CHOICES IN PRACTICAL DATA DESIGN

W. Kent

IBM General Products Division
Santa Teresa, California

ABSTRACT

Current data design methodologies tend to generate a single design for an application, neglecting many other possible designs. This is partly because user requirements have to be expressed in semantic models which are too closely coupled to the data design. Failure to recognize the variety of designs which can represent the same facts also hampers other data administration functions which depend on adequate data documentation. Such functions include application planning, redundancy management, and information center services. And there are even implications for the design of high-level interfaces to diverse data.

In this paper we make some initial attempts to catalogue the range of such design options, and to suggest how design methodologies might be enhanced to exploit the options. We also point out that a similar range of options exists for the semantic enterprise descriptions which serve as input to such methodologies.

1.0 INTRODUCTION

1.1 Multiple Design Options

A given set of facts can be expressed in many different configurations of data elements. Some of the configurations can be obtained by normalization [Kent 81a] and denormalization [Schkolnick], which are techniques for regrouping a given set of data elements. But the range of design options is much larger than that. The initial set of data elements is itself variable; different kinds and numbers of data elements could be used to express the same facts. Experienced data designers generally take advantage of many such options in practice, but the options seem to be neglected in current methodologies for data analysis, design, and documentation. We rediscovered these design alternatives in the course of some work on a data documentation tool [Kent 81b].

To illustrate the kind of variability we have in mind, consider that the weights and prices of parts might be kept in any of the data configurations shown in Figure 1.

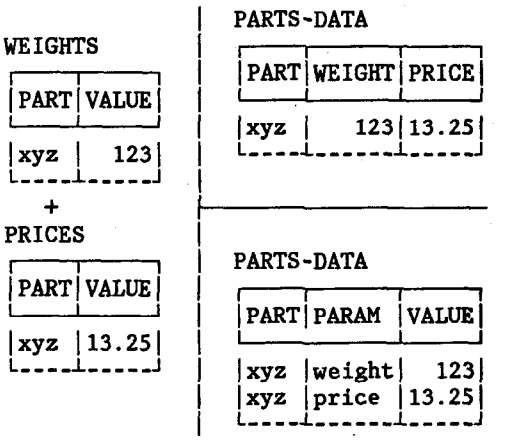


Figure 1. Three alternative designs.

1.2 Consequences

These observations have consequences for data analysis and design, and also for the techniques and uses of data documentation.

In the first place, most designers are probably not aware of all the available design options. A catalog of such options would thus in itself be a valuable aid in data design.

Beyond that, we observe that current methodologies and tools do not consider all the design alternatives. They tend to lock in on one logical data design generated mechanically from a given enterprise description. At best, they might consider the alternatives which can result from this initial design via normalization or denormalization. The results may be suitable for the novice designer, or for small or "quick and dirty" applications. But the experienced designer is likely to be disappointed, since he often can see that alternative designs would be preferable.

(By "enterprise description" we mean a data-independent description, expressed in terms of entities and relationships, of something for which data will be maintained in some data store. A "data design" is some configuration of record formats, including record type names, field names, key specifications, and field representations.)

Design methodologies and tools would be more useful if they considered all the relevant design options. At first, such tools are likely to yield a bewildering array of designs for the user to choose from. But over time we should be able to formalize the criteria for choosing among such options. A few are suggested in this paper. These can eventually be incorporated into more sophisticated methodologies and tools that help the user select the best design.

One of the reasons that the methodologies generate single designs is that the user's requirements have to be expressed in semantic models which are too closely coupled to the data design. They use a form of entity/relationship model in which entities and attributes are little more than thinly-disguised records and fields. Furthermore, entity types are usually required to be disjoint (no subtypes or overlapping types), so that they already correspond naturally to record types. By the time a user has specified his enterprise description in these terms, he has already made many of the difficult design decisions. Furthermore, if he wants to modify his logical data design, it turns out that he has to tinker with his enterprise description.

It would be useful if such methodologies and tools would adopt a more data-independent form of entity/relationship model, being more neutral to any particular implementation in terms of records and fields. The tools could then more readily take over the design burden from the user. The description of such a model is beyond the scope of the present paper, but one approach is sketched in Chapter 11 of [Kent 78].

The responsibilities of data administration go beyond the design of databases. There are a number of functions which depend on the proper documentation of data. Many of these require the ability to determine that the same information exists in two different contexts. One consequence of our observations is that such correlation can be quite difficult. Two "data things" may refer to the same piece of reality, and yet have substantially different data designs (recall Figure 1, as an example). If we are limited to simple matching techniques, such as checking if the two things have the same data elements, we are not likely to find the match. What is needed are more sophisticated documentation techniques, along the lines suggested in [Kent 81b] and [Kent 82], and more sophisticated correlation techniques, perhaps based on the transformations suggested in the present paper.

For example, existing data that might be sharable by a new application may not be discovered if the search is conducted in terms of matching data elements. I.e., if we simply look for data elements in the dictionary that match the data elements designed for the new application, we will miss cases where the same information has been implemented in a different configuration.

In the same way, redundancies in implemented data may be overlooked if we simply search for matching data elements. Also, possible interfaces between applications will be missed if we simply look for matching data elements. And the suitability of one system as a potential replacement for another will not be correctly assessed if we simply look for matching data elements.

One function of an information center is to find information requested by people not familiar with the data structures. This can't be done without knowing all the data designs in which such information might be implemented.

Beyond data design and documentation, our thesis has implications for high-level interfaces to data, and especially to distributed data. Such systems will, in general, have to provide some kind of uniform access to inhomogeneous data, implemented in a variety of data models under a variety of database management systems. Our observations add a new dimension to "inhomogeneity". We usually think of inhomogeneous databases as being built in different data models under different database management systems. We now have to recognize that databases using the same data model under the same database management system can still be inhomogeneous, if different data designs were used to implement the same facts.

1.3 Overview

In subsequent sections, we present:

- Examples of the design options.
- Ideas for a methodology to generate such options.
- Some observations on a parallel set of options in the enterprise description.

We use the following conventions in the illustrations of record formats:

EMPLOYEE

EMPNUM	SSNUM	NAME
111222	98765	Smith

INVENTORY

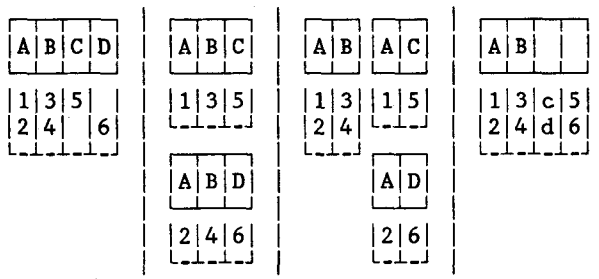
PART	WAREHOUSE	QUANTITY
Q33A	Central	950

EMPLOYEE and INVENTORY are record type names. "====" indicates keys: EMPNUM and SSNUM are each an alternate key, while PART and WAREHOUSE together form a compound key. The bottom row shows sample field values.

