By focusing on the facts to be maintained in a database, we obtain a methodology for data analysis and design which is at once simpler and more powerful than other methodologies.

1. INTRODUCTION

The methodology for data analysis is made simple by:

- Shifting our focus from entities to facts about entities, i.e., relationships.
- Using a simplified form of entity-relationship model.
- Deferring problems of identification and representation until late in the process.

In addition, we provide extended capabilities to deal with difficult representation situations, and to examine alternative data designs.

We focus on the middle portion of the analysis and design process: translating entities and relationships into record designs. This paper does not describe procedures for identifying the entities and relationships, nor does it deal with physical data design.

We go a little beyond what might ordinarily be considered “logical data design.” While this methodology can produce records in third or even fifth normal forms [11], our aim is to produce the actual record designs that will be implemented in a database, even if they might be inelegant or redundant in some ways. What we do not deal with are the other aspects of resource and access path management, such as storage mappings and index schemes.

We deal only with flat files. The methodology applies to relational databases, to other “flat file” data management systems, and to the design of the segment types in a hierarchical database or the record types in a network database. Nonflat data models (e.g., hierarchies and networks) will require additional design steps to replace some fields with structural paths (pointers).

The methodology has not yet been developed to the point of a detailed procedure. This paper describes the concepts on which the methodology is based, and the major steps in the methodology.

The first step is to identify the facts that are to be maintained, in terms of entities and relationships. Thereafter, the methodology essentially

1. Collects all single-valued facts about the same thing into one record.
2. Provides a separate record for each many-to-many relationship, including all single-valued facts about the relationship.

These two results are accomplished together in the following steps:

1. Generate a “pseudo-record” for each fact.
2. Identify “pseudo-keys” for each pseudo-record.
3. Merge pseudo-records having compatible keys.

Other steps in the methodology include:

1. Assign representations.
2. Consider alternative designs.
3. Name the fields and the record types.

While the approach is fundamentally simpler than others, it sometimes appears more elaborate because it deals with more complex and realistic situations, including:

- Imperfect identifiers, including multifield and embedded representations.
- N-ary relationships.
- Multivalued attributes, attributes of relationships, and attributes of attributes.
- Entity subtypes.
Partial relationships (i.e., optional, not required for all occurrences of an entity type).

Variations in record format to accommodate data processing requirements, e.g., denormalization.

In the following sections, we will:

1. Begin with a simple example.
2. Concisely describe the steps of the methodology.
3. Develop some background concepts.
4. Explain the methodology in greater detail.

1.1. Introductory Example

This simple example illustrates the essential framework of the methodology. Many refinements are not taken into account here.

1. Specify the facts to be maintained. The facts:

Employees have employee numbers (1:1).
Employees have names (n:1).
Departments have department numbers (1:1).
Departments have names (n:1).
Employees are assigned to departments (n:1).
Employees earn money (n:1).
Employees manage departments (1:1).

We present the same facts in diagram form.

Such diagrams of the facts lead almost mechanically into the next step.

2. Generate a “pseudo-record” for each fact

Representations will be dealt with in a general way at a later step. But when some of the facts serve to assign obvious unique identifiers to things, then we can take a shortcut at this point by replacing such things with their identifiers. We do this now for employees and departments.

3. Identify “pseudo-keys” for each pseudo-record. The concept: if we merely assume that each of these “fields” actually contains a unique identifier, then what would be the keys in each pseudo-record?

We use “- - - -” to indicate the pseudo-keys.

Some pseudo-records may have several such keys, when one-to-one relationships are involved. Here, we see that each department (14) is managed by (at most) one employee (13), and each employee manages (at most) one department.

4. Merge pseudo-records having compatible keys. For this example, key fields involving the same entity types are compatible. Merging all the pseudo-records having EMP# as the key yields:

Merging all the pseudo-records having DEPT# as the key yields:
Note that the "manages" pseudo-record (13-14) can be merged in two ways, based on either of its keys. For the purpose of this example, we choose to keep the fact in the department record. The resulting department record could be keyed either by department or by manager, since managers are in one-to-one correspondence with their departments.

5. Assign representations. In the general case, some restructuring could occur at this point to deal with representation problems. In the present simple example, however, we observe that all fields except MONEY (12) already contain character strings that can serve as representations in the data. All that remains here is to specify a representation for field (12). We assume here that it will be expressed in dollars, in integers.

We show a number of alternative record designs which could be considered for representing the same set of facts:

1. The current design:

   EMPNUM | ENAME | DEPT | SALARY | DEPTNUM | ENAME | MGR

2. Subdivide the employee name field:

   EMPNUM | FIRST | MID | LAST | DEPT | SALARY | DEPTNUM | ENAME | MGR

3. Keep the "manages" fact in the employee record, i.e., identify the department which the employee manages (possibly none):

   EMPNUM | ENAME | DEPT | SALARY | MGS-DEPT | DEPTNUM | ENAME

   MGS-DEPT contains a department number or a blank. This design alternative is only valid if all departments have managers, or null values are permitted in the EMPNUM key field.

4. Put all the department information into the record of the department's manager, eliminating the separate department record (subject to the same restriction):

   EMPNUM | ENAME | DEPT | SALARY | MGS-DEPT | DEPTNUM | ENAME

5. If we have the additional constraint that a manager belongs to the department he manages, then the "manages" fact could be kept by a simple flag in the employee record:

   EMPNUM | ENAME | DEPT | SALARY | MGR-FLAG | DEPTNUM | ENAME

   MGR-FLAG is a simple binary flag indicating whether the employee manages his department.

6. We could create a separate record type for employees of each department, eliminating the need for a DEPT field:

   D01 EMPLOYEES: EMPNUM | ENAME | SALARY | DEPTNUM | ENAME | MGR

   D02 EMPLOYEES: EMPNUM | ENAME | SALARY

   Some of these alternatives can also be combined, e.g., maintaining the "manages" fact in both the department record and the employee record.

6. Name the fields and the record types.

   "EMPLOYEE RECORD":

   EMPNUM | ENAME | DEPT | SALARY

   Representations: EMP# | NAME | DEPT# | INGR-DBR

   "DEPARTMENT RECORD":

   DEPTNUM | ENAME | MGR

   Representations: DEPT# | NAME | EMP#
1.2 A Skeleton of the Methodology

Subsequent descriptions of the methodology will be interspersed with definitions and explanations of background concepts, perhaps making it appear more elaborate than it really is. We present here a complete skeleton of the methodology, without benefit of definitions or explanations, to keep its overall extent in perspective.

1. Specify the facts to be maintained, in terms of relationships among entities.
   A. For each fact, specify:
      1. A fact identifier (arbitrary).
      2. The relationship name, or sequence of composed relationships.
      3. The entity type occurring in each role, if more restrictive than that defined for the relationship.
      4. Least and most participations, if more restrictive than in the relationship definitions.
   B. For each relationship, specify:
      1. Relationship name.
      2. Degree (number of roles).
      3. Role names.
      4. The entity types permitted in each role.
      5. Least and most participation for each role, and also for selected sequences of roles.
   C. For each entity type, specify:
      1. Entity type name.
      2. Whether it is a subtype of another entity type.
      3. Whether it is a symbol type and, if so, whether it is structured.

2. Generate a "pseudo-record" for each fact. Representations may optionally be assigned at this point.

3. Identify "pseudo-keys" for each pseudo-record.

4. Optionally merge pseudo-records having compatible keys.
   A. Merge on compatible keys having the same population.
   B. Merge on compatible keys having different populations if null values are acceptable in other fields.
   C. Merge on subtypes of a common supertype.

5. Assign representations.
   A. For each field in a pseudo-record:
      1. If the entity type in the field is itself a structured symbol type, replace by multiple fields, and exit.
      2. If the entity type is a simple symbol type, leave it alone, and exit.
      3. If the entity type needs no representation, delete the field, and exit.

4. If the representation is derivable from another field (including indirect representation), delete this field, and exit.

5. Find a set of facts whose combination provides a representation having the required qualities of completeness, uniqueness, and/or singularity (as well as other qualities). Introduce new facts if needed.

