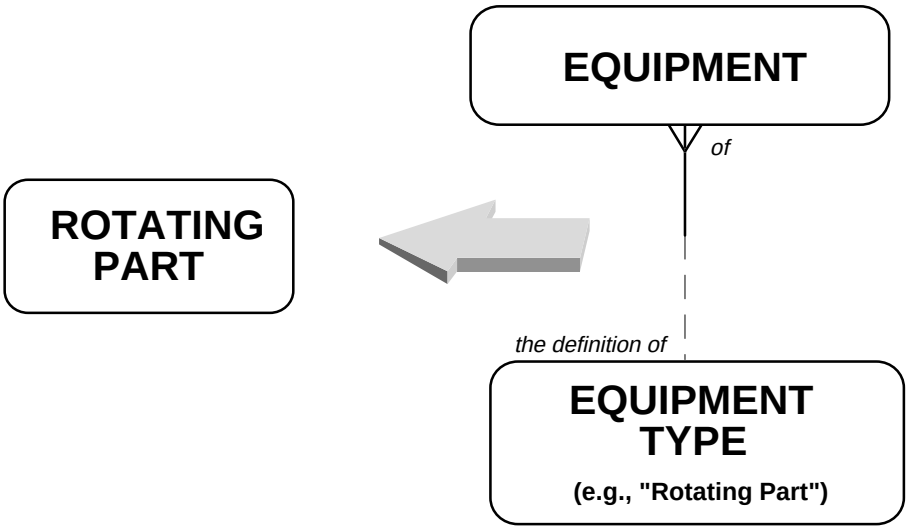

Figure 5: A Departmental View

Relationships change as well in different views. For example, the departmental model may show that a PRODUCT is of *one and only one* PRODUCT TYPE, but from the point of view of the company, a PRODUCT may be of *one or more* PRODUCT TYPES.

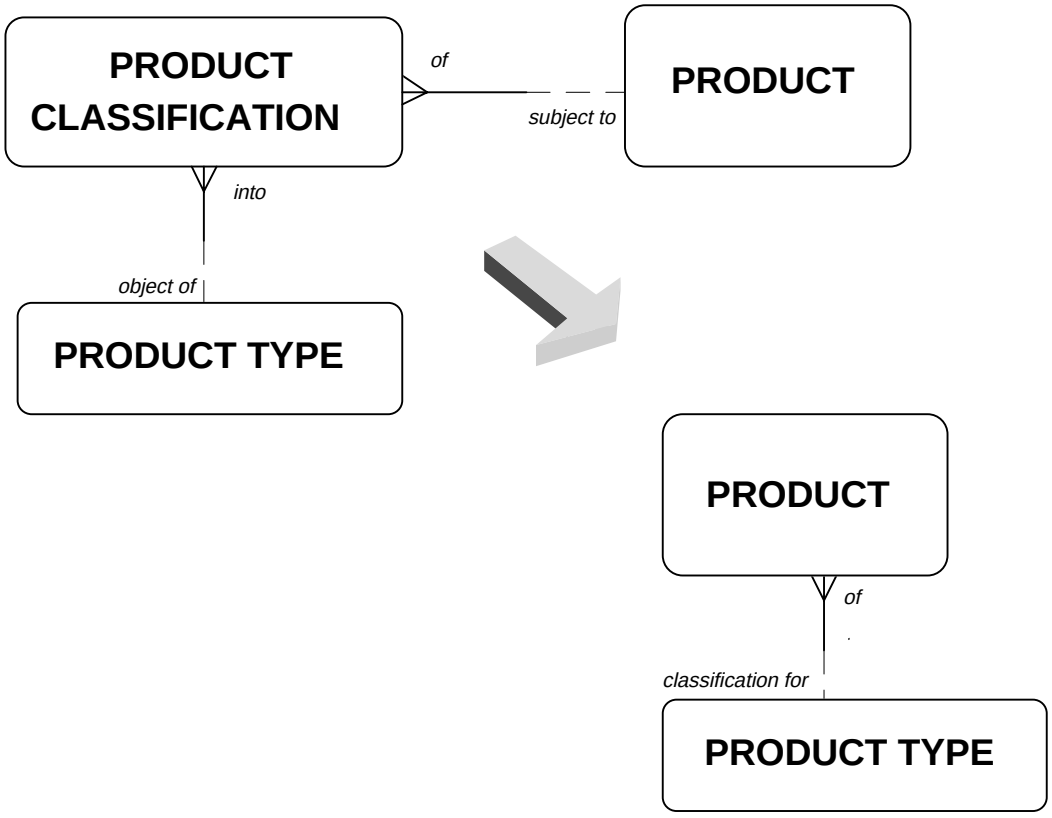

Figure 6: Another Departmental View

The mechanism for keeping these two views synchronized is what we seek here. (See Figure 6.)