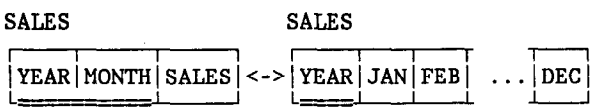
2.0 THE KINDS OF CHOICES

2.1 Logical Record Alternatives

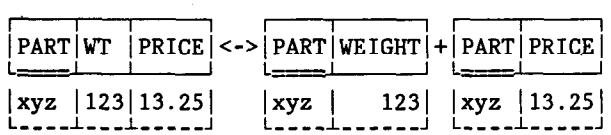
Many of the design options take the form of record transformations, i.e., one set of record formats can be transformed into another set, more or less equivalently. Some such transformations are suggested in this schematic overview:



In many cases, information occurring as field values in one design may occur in field names or even record type names in other designs. Figure 1 was one such example. Another is:



Records can be split or combined both horizontally and vertically. A simple example:



A more complex combination of all these options is shown in Figure 2, which provides five different ways to deal with subtypes. All employees (in this simplified case, accountants and salesmen), have departments and jobs, but only salesmen carry products.

In most of the cases we will consider, the alternatives are not exactly equivalent. There are secondary characteristics which might make one configuration or another more suitable for a given application. It is precisely to the point that a good methodology or tool should know about such secondary characteristics, and help the user to evaluate them in choosing his final design.

One such characteristic pertinent to the example of Figure 2 might be called "sparseness". Are most of the attributes applicable to most of the entities? If very few employees were salesmen, and/or there were many fields that applied only to salesmen, then form (a) would be very sparsely filled, i.e., there would be lots of blank space. One of the other forms might be preferable.

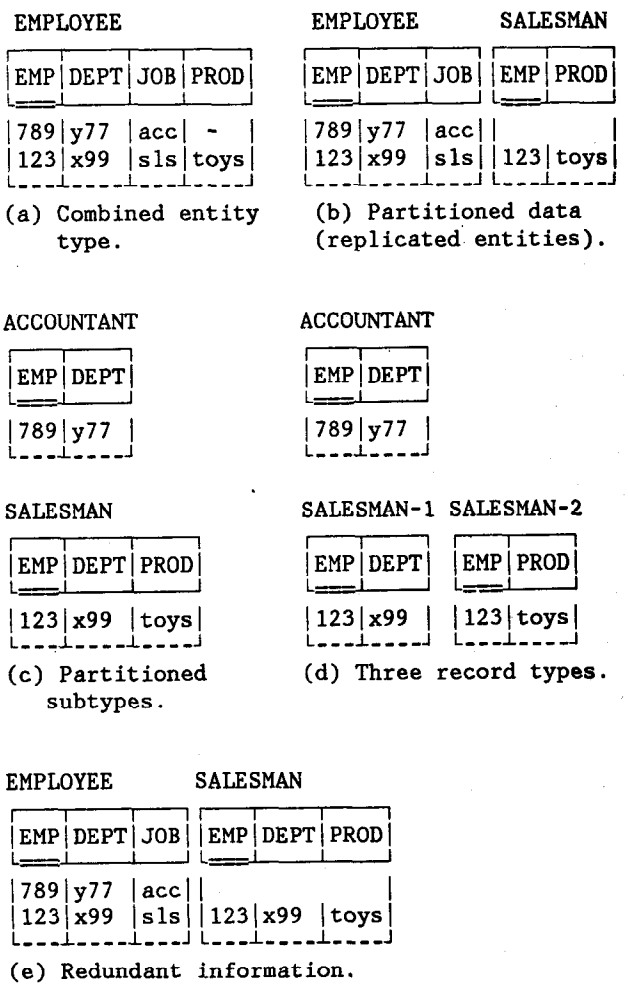


Figure 2. Five choices for subtypes.

Another consideration has to do with the management of "master lists". The EMPLOYEE record type provides a simple master list for control of all employees. There is exactly one record for each and every employee; any employee not in this list is not in the database at all. With the other forms, it takes extra work to achieve this same level of control, if it is needed. In form (b), there could be a salesman record with no corresponding employee record. In form (c), the same employee number might show up for both an accountant and a salesman. Forms (d) and (e) can have similar problems.

Without an EMPLOYEE record type, it becomes difficult to retrieve all employees. In forms (c) and (d), one has to know all the job types, and retrieve the corresponding records.

Also, the EMPLOYEE record type makes it possible to insert a new employee before his job is established. In forms (c) and (d), if we don't know the employee's job we can't insert his record.

Sometimes the subtypes are not explicitly distinguished or named. The EMPLOYEE record in Figure 2(a) might not have an explicit JOB field. Then form (a) is the only one that works. If necessary, the job can be implicitly determined, e.g., by a blank PROD field. Form (a) is the general mechanism for handling information which is not applicable to all occurrences of an entity type, when there is no desire to explicitly designate entity subtypes. For example, a maiden name field is not applicable to all employees, and hence is blank in many cases, but there may be no separate field to distinguish between married females and other employees. If needed, that could be inferred from non-blank maiden names.

Finally, there is an important criterion which applies to this and many other cases: the "fixed population" constraint. Form (a) is equivalent to the other forms only if certain constraints are specified and enforced. In this case, the JOB field must be restricted to the values "acc" and "sls" (accountants and salesmen). Otherwise, updates to the data could introduce jobs not present in the other forms, i.e., not corresponding to any of the other record types.

Forms (a), (b), and (e), with an explicit JOB field, are appropriate when we prefer to keep the list of job types open-ended and free to grow. Forms (c) and (d), with a distinct record type for each job, are appropriate when we wish to constrain the set of jobs.

Still another set of examples is provided in Figures 3-5, showing three database designs for the same application. The first is a straightforward design. The second one maintains all relationships between employees and departments in one parametric form. Essentially some field and record names have migrated into the data.

EMPLOYEE

EMPNUM	NAME	DEPT	SAL
123	Dick	x99	500
456	Dick	x99	600
789	Dick	y77	500

DEPARTMENT

DEPT	MGR	AUDITOR
x99	456	789

(others omitted)

SSNUMS

SSNUM	EMPNUM
999	123
998	123
997	456
996	789

LOANS

EMPNUM	DEPTNUM
123	z88
123	y77

Figure 3. A straightforward record design.

The third design (Figure 5) corresponds to an implementation of a binary relation model. The last two columns form a composite identifier. All binary facts can be represented here as entity-relationship-entity triples.

This example provides an excellent illustration of our concerns: if these three databases had actually been implemented, to what extent would it be possible to automatically detect the redundancy among them?

EMPLOYEE

EMPNUM	NAME	SAL
123	Dick	500
456	Dick	600
789	Dick	500

EMP/DEPT FACTS

EMPNUM	CONNECTION	DEPT
123	assigned	x99
456	assigned	x99
789	assigned	y77
123	loaned	z88
123	loaned	y77
456	manages	x99
789	audits	x99

SSNUMS

SSNUM	EMPNUM
999	123
998	123
997	456
996	789

Figure 4. Consolidating all facts connecting employees with departments.