6. If several suitable sets of facts are available, choose one.

7. "Import" such facts into the record, if not already there.

8. Delete the field.

9. Repeat the assignment of representations for any imported facts.

B. Compress fields, if desired.

C. Remerge if needed to eliminate redundancies.

(6) Consider alternative designs.

(7) Name the fields and the record types.

2. THE UNDERLYING IDEAS

2.1 Facts

We have used the term "fact" to mean something like "employees are assigned to departments." Strictly speaking, we should use a term like "fact type" or "fact pattern" here, and reserve the term "fact" for a specific item of information, such as "Joe Smith is assigned to the Sales department." However, wherever the intent is clear, we will continue to use the term "fact" in the broader sense.

A data record usually contains a number of facts, such as an employee's department, salary, etc. In our approach to data analysis, we first identify the facts to be maintained, then aggregate them into data records.

Most methodologies suggest that data represents two kinds of facts: facts about things (attributes) and facts that connect things (relationships). Facts about an employee might include his salary, department, birthdate, and birthplace. Connecting facts might connect an employee to a project, or to his/her spouse.

If you're paying attention, you should be confused at this point. How did we decide that an employee's department was a fact about the employee, rather than a fact connecting two things? Why did we also decide that about his birthplace? I don't know the answer to that, and I don't think anyone else can give you a really effective answer either. It doesn't have to matter.

Facts connect things. We can avoid the question altogether by treating all facts as connections between things. This may seem surprising at first, and it does
take a bit of getting used to. But it simplifies things. Why shouldn't we think of some money as a thing, with the salary fact connecting that thing to an employee? We can just as easily think of dates or distances as things; then birthdays and heights become facts connecting such things to employees. Later on we shall see that identifiers can also be treated in the same way.

We seem to have the idea in mind already, in some of the ways we talk about things. When we say that "6 feet" and "72 inches" mean "the same thing," what do we mean? It isn't the character strings that are the same. Rather, there is some other thing that those character strings both stand for. The thing happens to be a distance, and the "height" fact connects that thing to some people.

That's what we focus on in this methodology: facts that connect things together.

How many things? Not all facts involve pairs of things. Facts can connect any number of things simultaneously. Consider the fact that John bought a certain computer from Sears. That involves three things: John, the computer, and Sears. This fact can't be replaced by the two simpler facts that John bought a computer and John shops at Sears. He may have bought the computer elsewhere, and he may buy other things at Sears. Thus this fact has to be treated as a unit involving three things. In general, facts can involve any number of things.

Why they're connected. We usually have to say what kind of connection the fact represents. It's not enough to say that a fact connects an employee to a department, or to a date, or to a distance. We have to say whether he's assigned to or loaned to or manages the department, whether the date is his birthdate or hire date, and whether the distance is his height or something else.

In ordinary language, we don't always have to be explicit about these things. Sometimes we can guess correctly: an employee's department almost always means the department to which he is assigned. Sometimes we use a word that tells us two things: "birthday" says the thing is a day and the connection is "born on"; "height" tells us that it is a distance and also how it relates to a person; "salary" tells us that it is money and that it is earned. Sometimes we can't even find two different words for the two ideas: "color" is the name of an entity type (kind of thing), and it is also the name of its relationship to colored things (at best, we could call the relationship "color-of", or "has-color"). But in general we need to know at least these things about a fact: what it's connecting (employee and money) and why they're connected (it's his salary).

The roles they play. Sometimes we have to clarify a third thing about a fact: what role each thing plays in the fact. This is particularly necessary when the fact connects several things of the same kind. We may have two employees connected by a "manages" relationship, or two parts connected by a "contains" relationship, but that's not enough information. We have to know who manages who, or which part contains which. One way to do this is to assign a sense of direction to the connection. We can say that the "manages" relationship goes from the manager to the employee, and the "contains" relationship goes from the assembly to the component. However, direction is difficult to deal with when the fact involves more than two things. When John buys a computer from Sears, which is the "from" and which is the "to"?

A more general approach is to use role names, like "manager" and "managee," or "component" and "assembly," or "buyer," "seller," and "commodity." The use of role names actually covers directed relationships as well, since we can always use "from" and "to" as role names. As before, we don't always feel the need to be explicit about roles. Very often, we can use the entity type names as default role names: in an "assigned" relationship, there is no doubt about the roles played by the employee and the department.

Example. Let's put this all together into an example that illustrates various aspects of a fact. In today's complex economic world, some companies own other companies, and they might sell such companies to each other. If we wanted to keep track of which sold which to which, we might need this form of the "sale" relationship:

```
relationship: SALE
roles: BUYER SELLER COMMODITY
entity types: COMPANY COMPANY COMPANY COMPANY
```
One additional aspect of facts will be covered later: how things are represented. We haven’t said anything in this example as to how the companies are to be identified. It might be by name, by stock exchange symbol, by some taxpayer code, by some other conventional abbreviations, etc.

2.2 Entities and Relationships

We have identified the following things to say about a fact:

1. The kinds of things it connects.
2. The nature of the connection.
3. The role each thing plays in the connection.
4. How the things are represented.

**Entities.** The “kind” of thing involved in a fact is what we mean by the term “entity type.”

This term gets used in several ways, which can be confusing. It is sometimes expected that employees and departments are entities, but money and colors and dates are not. We don’t make that distinction. It gets to be too troublesome in many cases. Children and birthplaces, for example, are often not considered entities, but we’re hard pressed to explain why.

We treat all these things as entities. Any person, place, thing, or idea is an entity. Anything to which a noun or noun phrase can refer is an entity. That simplifies things for us. We don’t have to worry about which are and which aren’t entities. In terms of data, it is convenient to say that all data fields refer to entities. We will later explore why other design approaches want to distinguish between entities and nonentities.

**Instances and types.** The term entity, as we use it, refers to an individual instance, or occurrence. You and I are entities, being occurrences of the entity type “person”.

In general, database design and description deal with types, e.g., “employee” and “department,” while the database itself mentions the individual occurrences, namely the individual employees and departments.

As with most such generalizations, there are exceptions. Sometimes individuals are mentioned in the database description, when constraints specify the allowed values for a field, such as the set of valid colors, or the valid sexes. Furthermore, as we shall mention later, when the population happens to consist of a single individual, it may not be mentioned anywhere at all.

Conversely, we sometimes name types in the database. Most of the time, e.g., when we talk about “parts” we really mean part types. “Part number” generally identifies a part type, whereas it is “serial number” that really identifies individual parts. In an inventory record, the PART field doesn’t tell us which part we’re counting (one part is one part, after all), it tells us the type of part we are counting.

**Subtypes.** Later on, when we discuss the merging of pseudo-records, we will be interested in entity subtypes. For example, we might be interested in merging data about salesmen with general records about employees. Therefore it will be useful to specify which entity types are subsets of which others. One entity type is a subtype (or subset) of another entity type whenever all members of the first are always members of the second.

One important aspect of subtypes is the inheritance of properties. Any fact involving a type automatically applies to a subtype. For example, if employees have employee numbers and departments, then salesmen have employee numbers and departments.

Types can overlap in a more general way, with neither being a subset of the other. Employees and customers are two such types. We will not need this generalized overlap concept in our design methodology.

**Relationships.** The kind of connection that binds entities together in a fact is what is meant by a “relationship type.”

A relationship type has a name and a fixed number of things that it can connect together. That number is called its “degree.” A relationship between two things has degree 2, and is called a “binary” relationship. A relationship among three things has degree 3, and is called a “ternary” relationship. In general, a relationship connecting \( n \) things has degree \( n \); however, the term “\( n \)-ary” is often reserved for relationships of degree greater than 2.

When a directional convention is used for binary relationships, a distinct name is sometimes given to each direction, e.g., “owns” and “owned by.” While this may be useful at an end user interface, it is not essential to this methodology, and we do not adopt that convention here.

**Participation.** To determine pseudo-keys, it is necessary to know whether a relationship is one-to-one, one-to-many, or many-to-many. Furthermore, we may have to know this about combinations of roles in \( n \)-ary relationships. Consider an inventory relationship, involving part types, warehouses, and quantities on hand (QOH). Though each part type and each warehouse are likely to occur several times, the combination of a given part type and a given warehouse should occur only once.