In both these cases, simply hiding entities is not sufficient. Relationships are different between the views.

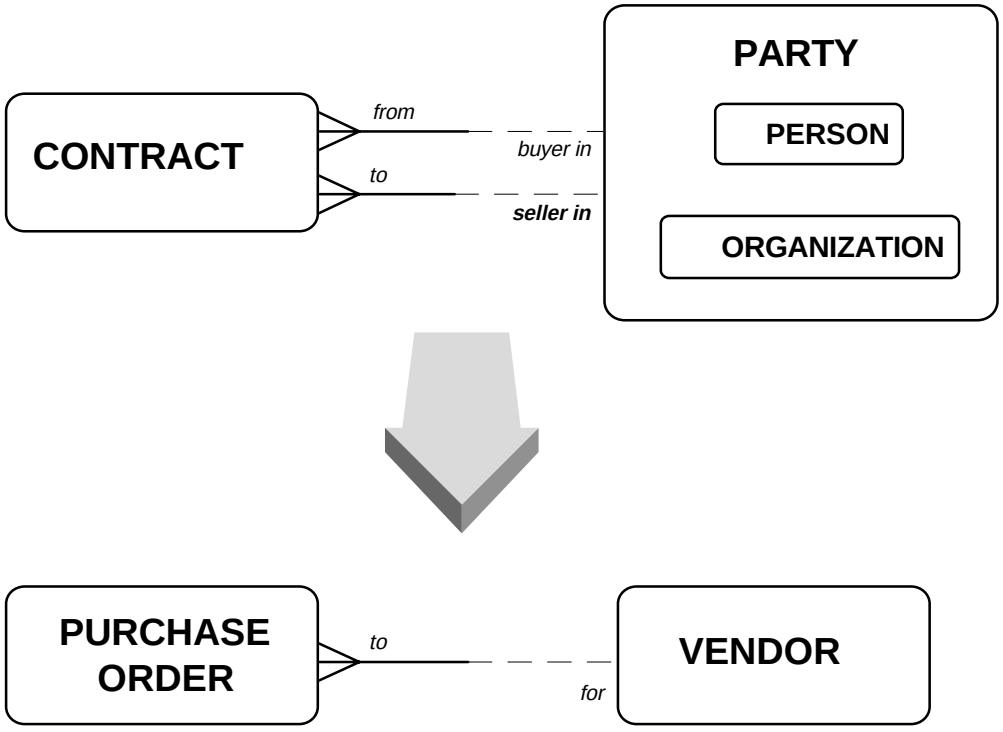

Figure 7: An entity Plus a Relationship

Combining Entities and Relationships

The third area where views of data models would be useful is the practice of incorporating a thing's relationships into the definition of the thing itself. This is a practice that conceptual data modelling tries to stamp out, but that audiences insist on doing.

The most common example of this is the entity VENDOR, which modelling purists know is only a PARTY (a PERSON or an ORGANIZATION) which is *a seller in* a CONTRACT. (See Figure 7.) That is, the word "vendor" contains within its definition not only the thing itself (a person or organization), but its *relationship* with other things.

Many people are more comfortable, however, with the use of VENDOR and CUSTOMER entities (probably because this is the way we have always built purchasing and sales systems), even though the underlying entities (PERSON and ORGANIZATION)

are the same, whether they are buying or selling. In the interest of harmony, it should be possible to portray the more familiar entities.

Again, a view syntax would allow us to say:

VENDOR <= PARTY
WHERE
PARTY is "seller in" one or more CONTRACTS;

Cross-departmental Views

A fourth area where data modelling views would be useful concerns entities that relate to many or all aspects of the business, in a way that redefines local entities. An accounting transaction, for example treats quite different entities (such as DEPARTMENT, or ASSET) as "cost centers". (See Figure 8.)