EMPLOYEE FACTS

EMPNUM	FACT	TYPE	VALUE
123	alt-ident	ssnum	999
123	alt-ident	ssnum	998
123	name	name	Dick
123	assigned	dept	x99
123	earns	money	500
123	loaned	dept	z88
123	loaned	dept	y77
456	name	name	Dick
456	assigned	dept	x99
456	earns	money	600
456	alt-ident	ssnum	997
456	manages	dept	x99
789	name	name	Dick
789	assigned	dept	y77
789	earns	money	500
789	alt-ident	ssnum	996
789	audits	dept	x99

Figure 5. Consolidating all facts about employees.

2.2 Ideal vs. Efficient Designs

The following alternatives generally go beyond the objectives of logical data design. In many cases, the trade-offs are between so-called "update anomalies" and efficiency.

2.2.1 Alternative Meanings

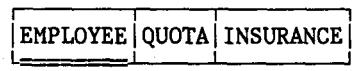
Sometimes a field can have several alternative meanings, varying from one record occurrence to the next. This is especially useful with facts that are mutually exclusive.

The motivation is simple: in the "idealized" design, mutually exclusive facts guarantee that every record will have a blank field. The idealism may not always justify the unused space.

For instance, suppose that:

- Salesmen had quotas, in dollars.
- Non-salesmen had life insurance, also in dollars.

A simple logical design yields:



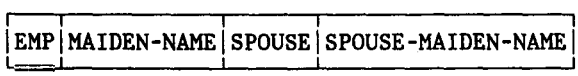
in which every record would be blank in either the second or third field.

The alternative practical design would be:

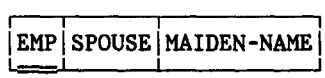
in which the Q/I field contained quotas for salesmen and insurance amounts for non-salesmen. Of course, there has to be some way of identifying which employees are salesman. There might be a distinct JOB field in the record, or it might be encoded somehow in the employee number.

A similar example gives rise to a different kind of trick. Suppose we had employees and spouses and maiden names (let's simplify by assuming everyone's married).

A simple logical design yields:



It doesn't take long to realize that there will be at least one blank field in every record. The smarter design is:

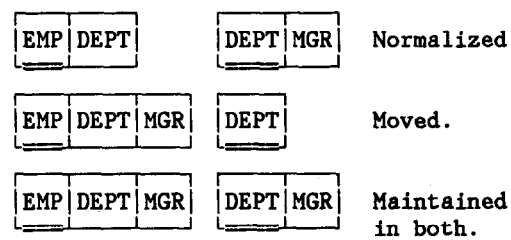


in which the last field means "maiden name of employee or spouse, depending on which is a married female". The sex of the employee might be indicated in a distinct field in the record, or it might be encoded in the employee number.

This differs slightly from the previous example involving quotas and insurance. In that case, we had different facts about the same object. Here, in a sense, we are allowed the same fact about different objects. That is, maiden name is a fact either about the employee or about the spouse. Of course, it is also true that, indirectly, they are both facts about an employee, being either her maiden name or his wife's maiden name.

2.2.2 Denormalization

Denormalization is a form of data redundancy which may be introduced to optimize performance [Schkolnick]. It can actually yield two different forms of redundancy, depending on whether the field involved is simply moved to another record, or maintained in both records:



- Moved:

  The department manager's name is moved from the department record to the employee record. A manager's name is now replicated with every employee record for that department.

- Maintained in both records:

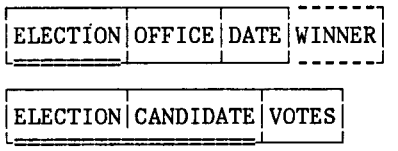
  In addition, the manager's name is also retained in the department record. Now it appears in multiple record types as well as in multiple record occurrences.

2.2.3 Derivable Information

Many kinds of derivable information are frequently included in the stored data, usually for reasons of performance, even though they are redundant. Sometimes computation time is avoided, sometimes access to other records is minimized.

Denormalization can be considered an example of this. In effect, the relationship "manager of employee" is being stored, even though it is derivable from "manager of department" and "department of employee".

It is not uncommon to store a sum (e.g., Order Total), even though it is computable from detail items. Similarly, one might include the winner's name in an election record, even though it could be derived from the votes obtained by each candidate:



This avoids the computation time involved, and also the need to access the records of all the candidates. Such a field might not be produced by a strictly logical approach to data design. Alternatively, if the user did specify such a relationship in his enterprise description, he might well neglect to mention the interdependence: the winner of the election had better be the candidate who got the most votes.

It should be noted that this kind of "derivability" relationship again illustrates the complexities of redundancy management, information inventory, and inquiry. A search for unplanned redundancy should take note of the fact that one data base identifies the winners of elections, while another records the number of votes received by each candidate. Inquiries about the winners of elections should be answerable from that latter database.

2.3 Field-Level Alternatives

Next we deal with design alternatives at the field level rather than the record level. The first case involves alternatives in a single field. The others involve multiple fields, possibly changing in number from one alternative to the other.

The general principle underlying these cases is that, for a given piece of information, the content and number of fields can vary depending on the form of representation chosen. This constitutes a family of design options, and also some impediments to detecting commonality of information.

2.3.1 Alternative Representations

This is a familiar and commonplace design choice: selecting among several possible identifiers or representations. Examples:

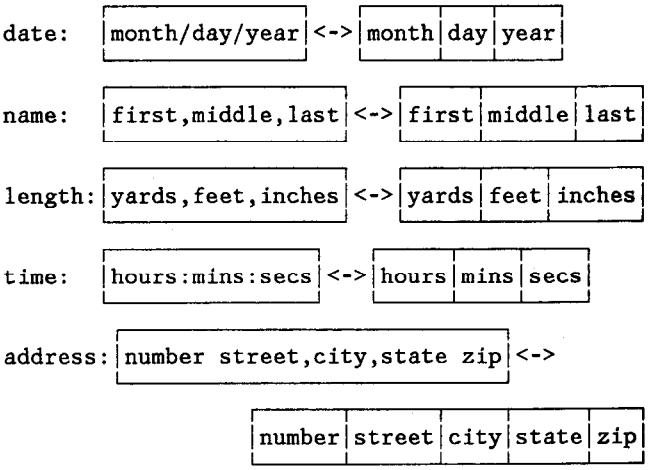
- Employee numbers vs. social security numbers vs. operator codes vs. people names.
- Part number in various formats, or vendor's codes for the parts.
- Various representations for time and date.
- For numeric quantities, the choices of units, data type, number base, precision, etc.

These are not of major concern for us as a data design issue. But it's worth noting some potential problems with respect to data documentation and other aspects of information control. Specifically, it's important to document both the entity represented and the form of representation as separate attributes of the data element. If the underlying entity types are not clearly indicated, then such things as redundancy or implied information can be difficult to detect.

For example, if one database is documented as containing part numbers and prices, and another is documented as containing vendor's codes and prices, then the redundancy may go undetected unless a human being happens to notice.