We will use a notion of “participation” to express such characteristics.
For each role in a relationship, we will specify whether an entity can play that role in more than one occurrence of the relationship. For example, an employee can occur in at most one occurrence of the relationship of being assigned to a department. We will also identify sets of roles in which such participation is limited to one. For example, the combination of a part type and a warehouse may occur no more than once in the inventory relationship.

Since we are mainly interested in whether the maximum participation is one or many, we will say that a role, or set of roles, has a maximum participation (MP) of 1 or n. 1 would signify a single-valued relationship, n a multivalued one. For example, for employees assigned to departments, MP for employees is 1 (each employee can have at most one such assignment), while MP for departments is n (a department could have many such assignments.)

In a similar way, when we plan to merge records, we will have to know whether all entities of a given type will necessarily participate in a relationship. That is, we have to know whether the relationship involves all or only some of the entities.

We will specify whether or not an eligible entity has to participate at least once by saying that a role, or set of roles, has a least participation (LP) of 0 or 1. 0 would signify an optional relationship, 1 a required one. For example, for employees assigned to departments, LP for employees is 1 (each employee must have at least one such assignment), while LP for departments is 0 (a department might have no such assignments).

We combine the two concepts into a compact notation of the form LP*MP specified for each role in a relationship. In addition, it is useful to specify participations for sets of roles when the maximum participation is 1 for the set but not for all of the roles in the set.

The possible values of LP*MP are:

- 0*1 ("at most one")
- 1*1 ("exactly one")
- 0*n ("some", "many")
- 1*n ("at least one")

We illustrate the notation:

```
<table>
<thead>
<tr>
<th>EMP</th>
<th>DEPT</th>
<th>PART-TYPE</th>
<th>WAREHOUSE</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1*1</td>
<td>0*n</td>
<td>0*n</td>
<td>0*1</td>
<td></td>
</tr>
</tbody>
</table>
```

Exercises: what would the participations be if each type of part had to be stored in some warehouse? In every warehouse? What if each warehouse had to stock some type of part? All types?

Unary relationships. Although we have been treating facts as connections or relationships among things, there is a limited sense in which it might be useful to think of facts involving single things as well. This would be like a simple list of things which are of interest for some reason. We might have the list of all employees, or the list of all products, etc.

For this reason, it also makes sense for us to allow "unary" relationships, i.e., relationships of degree one. Thus, by analogy with the binary relationship "assigned(Smith, Sales)," we could have the unary relationship "employee(Smith)."

We have introduced this principally to permit the unary relationship "exists," which we will discuss later. We suspect that other unaries may also prove to be useful.

Attributes of relationships (facts about facts). So-called "attributes of relationships," such as the dates when people started working for companies, can sometimes be handled conveniently as n-ary relationships (in this case, a relationship among three things). But it is necessary to recognize the case when we have several attributes of the same relationship (e.g., if we also had the dates when people stopped working for companies), so that we can merge them into the same record.

For that purpose we can treat relationships and facts as entities themselves, able to participate in other relationships and facts. As a result, we can have "facts about facts," such as:

Fact 1: people have worked for companies.
Fact 2: fact 1 began on date.
Fact 3: fact 1 ended on date.

The distinction between facts and relationships. In most cases, facts and relationships are the same, as we have been treating them until now. However, it is convenient to let facts sometimes take on a richer structure, using relationships as a basic building block.

In the more general case, we can let facts be compositions or restrictions of relationships.

In composition, a fact is composed of several relationships. This is useful when the underlying relationships are known in the semantic model of the enterprise, but perhaps not all of the supporting details are to be maintained in the database. For example, we may have the relationships "employees have secretaries" and "secretaries have phone numbers," but the only facts we want to store in the database are "employees can be reached at phone numbers," i.e., "X is the phone number of the (unspecified) secretary of Y."

Restrictions are useful when the same relationship is to be maintained separately for different subsets of the
possible entity types involved. For example, we might have “owns” as a relationship, but the facts “employees own stock” and “departments own projects” might be maintained separately. Or, we might want to separately maintain the facts “domestic employees are assigned to departments” and “foreign employees are assigned to departments.” Or we might wish to use the “sales” relationship twice, once for a company’s sales records and once for its purchasing records.

Facts and relationships are much the same thing, especially if we extend the concept of “relationship” to include derivable relationships, defined in terms of others. However, we tend to use “relationship” for the primitive associations, and “fact” for the more general notion of any association, primitive or derived.

We can thus describe a fact as a named composition or restriction of relationships among entity types (which may in turn themselves be facts). Facts have degrees and roles, derived from their constituent relationships.

2.3 Representation

We turn now to the fourth aspect of a fact: how things are represented. As we shall see, representation of entities can turn out to be a complex and difficult topic. One of the advantages of our methodology is that we do not allow this complexity to overshadow the whole design process. By deferring this to the end, we can first design for the facts we want maintained in the data, and then work out questions of representation.

Most methodologies assume that a simple representation is available for all entities. What we deal with in this section goes beyond the capabilities of other methodologies. The reader who wishes to deal with the simple case first, or who wishes simply to compare this with other methodologies, is encouraged to skip this entire section on first reading.

Why represent? So far, we have really been discussing facts in the real world, not how they are expressed in data. When we said that an employee is assigned to a department, we had in mind a real flesh-and-blood employee, and a real working department. We didn’t say that a certain person’s name was assigned to a certain department name, nor did we say that a certain employee number was assigned to a certain department number.

Unfortunately, we can’t put people and departments into the machine. We have to let symbols—typically, character strings—serve as surrogates. To represent a fact which connects two things, we create some conjunction of surrogates for those two things, generally by putting the symbols into the same record. To do that we exploit certain kinds of facts, namely those which relate things to symbols. When we put an employee number and a department number into the same record, representing the employee’s assignment to that department, we are really dealing with three facts:

1. Some employee has that employee number.
2. Some department has that department number.
3. The represented employee is assigned to the represented department.

How represent? An ideal representation would provide every entity with a uniquely corresponding symbol (character string) to serve as its surrogate.

We use “symbol” and “representation” in a very general way to mean any character string, particularly as it is used to stand for some other thing. The terms include the general ideas of naming, identifying, denoting, etc. “Bill Kent” is one of the symbols that can represent me, as well as several other people. “6,” “6 feet,” “72,” and “72 inches,” are some of the symbols that can represent my height.

We know that idealized representation doesn’t exist in a universal sense. Most entities don’t have a single simple symbol that stands exclusively for them and nothing else in the whole world. For most things, we have to resort to some sort of descriptive phrase, such as “the IBM employee with employee number 999999.” The best we can provide is that for some types of entities, there are some types of symbols which can serve as surrogates. Under these conditions, a symbol serves as a surrogate for an entity—provided that we know what symbol type it belongs to, and what type of entity it is representing. Thus, “999999” represents a certain entity—if we know whether the symbol is a social security number or an IBM employee number or an IBM part number or perhaps something else.

Representations fit into our general structure of facts. “Employees are represented by employee numbers,” “people have names,” and “heights are represented in inches with picture 99.9” are facts. Corresponding to this, we acknowledge that symbols are entities, and they are collected into entity types which we will also call symbol types. Employee numbers and people’s names are two such symbol types. We can also think of all symbols collectively as one large type. Then, in terms of subtypes, we can say:

“Employee number” is a subtype of “symbol,” and “Symbol” is a subtype of “entity.”

In other words, employee numbers are symbols, and symbols are entities.
Symbol types. A precise treatment of symbol types quickly becomes complex, and too esoteric for our purposes. We take a pragmatic approach, adequate for a practical data design methodology.