In some cases a COST CENTER may be an EXPENSE ACCOUNT for a particular DEPARTMENT *and* ASSET, or it may just be the set of all expense accounts for a DEPARTMENT. The model could be drawn so that each COST CENTER must be *for either* one EXPENSE ACCOUNT, or *for* one DEPARTMENT, but that belies the fact that people who are concerned with COST CENTERS don't want to know about EXPENSE ACCOUNTS, and vice versa.

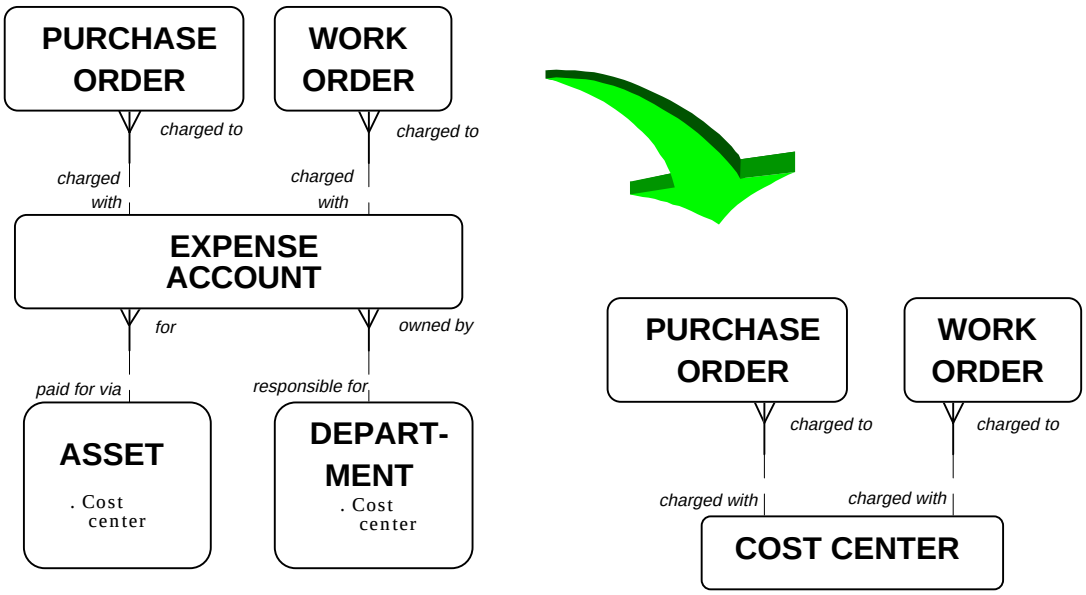

Figure 8: Cross-company Views

In another example, labor and parts usage charged to a work order require complex models in their own right to describe accurately all the relevant relationships specific to each. But to a project manager, they are simply "resources".

Meta Model Views

The real world is not, in fact, relational. Situations arise where different occurrences of an entity have different attributes, depending on the category of the occurrence. This could be handled by the use of sub-types, except in those cases which are very dynamic, with categories being added and deleted frequently. An example is PRODUCT, where the attributes of a fruit are quite different from the attributes of a computer.

The solution to this problem is shown in Figure 9. This model defines attributes for each ITEM TYPE, via one or more ATTRIBUTE ASSIGNMENTS. Each ATTRIBUTE ASSIGNMENT is *of* one ATTRIBUTE *to* an ITEM TYPE. A VALUE *of* the ATTRIBUTE can then be defined *for* an ITEM *of* that ITEM TYPE.