2.3.2 Concatenated Fields

In many cases, it's really a very arbitrary decision as to whether something should occupy one or several fields. Sometimes the relevant argument has to do with whether these things are perceived as one thing or many, and it sometimes has to do with whether there is any need to refer to the sub-components independently. Trying to put these on any theoretical foundation will probably divide the world into two equal camps (excluding those who don't care at all). Some examples:



Sometimes obviously distinct facts are concatenated for efficiency reasons, perhaps so that programs can refer to them with a single field name, perhaps to minimize the number of field names in a dictionary, or perhaps because the physical implementation of distinct fields would take up too much space. This leads to alternatives like:

EMP	SEX	MAR	SEE	CHG
Jones	M	S	1	0
Smith	F	D	0	0

<->

EMP	PD	AUTH
Jones	MS	01
Smith	FD	00

In which

PD (personal data) = sex + marital status.
AUTH (authorizations):
First digit: 1 = may see salary records.
Second digit: 1 = may change salaries.

2.3.3 Embedding

Facts may be combined by more elaborate embedding than simple concatenation. Sometimes one fact is embedded as a substring of another (frequently involving identifiers):

- Insurance claim number includes date of claim.
- Part serial number includes part type.
- Person number includes date of birth.

And sometimes the embedding is even more computational:

- The person numbers are even or odd based on sex.
- Winners and losers of elections could be distinguished by using positive and negative numbers for their vote counts.

The "Personal Data" of the preceding EMPLOYEE record might often be further encoded:

1 = male, single
2 = male, married
3 = female, single
4 = female, married

One could even take the view that derivable information implies a kind of embedding. One could say that the vote count field also identifies the winner (even without using signed numbers), simply because one can determine the winner by comparing votes. In effect, the count field contains two kinds of facts: the vote counts explicitly, and the winners and losers implicitly.

The alternative in all these cases is to provide distinct fields for all these facts. In many cases, this will lead to redundancy with the "known" contents of identifiers, unless the identifiers are also restructured to be purely random labels devoid of any information content.

2.3.4 Uniform vs. Mixed Representations

Representations of data have various attributes, such as length, data type, scale, or units of measure. In the vast majority of cases, these are uniform over a field, i.e., uniform for a given field in a record type. That's one of the foundations of the record concept.

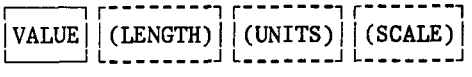
In this normal case, such attributes are factored out into the data description, e.g., into a data dictionary.

But there are exceptions. Sometimes mixed representations have to be used, and then the descriptions can no longer be factored out. The data must become self-describing. Additional fields must be introduced to describe the representations of other fields in the record.

The most common example is a length field which accompanies a field of varying length data. There are many other examples in which, for example, one field provides the units of measure for another. Time is sometimes handled in this way. The effective periods of contracts might vary considerably, and it might be desirable to store some in days, some in months, and some in years. This would require two fields: the quantity (e.g., 12), plus the units (e.g., weeks). (Of course, there is also the option to concatenate these into one field, e.g., "12 weeks".)

The alternatives in this general case are:

- Convert everything to a common representation (fixed length, same units, etc.) using a single field.
- Add fields to describe the variable representation.



2.3.5 Qualification vs. Unique Identifiers

For objects having no unique identifiers, assigning new unique identifiers is always a possibility.

Sometimes there is another option: identification via some unique attribute or related entity. For example, cities with the same name (in the United States) can be distinguished by the state in which they occur. These related things would serve as qualifiers.

The most direct effect is to increase the number of fields in a record. A given fact (e.g., people's birthplaces) could be expressed in a different

number of fields, depending on whether unique or qualified identification was used.

These options imply more than the simple addition or removal of an identifier field, or a change of representation within a field. They can give rise to substantial differences in field and record formats.

If cities had unique identifiers, then we might have a data base like:

PEOPLE

(data about people)

CITIES

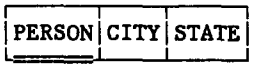
(data about cities)

But if cities are referenced by name, qualified by their states for uniqueness, then:

1. A state field has to be imported into the PEOPLE record (which would violate third normal form if city codes were retained).

2. The CITIES record becomes redundant, and would usually be discarded (assuming we aren't maintaining other information about cities). Users will obviously know not to ask what state a city is in; if they can uniquely name the city, they know the state. Application logic changes: applications needing to know in which state a person was born no longer need to access a second record.

The result is a data design with one less record type, and fewer data elements:



2.3.6 Nameless Entities

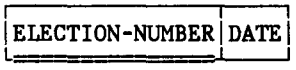
One thing worse than having no unique names is having no names at all. This can be treated as an extreme case of the qualifier situation. As before, there is always the option to assign arbitrary new unique identifiers.

But if there happens to be a related entity or attribute in one-to-one correspondence with these objects, then we can take advantage of them for unique indirect naming. Consider elections. They have no natural identification of their own, and we could assign them arbitrary election numbers. But we could also identify them by the year in which they occurred, e.g., "the 1980 election" (provided that we were only concerned with U.S. presidential elections). Similarly, it might be possible to identify sales territories by their headquarters city, e.g., the Chicago territory.

Strictly speaking, even social security numbers function in this manner. To be precise, a social security number identifies an account, although it is frequently used to identify the owner of that account.

In most respects the treatment is the same as for qualification, except that it only takes one field. In a sense, we are now also dealing with a form of embedding: one field is serving two purposes, both identifying an occurrence (an election) and telling us a fact about it (when it occurred).

We still have the "disappearing record type". With arbitrary election numbers, we would have distinct records for elections and for the performance of candidates in elections:



But if elections were to be identified by their dates, then the database design is reduced to:

As before, users will obviously know not to inquire about the date of an election; if they can name the election, they know the date.

Note that if presidents were uniquely named and were limited to a single term, then elections could also be identified by their winners: the Carter election, the Reagan election, etc. In that case it would again become meaningful to ask when an election occurred, but it would become foolish to ask who won it.

2.3.7 Multi-Type Domains

Even when unique identification is available for an entity type, there may be a need for uniqueness across the union of several types. This may arise whenever multiple entity types can occur in one role of a relationship, i.e., in one field of a record.

Examples of such situations:

- The owners of things (might be people, departments, companies, etc.).
- The things owned (use your imagination).
- Things to which users might have access, under an authorization scheme.
- Units of work might be delayed ("held") with the responsible holder being either a person or a department.
- An employee might belong either to a department or to a branch office.

There are two factors affecting the design choices in this case:

- Whether or not identifiers are unique over the union of these types. (We assume there are unique identifiers within each type.)
- Whether or not the identifiers of the different entity types are of the same data type.

Depending on these criteria, the following are some design possibilities (not mutually exclusive):

- Use a single field, with a data type that matches the data types of all the identifiers.
- Use a single field, with a "loose" data type that accommodates all the identifiers. In the worst case, this would be a varying character string.

  This approach loses some of the benefits of type checking.
- Use a different field for each entity type, each with the proper data type.

  All but one of these fields should be blank in a given record occurrence.
- Provide an additional field explicitly specifying the entity type occurring in each record.

  This could either be a required qualifier if identifiers are not unique, or simply an additional attribute.
- Create a new super-type and assign new global identifiers.

The following are some possible designs for what is essentially a simple binary relationship between company cars and their users, where the assigned user might be either a department or an employee:

CAR NUMBER	USER

CAR NUMBER	USER TYPE	USER

CAR NUMBER	USER TYPE	DEPT NAME	EMPLOYEE NUMBER

In the third design, the user will appear in either the third or the fourth field. One field will be blank in every record. But the design does take advantage of built-in data type checking for the identifiers.