As a first approximation, we think of a symbol type as the set of character strings permitted to occur in a data field. (It is comparable to the "domain" concept in relational database theory.) Any characteristic which determines which actual character string can represent a given entity in a given field will also serve to distinguish different symbol types. For example, a given data field will probably accept exactly one of the following as a representation of my height: "6," "06," "6," "6.0," "6.00," "6 feet," "110," "VI," "72," etc. Each of those belongs to a different symbol type. That is, there are at the very least nine different symbol types that can represent lengths (the reader can surely identify a great many more). All of the following properties might be involved in the description of a symbol type, and might serve to distinguish one symbol type from another:

- Identifier type (e.g., employee number vs. social security number), or unit of measure.
- Number system (decimal, octal, binary, hex).
- Scale and precision.
- String length.
- Other editing characteristics, often captured in a "picture" clause, including such things as embedded decimal points, leading dollar signs, padding characters, etc.

In an elegant theory, each of those properties could be expressed as a distinct "fact" about a symbol type. For the purposes of our methodology, we will simply let those be part of the informal description of distinctly named symbol types.

Different sets of character strings necessarily constitute different symbol types. But we don't force the converse. That is, different symbol types might contain the same character strings. For example, employee numbers and part numbers might both consist of six-digit integers. It is acceptable to either treat this as one symbol type ("six digit integers"), or as two symbol types ("employee numbers" and "part numbers").

Some symbol types are "self-describing," instead of having all of their properties fixed and specified. One symbol type might include both "6 feet" and "72 inches." It is "self-describing" because the unit of measure is included with each item, rather than being specified for the entire set.

Symbol facts. We will refer to a fact in which one of the entity types is a symbol type as a "symbol fact."

Not all symbol facts are used for representation. Some of them simply provide information, such as maiden names of employees, or a table of abbreviations (connecting two symbol types).

Others may be said to "represent," even if they are not unique. In a certain sense, people are represented by their names.

Other representation techniques. Not all representation is done by simple symbols serving as surrogates.

Structured symbols. Some representations are structured in a form that requires multiple fields to represent a single entity. Examples:

- Dates (month, day, year).
- Addresses (number, street, city, etc.).
- Names (first, middle, last).
- Mixed-unit measurements (feet, inches).

Quite often, it is an arbitrary design decision whether such things are represented in single or multiple fields. It has little to do with the nature of the facts being maintained. Addresses may get split into multiple fields to facilitate formatting for letters and labels.

It is sometimes argued that in such situations we ought to describe components of things, or relationships among things (e.g., a date is a "relationship" between a month, a day, and a year), but that can be quite artificial. For the purposes of this design methodology, we bypass such considerations, and simply allow a symbol type to be specified as structured, along with an informal description of its structure.

Various situations can give rise to multifield representations, including:

1. Arbitrary design decision, e.g., addresses.
2. The entity is a relationship. For example, in expressing a fact about a marriage, the marriage might be represented by the identifiers of the two people involved.
3. Additional entities are introduced as qualifiers to provide unique identification. For example, state or country names are introduced to provide unique identification for cities. This is especially significant when such facts are introduced purely for identification purposes. For example, there may have been no desire to record which state or country a city is located in.
4. Multiunit measurements. For example, separate fields for feet and inches.
5. Self-describing representations. For example, auxiliary fields for the field length, unit of measure, scale, etc.
6. Multiple entity types. For example, when holders of company assets can be either departments or employees. There are two possible treatments: (1) When the representations are compatible, then there might be an auxiliary field to indicate the entity type which is present. (2) When the representations are incompatible, a separate field may be provided for each entity type. All but one are blank in each record. And there still might be one more field indicating which entity type is present.

Derived and indirect representations. Sometimes an entity will not have a distinctly visible representation in the data. Such an entity will not (directly) correspond to any field.

For example, an entity might be inferable from a related entity, or from its representation. Dates are sometimes embedded in the identifiers of things (date of claim is part of claim number, birth date is part of some European person numbers). A great deal of information may be embedded in part serial numbers: part type, place of manufacture, certain technical characteristics, etc.

In some cases the inference is more computational. Part numbers in certain ranges, or employee numbers in certain ranges, imply certain company locations. In some countries, person numbers are odd or even according to sex. Day of week can be computed from a date. Winners of elections can be computed from a vote count.

In the extreme case, some entities don't have any identifiers of their own at all, but are only known indirectly via some related entity. Elections don't have identifiers, but are known by the year in which they occurred: "the 1980 election." Sales territories might only be known by their headquarters cities: "the Chicago territory"

Nonrepresentation. The representation of some entities is derivable from the "context." Such entities are constant over all occurrences of the record type, and hence don't have to be identified via any data field. A given company is always the "seller" in its own sales records, and always the "buyer" in its own purchasing records. The playing schedule for a particular team will not list both teams for each game.

Sometimes the identity of an implicit entity is included in the record name. For example, if we have separate employee records for each job type, then we don't need a field for the employee's job. If we have separate employee records for each company location, then we don't need a field for the employee's location.

And sometimes entities are not represented because we don't care about identifying the individual! Such entities are only involved as intermediaries in specifying the fact. For example, we might record the phone number of an employee's secretary, without identifying the secretary. (In some environments, that changes too fast.)

Compressed representations. The derived and indirect representations mentioned above essentially compress several entities into one field. In effect, one field represents a sales territory and its headquarters city. Another field represents a person (via his person number) and also his date of birth and his sex.

In practice, this is sometimes carried to the extreme of compressing several entities into one field simply because their representations are "too short"; letting them each occupy a single field would be inefficient.

For example, if sex and marital status were each representable by a single-character code, they might be combined into a "personal status" field containing the two characters. Even more extreme are computational forms of encoding: e.g., digits 1–4 represent the marital status of males, 5–8 the marital status of females.

While this is not considered a fit topic of discussion in logical data design, it is in fact a common practice which alters the logical design (field layout) of a record. It needs to at least be acknowledged, and documented when it does happen.

Mapping techniques. While some representations can be expressed in terms of symbol facts, we don't require that for all cases. It makes the methodology too cumbersome.

In effect, we will simply allow any entity to be replaced by a suitable representation. This representation might cause the introduction of multiple fields, or it might cause several entities to be represented in the same field, or the representation might occupy no fields at all. Sometimes additional facts will have to be "imported" in order to achieve suitable representation.

For documentation purposes, it would be advisable to record the mappings between the data fields and their corresponding roles in the facts. As indicated, such mappings may be rather elaborate.

"Quality" of the representation. We have so far described some assorted techniques for representing entities, e.g., by simple symbols, structured symbols, indirect reference, etc. We haven't yet said that such representations are necessarily unique. Uniqueness is just one of a number of qualities that a representation might have.

The qualities are not always all required. The overall
process of selecting representations involves three distinct steps:
1. Determine the quality of representation required.
2. Determine what representation techniques are available.
3. Select an appropriate one, or introduce something new if needed.

We first discuss some of the qualities of representation.

**Existence.** The first desirable quality of a representation is that one should in fact exist. Most things in the world around us don't have simple identifying labels. We refer to most things by a combination of pointing, complex noun phrases involving relationships to other things, and default assumptions regarding context and implicit qualifiers.

Entity types without direct representations include the earlier examples of elections or sales territories, and the individual pieces produced by some assembly lines. Animals are another example (though pets usually have representations, while wild animals usually don't).

Data representations for such entities can usually be obtained either by assigning arbitrary new identifiers (part numbers), or finding related entities suitable for indirect representation.

**Completeness.** A complete representation provides a representation for every member of an entity type. Employee numbers are complete for employees (of a given company), but not for people in general. Maiden names are incomplete representations. Representations that are incomplete can't really be used for any effective sense of "representation," but would simply be recorded as facts about the entities. Military service numbers, social security numbers, and maiden names might occur in this way in employee records.

**Uniqueness.** A unique representation meets the common requirement for identification: a representation corresponds to at most one of the entities. People's names are generally nonunique, the same string can correspond to several people.

Nonunique representations commonly occur in data, and can still be quite useful to human users. Such users frequently make use of subtle default assumptions and local conventions. It is a natural and useful device, not to be rejected automatically as poor design.

Uniqueness is mainly required when the agent that will retrieve or interpret the data won't take advantage of such assumptions or conventions. Such an agent might be a computer (database manager) or a file clerk.