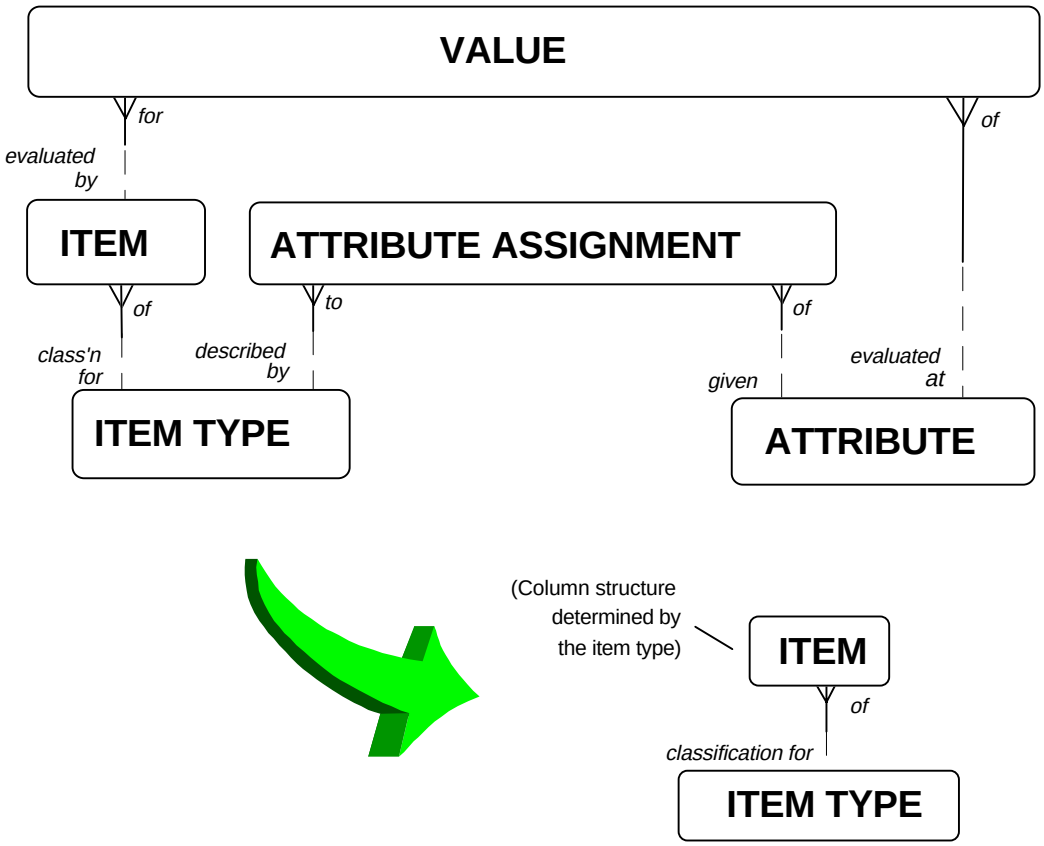

Figure 9: A meta model of a variable-length table

This model is too abstract for many audiences, however. Many people only want to know that an ITEM is *of* one and only one ITEM TYPE, and that the format of a particular ITEM will depend on its ITEM TYPE. The intricacies of the model that accomplishes this in a relational environment are of no interest. A CASE tool could present this, even though the dictionary contains the more complex underlying model.

Nothing in the existing tools prevents us from drawing meta-models such as this one. What is missing, however, is the ability to link this model to one that shows ITEM the way everyone sees it -- as a single entity with variable length rows.

RECOMMENDATIONS

In each of these cases, a systematic way of presenting these views of data would significantly increase data modelling's power as a tool for managing data base design.

CASE tools should officially recognize the concept of "view" and have a language for describing one, as presented here. Among other things, this would involve:

1. Separating drawings from their underlying model. This would allow a user to select which entities and relationships would appear in a particular drawing. This is a basic requirement which some CASE tools allow for now.

2. Making it possible to select the synonym to be displayed as the name of an entity in a particular drawing.

3. Making it possible to define a "virtual entity" whose definition is derived from other entities and relationships.

4. Making it possible to define a "virtual relationship" in terms of (or as a synonym for) one or more other relationships and, taking into account any intermediate entities.

5. Documenting the links between virtual and real entities and relationships, and manipulating them in reports and selection criteria. ("Show me the details behind this relationship...") It would be a nice touch (although not necessary) also to indicate such a virtual object on the drawing. For example a "(v)" could be placed next to the object name.

6. Making it possible to show a sub-type entity without having to show its super-type. Indeed, different applications should be allowed to own different sub-types.

These are only the most obvious requirements. Clearly the full implications of this idea have yet to be explored. Your author would welcome comments from anyone who has struggled with this problem and has further ideas on how to address it.

ABOUT THE AUTHOR

David Hay is the President of Essential Strategies, Inc., a management consulting firm specializing in Computer-aided Systems and Information Engineering. He has over twenty

years experience in information management, specializing in the design, development and implementation of interactive database applications, primarily in manufacturing. For the last six years, he has been a CASE consultant. He has done strategic information planning using data modeling and other modeling techniques in a wide range of industries, including, among others, news gathering, clinical research, oil refining, and land management.