We might note a curious symmetry. In an earlier section we discussed alternate meanings for a given field. Here we have a case of alternate fields for a given meaning.

3.0 IDEAS FOR A SYSTEMATIC METHODOLOGY

3.1 Overview

The main steps in a systematic design methodology will probably be:

1. Develop an initial data design from the input semantic model.
2. Apply some elementary transformations to generate possible alternative models.
3. Gather data for the parameters governing the choice between such alternatives.
4. Evaluate that data and choose a design.

In the present paper, we deal only with Step 2.

3.2 Elementary Record Transformations

Figure 6 shows a suggested set of elementary transformations (operations) by which it is possible to generate many alternatives (not including the field-level alternatives) from a given initial design. They may also be used in trying to determine whether two designs contain similar information, by trying to transform one into the other. Because we have tried to make them elementary, they may not all provide interesting design alternatives in themselves. Many of them are only significant when used in combinations, as will be illustrated later.

This is only a suggested list. We haven't taken the time to verify that all the examples can be expressed as combinations of these transforms.

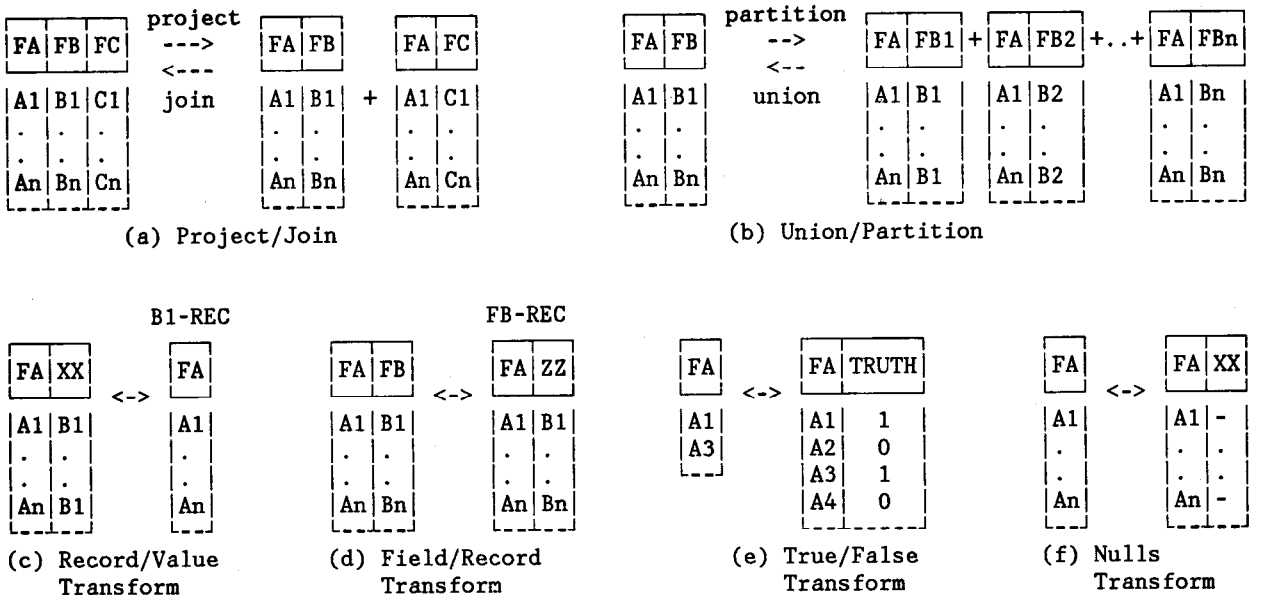


Figure 6. Elementary Record Transformations.

The transformations bear a strong resemblance to the operators of the relational model, particularly those defined in the extensions for RM/T [Codd]. Though the functions are similar, these transforms differ significantly in intent from the relational operators. The relational operators are designed to operate on relation instances during "execution" time, while the transforms are intended to operate on relation definitions, during the design process that precedes even the definition stage. Thus, for example, concepts like intermediate unnamed relations, or elimination of duplicates, apply to the relational operators but not to these design transforms.

In a sense, these transforms may be likened to view definitions in relational systems, the principal difference being that both the initial configurations and the final results are candidates for being considered as base tables.

Another important distinction is that these design transforms are not intended to provide precisely equivalent structures. Rather they are meant to generate plausible design alternatives. The degree of equivalence that may be achieved under various constraints, and the relative merits of the alternatives, are topics that remain to be explored systematically. In the extreme, these transforms may even be used to detect forms which are not equivalent, but where there are considerations of consistency to be maintained, i.e., implications.

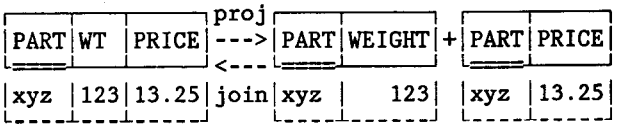
The notation is not rigorous. It is merely a shorthand way to suggest some basic concepts. We use the following terminology:

- "Record" = record type name = relation name.
- "Field" = field name = column name = attribute name.
- "Value" = field value = tuple element = domain element.
- "FA" is an example field name, and A1,...,An are possible values occurring in that field.

3.2.1 Project/Join

This analog of the familiar relational operators serves several purposes:

1. When the common field (FA) is a key, then we have a simple possibility of partitioning or merging records. Example:

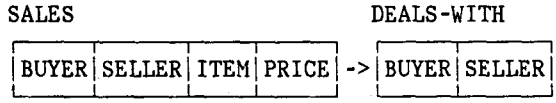


The joined form provides for a single "master list" or "existence list" of FA values (e.g., a master parts list), each occurring as a record key once and only once in the database. The function here is somewhat analogous to the "E-relations" of [Codd]. The joined form also suggests that every FA value has corresponding values of the FB and FC attributes, e.g., every part has a price and weight. Nulls (blanks) must be used when this does not hold.

The partitioned form avoids the need for nulls, but also loses the "master list" capability unless additional constraints are specified and enforced.

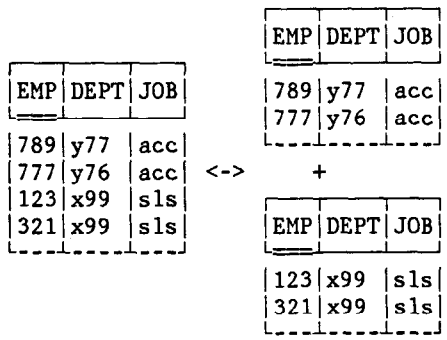
2. Normalization and denormalization can be done by sequences of joins and projects.

3. Projection provides a form of implication, by which a super-relationship can be derived from a sub-relationship:



It does not yield an equivalent alternative data design, but it must be taken into consideration when exploring for possible redundancies among databases.

3.2.2 Elementary Union/Partition



The partition creates n record types having constant values in the "FBi" fields (referring to Figure 6b). It is the same as the "partition by attribute" (PATT) operator in [Codd].

The partitioned form is rarely useful by itself, since it has a field of constant values in each record. The transformation would normally be used as an intermediate step, in conjunction with others.