In order for such an agent to retrieve a record about an employee, the employee's identifier must lead only to his record and nobody else's. If we want the agent to follow a path, perhaps to go from a department record to the employee record of its manager, then the manager's identifier must lead only to his record and nobody else's.

Uniqueness is especially important when the simple-minded agent is only able to cope with one record at a time. Nonuniqueness will disrupt the processor, if it retrieves several records when it was prepared to receive only one.

Finally, uniqueness has some significance simply because it is mechanically enforceable. The simple-minded agent can detect and reject multiple occurrences of an identifier, if and when the agent is instructed to do so.

**Singularity.** A singular representation provides each entity with at most one corresponding representation. This is the reverse of the notion of uniqueness. Social security numbers, though unique, are not singular: a given person might have several of them.

Singularity is not always required, and is sometimes undesirable. Measurable quantities often need different representations in different contexts—at the very least, different units of measure. Sometimes it is important to represent people by employee numbers in one context and by social security numbers in another.

Lack of singularity could lead to:

Failing to find an entity's involvement in a fact, by virtue of using the wrong representation. This could occur either in retrieval, or in path following. Some constraint enforcement also involves a form of path following. For example, to be able to check that a department's manager is a real employee, the manager should be represented in the same way in the department file and in the employee file.

Inadvertent violation of uniqueness, by allowing the same entity to participate several times, once under each of several representations.

Lack of singularity could be compensated if the processing agent could recognize the equivalence of several representations of the same thing, and perform conversions. Such conversions might be computational, as for measured quantities, or supported by other stored facts, such as the mappings between social security numbers and employee numbers. Current data processing agents rarely provide such capabilities, making singularity all the more important. The ability to navigate through nonsingular identifiers is beyond the capability of current query processors (relational or otherwise) [9].
While uniqueness is a widely recognized requirement for identifiers, singularity is little mentioned in the literature. This might be for several reasons:

The distinction between uniqueness and singularity is not recognized, and people really mean both when they say "uniqueness."

There is a historical bias towards machine requirements as opposed to human factors. Nonunique identifiers confuse the data management system, while nonsingular identifiers only confuse the users, not the system. As long as the identifier leads to a single record, the system is content. It's not the system's problem if a person has two social security numbers, so that some facts about him go in one record while others go in another.

There is no mechanical way to enforce singularity, so why bother talking about it.

We mention in passing that there are at least three ways to violate singularity:

An entity might correspond to two symbols within the same symbol type, as with social security numbers.

An entity type might have several different symbol types for its representation, as with social security numbers and employee numbers.

An entity might belong to two different entity types, each having its own identifying symbol type, as with employees and military personnel.

Other qualities. We mention some other qualities of representations that need to be considered. The importance of such criteria is a matter of judgment to be exercised during the design process.

A representation should be stable. If it were subject to change, then it would have to be updated wherever the entity was mentioned. This applies to "direct" representations, e.g., names, and also to indirect and qualified representations. It would be inadvisable to name sales territories after their headquarters cities if the headquarters are likely to move. It is only reasonable to qualify city names with their states because we don't expect state boundaries to shift, transferring cities from one state to another.

It is sometimes held that the ideal identifier contains no information. For example, embedding dates into serial numbers is considered undesirable. However, it would seem that embedded information is acceptable provided that (1) the information is stable, and (2) the embedding of information can be documented in the dictionary or other data description.

Mnemonic identifiers are generally easier for humans to deal with.

Simplicity is generally preferable. Direct and simple representation is preferable to indirect, derived, or structured representations.

Length is a factor. Anything can be uniquely identified if we can write enough paragraphs about it. Representations that are too long, or too variable in length, are undesirable. Even representations that are too short can sometimes create problems, leading to the kind of compression discussed earlier.

Where possible, uniform representations are preferable, for example, the same representation for the same entity type, where possible. Also uniformity within representation is preferable to self-describing forms. This maximizes the amount of information that can be factored out, minimizes the amount that has to be stored, and simplifies the processes accessing the data.

Using combinations of facts. Completeness, uniqueness, and singularity can be described in terms of participations in a fact which provides representation:

\[
\begin{array}{c}
\text{THING} \rightarrow \text{REPRESENTATION} \\
1 \text{ if complete, else 0} \\
1 \text{ if singular, else n} \\
\text{don't care (usually 0 for some names aren't used)}
\end{array}
\]

By representations here we mean one or more related things. Normally they would be symbols, but they might not be if we are dealing with indirect or qualified representations. If they are not symbols, then they in turn will have to be provided with representations. For example, cities can be identified by a combination of city names and states, where the states in turn have to be provided with some representation. For the unique identification of the cities, it doesn't matter whether the states are identified by full names or abbreviations; we don't have to specify that as part of the city representation.

Very often, as in the case of qualified names, the representations will be obtained from combinations of relationships, such as:
The participations for such combinations have to be specified, since they cannot be derived from the component relationships. In effect, these constitute additional constraints.

2.4 Other Design Concepts

Entities and records. Choosing the record types is an essential part of data analysis and design. In the present methodology, records are chosen in an objective and systematic way, based largely on the pattern of one-to-many and many-to-many relationships in the facts to be maintained.

Earlier methodologies choose records on the basis of a more restrictive "entity" concept (which we will call "major entities"), on the assumption that a record corresponds to an entity type. There are several difficulties with this approach:

1. It is not always obvious which are the major entities. While various guidelines are offered, they tend to be vague and subjective (e.g., "the things which are important to the business, about which data is to be kept"). In practice, choosing the major entities often inverts the design process: users develop an ability to choose major entities that will correspond to the records they want in the data design.

2. The correspondence tends to break down in the design process anyhow. Records corresponding to the major entities are taken to be a first approximation. The design process will sometimes remove some of these records, and add others. In the end, the records in the database don't correspond to the initial set of major entities. Very often, users are implicitly directed to alter their views of the business, by documenting major entities corresponding to the final set of records.

3. A corollary decision is often forced, namely a distinction between "relationships" and "attributes." This decision is sometimes equally difficult to make, and depends somewhat on the choice of (major) entities. One guideline is that a relationship is an association among entities, while an attribute associates entities with "values" (an ill-defined concept).

4. Major entities can't overlap, e.g., can't be subsets of one another, because record types can't. Users are forced to specify either "employees" or "clerks" and "salesmen." Whichever they specify will determine the records in the data design. Users are unable to say that they perceive clerks and salesmen as subtypes of employees, leaving the choice of records to the data design tool.

Though our approach is more direct than the "major entity" approach, the results are often quite similar, and for good reason. The major entity approach works fairly well because if an entity type is "major," then we are very likely to have single-valued facts about it, and hence we will naturally have a record for it. Perhaps the popular notion of an entity being "something about which we maintain facts" should be revised to "something about which we maintain single-valued facts."

Existence management. Sometimes another significant characteristic of "major entities" is that their existence needs to be explicitly managed. Some entities can come and go freely in the database, without any particular announcement of their arrival or departure. We are usually willing to mention a length, or a color, as a fact about something, without announcing that that particular length or color exists. On the other hand, we don't want to refer to an employee without prior assurance that such an employee exists. That is, for some entities it is necessary to have some mechanism for announcing their existence, and also their erasure from our sphere of interest. This can be a criterion for identifying major entities.

Although we try to treat this as a high level semantic concept, the only mechanism for such announcements in conventional database systems is to insert and delete records. Therefore, it becomes important to have a record type corresponding to an entity type for which such announcements are required.

This is rarely significant, in practice. It almost invariably turns out that single-valued facts will be maintained about anything whose existence needs to be managed, so they will have records anyhow. Exceptions to the rule almost have to be contrived. One case might occur if all the facts we wanted to maintain about something happened to be multivalued. For example, if projects had multiple managers and multiple completion dates and multiple goals and multiple budgets, then we might have many "intersection" records for these m:n relationships, but no one record keyed on project numbers. An explicit "announcement" specification would be needed in this case.

In our methodology, we can take advantage of unary relationships to accomplish this. For any entity type T for which such "existence management" is required, specify "exists(T)" as a fact to be maintained. A pseudo-record with one field will be generated, and the field will be a pseudo-key. If other records exist with the same key, then this record will vanish harmlessly during the merge step. Otherwise it will survive into the final design as an "existence list" for such entities.