The partition is only valid under the fixed population constraint. The partition must yield a fixed number of record types, and is therefore only acceptable when field FB (or its underlying domain, in relational terms) has a fixed and known population of permissible values. And, preferably, the population should be relatively small. Plausible examples might include fields representing the months of the year, the work shifts in a day, or possibly the sales territories of a company. When this transform is combined with others, it must be understood that the fixed population constraint still applies.

Going in the other direction, the union form is not truly equivalent to the partitioned form unless there are enforced constraints on the permissible values in FB. Otherwise, updates to the data could introduce values not corresponding to any of the data in the partitioned form. The trade-offs with respect to population control are these:

- The union form is preferable if the set of values is to be open ended, allowing freedom of growth to accept new kinds of information without having to restructure the data.

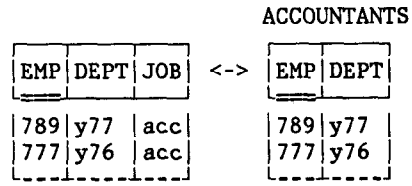
- The partitioned form is preferable if one desires to embed constraints in the structure, in order to exercise control over the allowable values.

There is another consideration on the union, regarding identifiers. The union yields a fully equivalent form only when there is uniqueness and data-type compatibility among the "FB" field values. Otherwise, additional transforms may be needed (see section 2.3.7, "Multi-Type Domains").

The union and partition forms differ in the enforceability of functional dependencies. In the union form, making FA a key insures that each value of FA can only occur once, i.e., with only one FB value. In the partitioned form, the same FA value can occur with a different FB value in each record type.

3.2.3 Record/Value Transform

A value ("B1" in Figure 6c) can be expressed either as a field value or in a record type name:



The record-to-value transform is a minor extension of the TAG operator in [Codd], where it is limited to preserving the record name in the field value.

3.2.4 Field/Record Transform

Field names and record type names can often be interchanged, especially when a record represents a single relationship (or attribute):

3.2.5 True/False Transform

This transform, in a sense, considers a list of all "possible" occurrences of something, and flags those which are true, or which exist.

Normally, we keep data by recording the true facts and omitting the false ones. But it is also possible to record all possible assertions, and then flag which are true and which are false. If the "things" we are considering are the occurrences of relationships, then this transform appears as:

FA	FB
A1	B1
A2	B2

<->

FA	FB	TRUTH
A1	B1	1
A2	B1	0
A1	B2	0
A2	B2	1

3.2.6 Nulls Transform

A field of nulls can be added to or deleted from any record type. We will use this in an even more general form: a new record can be created at any time consisting of an arbitrary field and a null field.

3.3 Compound Record Transformations

We also designate some useful combinations of the elementary transformations (Figure 7).

3.3.1 Generalized Record/Value Transform

This is a simple combination of the elementary union/partition and record/value transforms:

EMP	DEPT	JOB
789	y77	acc
777	y76	acc
123	x99	sls
321	x99	sls

<->

.(intermediate).

EMP	DEPT	JOB
789	y77	acc
777	y76	acc

<->

EMP	DEPT	JOB
123	x99	sls
321	x99	sls

ACCOUNTANTS

EMP	DEPT
789	y77
777	y76

SALESMEN

EMP	DEPT
123	x99
321	x99

B1-REC ... Bn-REC

FA	FB
A1	B1
.	.
An	Bn

<->

FA
A1
.
An

+ ... +

FA
A1
.
An

(a) Generalized Record/Value Transform

FA	FB
A1	B1
.	.
An	Bn

+

FC	FD
C1	D1
.	.
Cn	Dn

<->

FAC	FBD
A1	B1
.	.
An	Bn
C1	D1
.	.
Cn	Dn

(b) Generalized Union/Partition

FA	FB	FC
A1	B1	C1
.	.	.
An	Bn	Cn

<->

FA	B1.FC	B2.FC	...	Bn.FC
A1	C11	C21	...	Cn1
.	.	.	...	.
An	C1n	C2n	...	Cnn

(c) Field/Value Transform

Figure 7. Compound Record Transformations

3.3.2 Generalized Union/Partition

Obtained by repeated applications of the elementary union/partition transformation. "FAC" in Figure 7b is the union of the FA and FC field values. When FA and FC have the same population of values, a frequent special case, then FAC=FA. As noted earlier, field transformations may have to be applied to deal with non-uniqueness or type-incompatibility of identifiers in the union.

3.3.3 Field/Value Transform

Obtained by combining the field/record and record/value transforms. For example, the months can occur as field values or in field names:

YEAR	MO	SALES

<->

YEAR	JAN-SALES	..	DEC-SALES

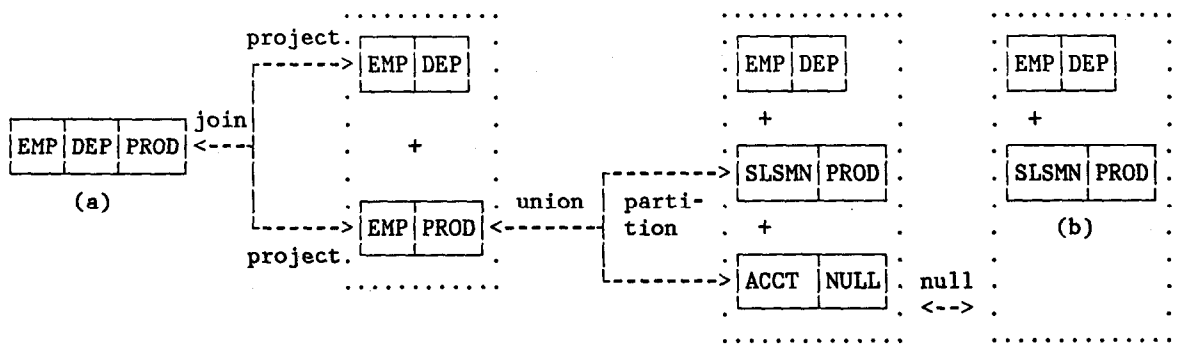
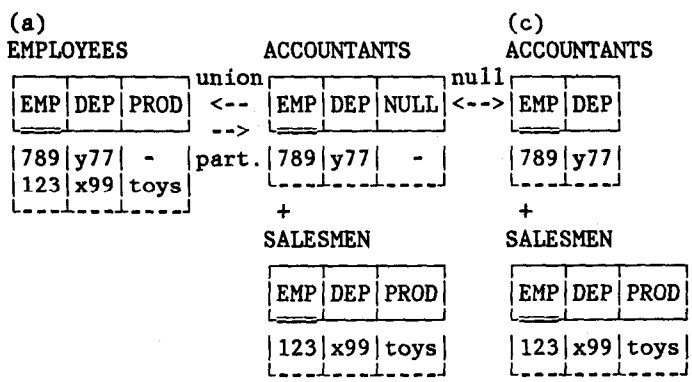


Figure 8.

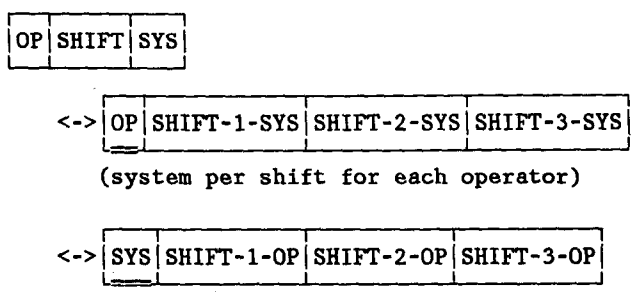
3.4 Sample Applications of the Transforms

Figure 8 shows the transformation between forms (a) and (b) of the subtype example (Figure 2), combining the null, union/partition, and project/join transforms. We omit the JOB field.

The transformation between forms (a) and (c) of the subtype example (Figure 2) also illustrates a combination of the null and union/partition transforms (we omit the JOB field):

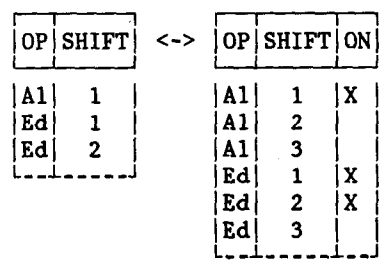


For an example of the field/value transform, consider a work schedule which assigns different operators to different systems on different shifts. (For the sake of this example, we assume that one operator is assigned to one system for one shift, and allow operators to work more than one shift.) This example can be transformed in at least two different ways:

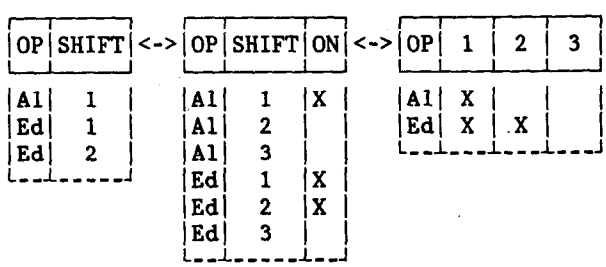


In the most general case, there are many ways in which this transform can be applied to a given record. Any one of the initial fields can be retained as a key. Multiple combinations of some remaining fields could be migrated into field names (by repeated applications of this transform), with the remaining fields left to provide field values.

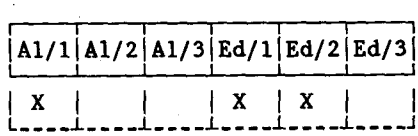
The true/false transform can be applied to a simplified work schedule, in which we simply track which operator works on which shift:



By itself, that is not usually an interesting option. But if we combine it with the field/value transform, we obtain a familiar schedule form:



In the extreme, we can reduce the entire schedule to a single record occurrence:

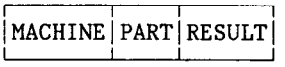


Observe the "fixed population" constraint: this design option is only viable with a fixed set of shifts and a fixed set of operators.

4.0 CHOICES IN SEMANTIC MODELLING

What we have said about data designs is also true for enterprise descriptions. Just as there are many possible data designs for representing a slice of reality, so also are there many ways to structure the corresponding enterprise description in terms of entities and relationships.

To illustrate the extensive range of possible enterprise descriptions, consider the production of certain parts by certain machines. The resulting parts might be finished, damaged, or lost. A reasonable data design is:



The prose description of this enterprise might be structured into entities and relationships in any of the forms shown in Figure 9.

4.1 Subtypes

Returning to an earlier point, the examples of Figure 2 can be analyzed in terms of the semantics of subtypes. The main problem with subtypes is that they require an "L-shaped" data structure (Figure 10), which is incompatible with the natural semantics of records [Kent 79]. There are some attributes which apply to all entities of a given type, and some which apply only to a subset of the entities, i.e., the subtype.

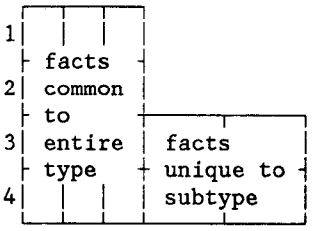


Figure 10. The subtype "L".

Unfortunately, record types are rectangular. Every instance of a given record type contains the same fields. So, to represent subtypes in records, we have to transform the L-shape into rectangles. This is where choices arise -- there are at least five ways to convert an "L" into rectangles, as illustrated in Figure 11.

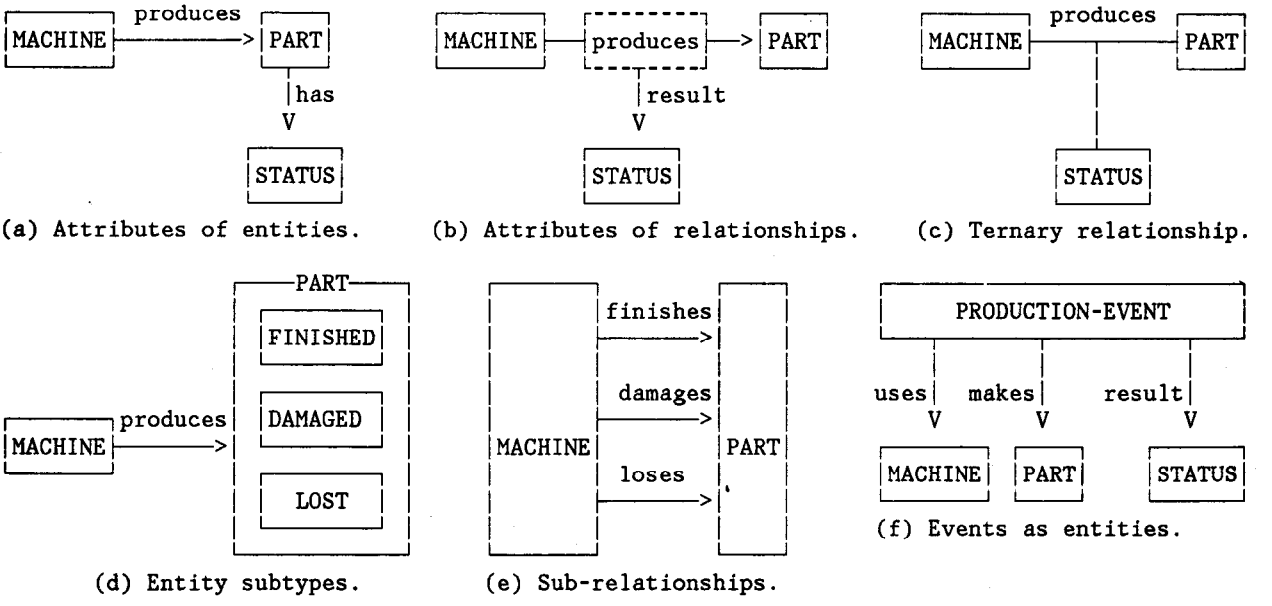
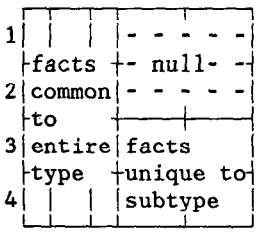
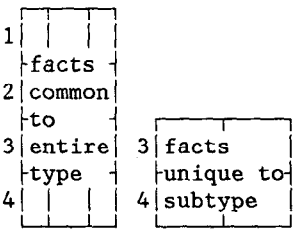


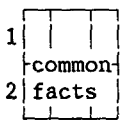
Figure 9. Alternative Semantic Models

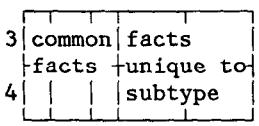


(a) Single record type, with null values.