Complex situations. Other methodologies, based on earlier concepts of entities, relationships, and attributes, tend to have difficulty with:

N-ary relationships. For example, buyer X bought item Y from seller Z.
Attributes of relationships. For example, people started working for companies on dates.
Entity subtypes. For example, employees and salesmen.
Multivalued attributes. For example, multiple colors of an object.
Attributes of attributes. For example, what percent of the object has each color.
Complex identifiers. For example, multifield or imbedded.
These cases are dealt with naturally in our methodology.

Normal forms and normalization. Normal forms [11] are valuable as a design objective, yielding designs with minimal redundancy and minimal exposure to update anomalies, sometimes with attendant performance penalties for retrieval operations.

However, some methodologies apply normalization as a process, in which badly-designed records get decomposed (normalized) into better records. Ours is a "synthetic" approach [2], constructing records in normal form directly.

Intuitively, it seems clear that record designs produced by this methodology are in fifth normal form [11]—which also implies the first four normal forms—at least prior to the step where alternative designs are introduced. Informal support for this hypothesis is that (1) the initial pseudo-records are in fifth normal form, and (2) merging on keys preserves normal forms. Formal arguments may be hard to develop, since merges are not quite the same as relational joins (see Section 3.4).

Derivable facts. We are implicitly assuming that the specified facts are independent of each other, in the sense that one can't be derived from the others. This might be violated, e.g., if we specified the facts:
Employees are assigned to departments.
Departments have managers.
Employees have managers.
These facts are redundant. If implemented unintentionally, we would have a badly designed database. At the very least, we should require that derivable facts be separated from the others. If it is necessary that they be made available from the database, there are two major design alternatives:
1. Store them redundantly. This should be augmented with some facility for checking the consistency of the redundant data.
2. Derive the data on request from the underlying primary data. This option does not allow for updating the derivable information.

A similar assumption is that the specified facts have been decomposed into independent components as much as possible. For example, the fact that employees have managers in departments is not, in itself, redundant. But this should (assuming a normal company policy) be presented in the more elementary form:
Employees are assigned to departments.
Departments have managers.

These assumptions, if formalized, would underlie the demonstration that this methodology can yield record designs in fifth normal form.

3. THE DESIGN PROCESS

Up to this point we have been developing the underlying concepts. We are finally ready to present the design process itself, in some detail. We suspect that, as with many other things, doing it will be easier than describing, teaching, or learning it.

3.1 Specify the Facts to Be Maintained

These will be specified in terms of relationships among entity types. In practice, we expect that the facts, relationships, and entity types will get defined in an iterative, concurrent fashion. Sometimes it may be easier to start listing the entities first, and then to collect the facts about them. New facts and relationships may in turn give rise to new entity types.

The set of entity types and relationships should be useful beyond a single application or database. The design of subsequent databases will probably involve many of the entities and relationships established in previous designs, although some new ones will undoubtedly be added.

In the following, we indicate the information needed for each construct. Implementations of this methodology can provide various mechanisms for specifying this information, including possible default rules.

For each fact, specify:
A fact identifier (arbitrary).
The relationship name, or sequence of composed relationships.
The entity type occurring in each role, if more restrictive than that defined for the relationship.
Least and most participations, if more restrictive than in the relationship definitions.
For each relationship, specify:

- **Relationship name.**
- **Degree (number of roles).**
- **Role names.**
- The entity types permitted in each role.
- Least and most participation for each role, and also for selected sequences of roles.

For each entity type, specify:

- **Entity type name.**
- Whether it is a subtype of another entity type.
- Whether it is a symbol type and, if so, whether it is structured.

We need not assume that all specifications are provided at the outset. In particular, additional facts to support representations may be required at a later stage.

**Examples.** We will follow two example situations: employees taking courses and people winning elections. We treat them as an extension to the introductory example, so that the facts presented there are also available, particularly if needed for representations.

For brevity's sake (though it's probably too late for that), we will not illustrate all aspects. We present the facts immediately in diagram form, with role names defaulting to entity type names:

3.2 Generate a "Pseudo-record" for Each Fact

A pseudo-record is just a fact in disguise. It contains one field for each role associated with the fact. When actually carrying out the methodology, we would record how each field relates to the fact, i.e., the field's corresponding relationship, role, and entity type. The least and most participations for each role or set of roles are inherited by the corresponding fields or sets of fields.

We do not as yet care anything about representations. If the role actually involves a symbol type, then we expect symbols to be in the field. Otherwise, we pretend that whatever thing is playing the role is actually sitting in the field. For example, in the pseudo-record corresponding to the fact that employees have employee numbers,

we imagine that the second field contains employee numbers, while the actual employees are sitting in the first field. It may be difficult to imagine some things in the fields, such as colors or days or weights (the actual heaviness, not any numerical measurement of it), but that's the appropriate mental picture at this point.

Pseudo-records could be candidate records for the final design, but they are likely to have two undesirable characteristics at this point:

- Most fields don't contain machine-processable representations.
- The records are very fragmented. There is essentially a separate record for each "attribute" of an entity.

A shortcut. Representations will be dealt with at a later step, making allowances for a large variety of difficult situations. However, we can take a shortcut at this point which will keep the methodology simple for simple cases, and which brings our methodology more in line with others.
When an obvious and adequate representation is available for an entity type (e.g., employee numbers), then we can use that representation immediately as a "surrogate" for such entities, as follows:

1. Eliminate the pseudo-record expressing that representation. For example, eliminate the pseudo-record corresponding to "employees have employee numbers."

2. Substitute the symbol type for the entity type in all other pseudo-records. For example, substitute EMP# for EMP everywhere.

In our imaginary picture, we have now actually let employee numbers occupy the fields, replacing the employees themselves.

In the present series of examples, we only take the shortcut for employees, replacing EMP with EMP#.

3.3 Identify "Pseudo-keys" for Each Pseudo-record

A pseudo-key is a sequence of fields that could be an actual key if all entity types involved had simple unique identifiers. A key is a set of one or more fields that provides uniqueness of records. A given entity or sequence of entities may only occur once in these fields. More precisely, we are only interested in minimal sets: a set of fields will not be considered a key if any subset is a key.

The main value of keys in record structures is that they enforce single-valuedness of facts, i.e., maximum participations of 1. If an employee's number is only permitted to occur in one employee record, he is thereby limited to having only one entry for his salary, his department, etc.

A pseudo-key corresponds to a role or set of roles in a fact having maximum participation of 1 (participations of 0 or 1).

Example (we use "====" to designate the pseudo-keys):

Intuitively, we want to merge these records to mimic the way a data record provides several facts about one "subject." More precisely, these are single-valued facts, so that the "subject" corresponds to a key of the record. Having several single-valued facts about the same subject exactly corresponds to having pseudo-records with the same key. Hence what we will merge are pseudo-records having the same key. The result of the merge will be a new pseudo-record containing all the fields of the contributing pseudo-records, without replicating the shared key.

Merging on simple keys. A simple key consists of one field.

Ideal merges. Merging is most likely to occur on simple keys. Two pseudo-records may only be merged if their corresponding keys are going to contain the same occurrences of the same kinds of things. The strictest way to assure this is to require that the corresponding keys contain the same entity types, and that they will both contain the full population of the entity type. In other words, the least participation for each key should be 1. Since all keys have an MP of 1, these keys will have LP*MP = 1.*

An "ideal" merge is thus possible when the two keys involve the same entity type and both have LP*MP = 1.*

Example:

(Note: in these examples, a field number of the form $^{*}$ means pseudo-record field number 23 with field number 2 substituted to provide a representation. Some of the numbers refer to the first example in the paper.)
Fact-Based Data Analysis and Design

Unequal populations. The restrictions can be weakened when we can "pad" the populations. In many cases, a pseudo-record whose key has a least participation of 0 can be "padded" to include the missing entities, with some sort of "null" values for the related entity. For instance, instead of excluding employees who have no departments from the "assignment" pseudo-record, we might include them with a blank department, or some other indication of null. Such padded pseudo-records will then have a full population of values in their keys, and they can be merged with other pseudo-records having either 1*1 keys or similarly padded keys.

Subtypes. The padding concept can be extended to allow merging of entity subtypes. The entity type in any key field can be "promoted" to some supertype, with the key having a least participation of 0. For example, the fact that "salesmen serve territories" can be promoted to "some employees serve territories."

The resulting pseudo-records can then be merged as above for unequal populations. Either one or both of the pseudo-records might be promoted. Thus this technique might be used to merge facts about salesmen with facts about employees, or it might be used to merge facts about salesmen with facts about clerks by treating them both as employees.

As before, the feasibility of such merges depends on the acceptability of null values. In practice, such promotions will generally be limited to those cases where suitable identifiers are available for the supertypes.

Such merges will sometimes require the importation of additional facts for distinguishing the subtypes, e.g., a job-type fact to distinguish salesmen from clerks.

Summary. Merging pseudo-records having simple keys is possible if:

1. The entity types in the keys are the same, or one is a subtype of the other, or they are subtypes of a common supertype.
2. The key fields will have the same populations, or the populations can be equalized by padding the fields outside these keys with null values.

Although these pseudo-records have simple keys, the corresponding records in the final design might have compound keys, introduced for representation purposes. A qualifier field might be introduced for uniqueness, or a single entity might be represented by multiple fields.

Merging on compound keys. A compound key contains more than one field.

In pseudo-records, ordinary entities always correspond to single fields. Compound keys correspond to facts. Merging on compound keys requires that the keys represent the same fact, which is a stronger requirement than that the keys contain matching entity types. Matching entity types could be involved in different relationships. Merging compound keys is, in effect, merging attributes of relationships, and we should only merge attributes of the same relationship. For example, we can merge pseudo-records corresponding to the following facts, all of which would have compound keys consisting of people and companies:

Fact 1: people have worked for companies.
Fact 2: fact 1 began on date.
Fact 3: fact 1 ended on date.

It would not be appropriate to merge these with some other facts involving people and companies, such as which people own which companies.

It is still necessary to be concerned with matching populations. In the example above, fact 3 would not exist for people in their current jobs. In order to merge, it must be acceptable to pad fact 3 with occurrences having some form of null dates.

Pseudo-records with multiple keys. A pseudo-record with multiple keys corresponds to a one-to-one relationship, e.g., between department and manager (employee), or between employee and spouse. This creates an opportunity for several entities to be "subjects" in the same record, which is sometimes desirable and
sometimes not. We probably wouldn’t object to combining the facts:

Employee born-on date
Employee married-to spouse
Spouse born-on date
into the record:

But the same process allows us to merge all department information into the employee records, by putting all such data into the records of the employees who manage departments. That option appears undesirable.

Extensive merging on both subjects becomes inadvisable if the connection between the subjects is likely to change. That’s one reason not to maintain information about a department in the employee record of its manager.

Another concern has to do with nulls. If one of the pseudo-keys has 0*1 participation, and the record is merged on that key, then all other fields in the record must admit null values—including the other pseudo-keys. We can call a pseudo-key that has to admit null values a soft key. Further merging on a soft key is generally undesirable, since softness is propagated. Whenever a record is merged on a soft key, all the fields in the other record must be able to accept nulls, including the key in the resulting merged record.

For example, consider the following pseudo-records, in which we use “=-.-=” to indicate keys with 0*1 participation:

Suppose we merge the first and third of these on the DEPT# field in order to combine information about departments. Since 14 is a 0*1 key, all other fields in that record must admit nulls after the merge, making 13 a soft key (indicated by “:”):

In the new record, field 13 must admit null values, since some departments don’t have managers. Suppose now we want to pursue our earlier trick of capturing all information about a department in the employee record of its manager. We can do that by merging these last two records on the EMP# fields—provided we are willing to have nulls in both the EMP# and MONEY fields in the result, to allow for the departments having no managers. (The reader is encouraged to work this out in terms of example records.)

**Merging is optional.** The natural and appropriate inclination is to merge as far as possible. It is preferable to aggregate all facts about an entity into a single record, creating a model in which that record “represents” that entity. That minimizes problems of “population management,” e.g., making sure that the set of employees who earn salaries is the same as the set of employees assigned to departments. It provides a “master file” concept, i.e., a central list of all the currently existing entities of that type.

However, merging to the maximum degree is not always desirable. Some considerations are logical, others physical.

Merging can be carried to absurd extremes via supertypes. Almost any pair of types can be perceived as subsets of a larger type (resources, assets, products, physical objects, priced things, etc.). Theoretically, these could drive some designs to a single monstrous record with nulls permitted in most fields. In practice, such merging is limited to cases where suitable identifiers are available for the supertypes. Occasionally, new identifiers will be invented to facilitate such merges.

Other considerations are more physical, and probably should not be considered in a purely logical design. Possibly wasted space for null values is one consideration. Or, merging might be undesirable if different facts are used exclusively by different applications, or on different systems, or at different locations.

The option to merge or not merge overlaps the topic of Alternative Designs (Section 3.6).

**Merge vs. join.** The merge step of this methodology is quite similar to the join operations of the relational model. There are a few distinctions to keep in mind.

In the first place, merging is a design step, rather than a database operation. That is, we only manipulate data descriptions, not actual data.

Nonetheless, merge operations could be visualized as actually combining record occurrences, just like a join operation. The essential difference here is that merging combines records on the basis of matching entities and facts, whereas the joins involve matching representations and/or matching attribute names. Merging is not even limited to matching entity types, since subtypes of a common supertype may be merged. Thus merging is much like the “entity join” proposed in [9]. Furthermore, joins delete records which aren’t matched ("dun-
gling tuples"), whereas the merge retains them by matching them with null-padded copies of the other record. In this respect the merge is like an "outer join" [4].

3.5 Assign Representations

At this point, our pseudo-records are a mixture of symbols and nonsymbols. In order to convert these to proper records, we have to eliminate the nonsymbol fields, making sure they are adequately represented via symbolic fields. Most of the time, this simply means replacing an entity type with a suitable identifier type. But sometimes other things happen.

We repeat an earlier note: most methodologies assume that a simple representation is available for all entities. What we deal with in this section goes beyond the capabilities of other methodologies. The reader who wishes to deal with the simple case first, or who wishes simply to compare this with other methodologies, is encouraged to skip this section on first reading.

To assign representations, do the following steps for each field in the pseudo-record, including any additional fields introduced in the process:

1. If the entity type in the field is a structured symbol type, replace by multiple fields, and exit.
2. If the entity type is a symbol type, leave it alone, and exit.
3. If the entity type needs no representation, delete it, and exit.
4. If the representation is derivable from another field (including indirect representation), delete this field, and exit.
5. Find a set of facts whose combination gives the required qualities of completeness, uniqueness, and/or singularity (as well as other qualities). In the vast majority of cases, this will consist of just one simple symbol fact. Introduce new facts if needed. Facts suitable for representing a supertype may be employed. For example, in seeking identifiers for salesmen, identifiers for employees may be used. If the related things are not symbols, the process may later have to be repeated for their representation.
6. If several suitable sets of facts are available, choose one.
7. "Import" such facts into the record, if not already there.
8. Delete the field.
9. Repeat the assignment of representations for any imported facts.

When finished with all records, it may be necessary to remerge, to eliminate redundancy of imported facts. For example, after a state field is imported into a record to provide unique qualification for a city, it may become redundant to maintain a separate record telling what state each city is in.

Example. We illustrate these steps in detail for record R1 (continuing from the section, "Ideal Merges".

Steps 1–4 don’t apply.