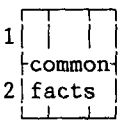


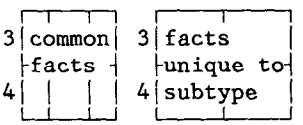
(b) Two record types, replicating entities.



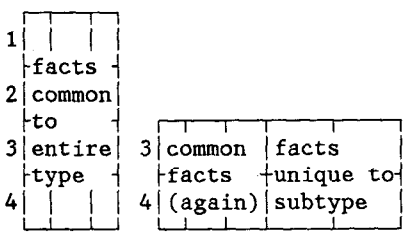


(c) Two record types, partitioning entity types.





(d) Three record types.



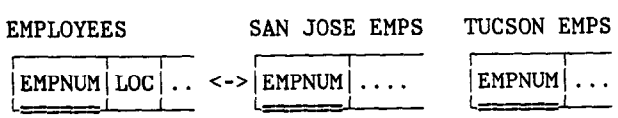
(e) Redundant information.

Figure 11. Five ways to make rectangles.

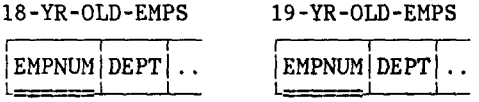
4.2 Type/Attribute Interchangability

The concepts of type and attribute can be interchanged. A fact can be treated as a basis for classification into groups, or simply as a property. We have treated "salesman" and "accountant" as entity types, reflected in distinct record types, and also as properties of employees, reflected in values of a job field.

Conversely, any attribute -- or combination of attributes -- could become the defining characteristic for some types. When mapped in the simple and direct way to data designs, this is reflected in possible migrations from field values to record types. For example, the locations of employees might normally be maintained as a field value in an employee record. But, alternatively, the data could be partitioned into a subtype form -- one record type per location -- with the location now implied by the record type name rather than being explicit anywhere in the data:



Carrying this case to its extreme, numeric information could be expressed in subtype fashion, with a distinct record type for each value. Thus we could conceivably contemplate a record type for 18-year old employees, one for 19-year olds, etc.:



Such alternatives make sense only to the extent that the value populations are fixed, i.e., a fixed number of locations or employee ages.

4.3 Types, Attributes, Relationships

Another observation is that types (and hence attributes) can often be interchanged with relationships. An employee is something which is employed by a company. A salesman is a person (or an employee) who sells for some company. Conversely, a person who runs in an election is a candidate; one who wins an election is a winner.

The parallel between relationships and types suggests that we ought to consider sub-relationships as an analog to subtypes. Just as we can characterize people with greater or less precision as being either salesmen or employees, so also we can sometimes choose among relationships which are more comprehensive or less. For example, we might perceive these as relationships:

- Which employee designed which product.
- Which employee assembled which product.
- Which employee tested which product.

Or we might perceive the more comprehensive relationship:

- Which employee worked on which product.

To preserve the same amount of information, we might then treat "designed", "assembled", and "tested" as attributes of the "worked on" relationship, just as a "job" field provides more information about the subtype of an employee.

For another example, we might perceive the relationships

- Who won which election.
- Who lost which election.

Or we might perceive the relationship

- Who ran in which election,

with winning and losing being reflected in attributes.

Just as salesmen and engineers are different kinds of employees, we could say that designing, assembling, and testing are different kinds of "working" on products -- and also that winning and losing are different kinds of "running" in elections.

Winning and losing are sub-relationships of "ran in", and winners and losers are subtypes of candidates. The same can be said about designing and designers, assembling and assemblers, and testing and testers.

4.4 Two Levels of Options

It is beyond the scope of this paper to pursue equivalence transformations of enterprise descriptions. [Falkenberg] discusses some equivalence transformations for relationships. He suggests an approach to selecting a preferred or canonical form out of each set of equivalents, which would be an important contribution toward dealing with some of these problems.

This variability on two levels leaves us with the situation of Figure 12, all of which applies to the same piece of reality. There is a set of equivalent enterprise descriptions, and also a set of equivalent data designs. There are some especially direct correspondences between certain enterprise descriptions and certain data designs, represented here by the vertical lines. These are the basis of the simplistic design algorithms of today's methodologies, in which a given enterprise description forces a particular data design.

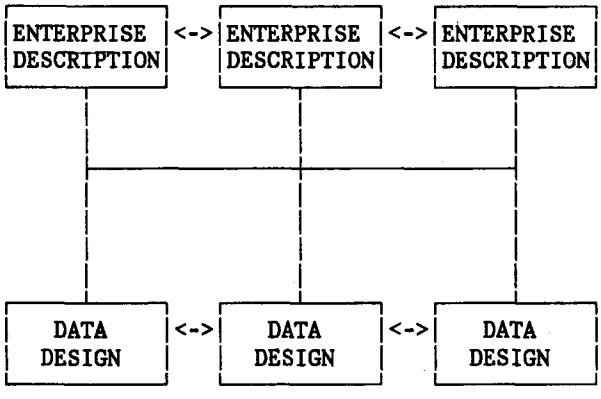


Figure 12.

We are proposing that a richer set of data design options be made available for a given enterprise description, either by directly transforming the data designs or by transforming the enterprise descriptions and then obtaining the corresponding data designs.

5.0 CONCLUSIONS

We have identified a wide range of data designs, and also enterprise descriptions, that can correspond to a given slice of reality. We have tried to point out the consequences for data design methodologies and tools, and also for data documentation techniques. And we have tried to suggest the beginnings of a systematic methodology for dealing with such variability.

The work is preliminary, and leaves many questions open. We may not have discovered all the design options. We may not have classified them in the best way. The transforms we suggested might be neither necessary nor sufficient nor optimal. And there is certainly considerable work left in the development of tools to help select optimal designs from the many available options.

ACKNOWLEDGMENTS

Several people provided valuable comments on earlier drafts of this paper, including Bob Bascom, Paula Newman, Goran Sandberg, and John Widger.

REFERENCES

[Codd] E.F. Codd, "Extending the Data Base Relational Model to Capture More Meaning", ACM Transactions on Database Systems 4(4), Dec. 1979.

[Falkenberg] E. Falkenberg, "Balanced Information Structures: The Basis for Well-Formed Conceptual Schemas", in preparation.

[Kent 78] W. Kent, Data and Reality, North Holland, 1978.

[Kent 79] W. Kent, "Limitations of Record Based Information Models", ACM Transactions on Database Systems 4(1), March 1979.

[Kent 81a] W. Kent, "A Simple Guide to Five Normal Forms in Relational Database Theory", IBM Technical Report TR03.159, Aug. 1981.

[Kent 81b] W. Kent, "Data Model Theory Meets a Practical Application", Proc. VLDB7, 1981.

[Kent 82] W. Kent, "The Meanings of Data Elements", in preparation.

[Schkolnick] M. Schkolnick and P. Sorenson, "The Effects of Denormalization on Database Performance", IBM Research Report RJ3082, March 1981.