Step 5: we introduce two new facts needed for the representations of courses and subjects:

Step 7: import these facts into record R1:

Step 8: delete the fields for which we have just provided representations:

Step 9 requires that we repeat the process for the newly imported fields, but by Step 2 we find that these can be left alone, since they already contain symbols. We keep track of the entity types and representations in the final result:

For record R2 (last seen in the section "Ideal Merges"), the main difficulty is in finding a unique identifier for offerings. The best we could find (by asking the people at the school) was that offerings were numbered uniquely only within course. Informally, what we will do for this record is to replace OFFERING by the offering number, recognize that the proper course is already in the record, and allow the combination of offering number and course to represent the offering. When finished with all fields in the record, consider compression, as discussed in the section, "Compressed Representations."
Steps 1-4 don’t apply. We begin at Step 5 by introducing a new fact assigning offering numbers to offerings.

We then recognize that the following combination of facts is required to uniquely identify offerings:

\[
\text{has-number} \\
\text{has-off} \\
on-off \\
\text{OFFERING OFF# COURSE} \\
\text{identifies}
\]

At Step 7, we observe that the “has-off” fact is already in the record, but we have to import the “has-number” fact, yielding:

We eliminate the OFFERING field itself at Step 8, and then apply the representation process again to find simple representations for the COURSE, LOCATION, and DATE fields. The final result:

In record R3, we again have to provide a representation for offerings. The process is much the same as before, except we now have to import both facts (course has offering and offering has number), since neither had been in the record. The result is the importation of a COURSE field:

In effect, the fact connecting courses (17) and offerings (18) now must be maintained redundantly in records R2 and R3, purely to solve a representational problem. To see the significance, note that this record would not mention courses at all if offerings had unique identifiers. (We might observe that this redundancy violates no normal forms.)

For record R4, we elect to identify elections indirectly by their year of occurrence, and simply eliminate the ELECTION field by Step 4. After supplying suitable representations for the other fields, we have:

The odd thing about this record is that, although it is all about elections, it contains no mention of elections!

3.6 Alternative Designs

This step of the methodology is not fully developed as yet. The kinds of design alternatives to be considered are described in [10] and [12]. [10] has a preliminary proposal for transforms to systematically generate some of the alternatives mechanically.

Each alternative design corresponds to some different way of specifying the facts. Fact specifications are not particularly unique. Even using entity/relationship concepts, there are many ways to describe a given situation. Our objectives are:

- To allow the user flexibility in how he chooses to provide his application description.
- To let the final design be driven more by the processing requirements than how the user happened to phrase his application description.

A variation of the first goal is to allow application descriptions to be standardized without necessarily imposing a corresponding rigidity on the data designs. It is very useful to be able to document that similar information has been implemented in different ways in different databases.

Of course, even our methodology cannot escape giving some preference to some designs. How the user phrases his application description will determine which design we generate first. The others are generated later in some extra steps. It is unavoidable that some users will learn to phrase their application descriptions so that their “preferred” designs are generated first. The best we can do is to make the alternatives available, in case the user really didn’t have a preference in mind, or in case he had neglected to consider some better alternative.
3.7 Naming Fields and Record Types

In general practice, there is no systematic discipline for the naming of fields and records. Hence there is nothing predictable about what such names signify. At best, there might be various guidelines and local standards.

Often, a field name includes indications of some or all of the following things, abbreviated and concatenated in various ways:

- The entity type represented in the field.
- The form of representation used.
- The relationship in which it is involved.
- The role played in that relationship.
- An arbitrary label.

The only uniform requirement seems to be that field names be unique within a record. This precludes the use of any one of those constructs consistently as the field name. Even role names may not be unique enough. In fact, the same relationship might occur twice in one record, definitely rendering the role names nonunique within the record. Example: a record containing two birthdays, the employee's and the spouse's.

We give arbitrary names to our fields:

```
"COURSE RECORD" (R1):  COURSE#  SUBJECT
"OFFERING RECORD" (R2):  OFF#  COURSE#  LOCATION  DATE
"EDUCATION RECORD" (R3):  EMP#  COURSE#  OFF#  GRADE
"ELECTION RECORD" (R4):  DATE  WINNER
```

3.8 Iteration

The whole database doesn't have to be designed at once. You can start with any initial set of facts, relationships, and entities, and then come back and add more later, as many times as needed. The key steps in subsequent iterations are:

- Specify a new pseudo-record type for each new fact introduced.
- Specify representations and keys.
- Merge with previously generated record types.

4. CONCLUSION

4.1 Highlights of this Approach

We focus more on what facts are to be maintained, rather than on what the facts are about. We start with one record per fact, rather than one record per entity.

The user isn't forced to make distinctions between such things as "relationships" and "attributes" (or was that "entities" and "attributes"?)? While there are times when he might feel more comfortable making that distinction, there are too many cases where it is hard to decide (e.g., birthplace).

Neither is the user forced to choose between entity types and subtypes.

Other design schemes have one stage for handling relationships, another for handling attributes. But they do practically the same thing in both stages. We don't distinguish between them, and use one process for handling both.

Some design schemes ask the user to proceed in two stages, first for "important" things and later for "details." There is no criterion for deciding what's important. But such methods really expect "important" things to be entities for which there will be corresponding records. We don't have any such expectations. Our design scheme is homogeneous. The user can deal with everything at once if he wants to. Or he can iterate in as many stages as he likes; if appropriate, he can view these as increasing degrees of detail.

Some methods feel that "keys" are the key (pun intended) to entities. If something has a unique identifier, then it is an entity and should have a corresponding record. But that doesn't always hold in practice. Even if cities had unique names, we wouldn't necessarily want a distinct record type for cities just because we happened to be recording people's birthplaces. Our approach defers treatment of unique identifiers until later, after we've determined when they are needed.

We provide a rational and objective basis for choosing the record types.

We use a simplified form of entity-relationship model. It conforms more closely to the way the user perceives his enterprise, rather than to the way the data is structured. He can therefore express his requirements more easily, without being forced to make premature data design decisions.

We go beyond other methodologies in suggesting alternative designs for the application.

The approach is synthetic [2]. While the value of normal forms is recognized as a sometimes desirable design objective, we do not adopt the process of normalization, in which badly assembled records are de-
composed (normalized) into good ones. We construct normalized records directly.

4.2 Indirect Benefits
Conceptual models

Facts are specified in terms of entities and relationships, which themselves ought to be documented. There are various approaches to developing the necessary collection of entities and relationships:

- Develop as needed for a database design, and then augment it for subsequent databases.
- Document the meanings of existing data.
- Do a top-down analysis of the business.

Whatever the approach, the resulting collection of entities and relationships would constitute at least the nucleus of a conceptual model [1] (business data model, enterprise description) for some portion of the business.

Data documentation. Keeping a record of how the data elements express facts, in terms of the entities and relationships of the business, would constitute excellent documentation of what data is being maintained and what it means.

Understanding the application. Data elements inherently do represent facts, in terms of entities and relationships, in the form we have developed. If you have trouble organizing your requirements in the form of such facts, then either the application is not suitable for implementation in a database, or you do not adequately understand the application.

A necessary part of the analysis leading to the design of a database consists of identifying the entity types involved, the relationships among them and, separately, the representations to be used. Trying to organize the requirements in that form invariably leads designers, and users, to a clearer understanding of the application.

4.3 Convergence with Other Work

This work is an outgrowth of entity/relationship theory [3] and its semantics [8]. and traces at least back to the "irreducible relations" of [6]. It is also an extension of "synthetic" design methodologies [2]. The aims of our work are close to those of Imielinski and Lipski [7]:

... we believe that the atomic, self-explainable relations, in terms of which all other relations considered are explained, should constitute the input of the design process. The design process should construct more complex database relations in terms of the semantics of input relations.

The work also converges with recent developments in the universal relation branch of relational database theory. Our facts and pseudo-records approximate the objects, associations, basic facts, and base relations of [5, 13, 14].

The principal differences between our work and the others:

- We are based on a deeper semantic structure, which distinguishes facts, relationships, roles, entities, and representations. In particular, we deal with entities and facts rather than representations and unlabeled dependencies, which don't always correspond well to these deeper semantics.
- We allow a more realistic set of transformations to the final data configurations to be used.
- We recognize that a variety of data structures might be used to represent the same facts. We let the user specify the required facts in a flexible way, and we generate a variety of suitable designs, not restricted by the form in which the user chose to describe the facts.

REFERENCES


