Data and Reality

Data and Reality

William Kent

cover design: helen holder / photos: bill kent

ISBN: 1-58500-970-9 (softcover)
ISBN: 1-4208-9888-4 (e-book)

1stBooks – rev. 3/28/00

ABOUT *DATA AND REALITY*

First published over twenty years ago, this little classic addresses timeless questions about how we as human beings perceive and process information about the world we operate in, and how we struggle to impose that view on our data processing machines. The concerns at this level are the same whether we use hierarchical, relational, or object-oriented information structures; whether we process data via punched-card machines or interactive graphic interfaces; whether we correspond by paper mail or e-mail; whether we shop from paper-based catalogs or the web. No matter what the technology, these underlying issues have to be understood.

You can read this book for insights into the basis of computer data processing. You can also read it for insights into the way we perceive reality, and the constructs and tactics we use to cope with complexity, ambiguity, incomplete information, mismatched viewpoints, and conflicting objectives.

This new edition preserves the original content with minor cleanup and a new preface. The format, though, has been thoroughly modernized. That ugly typewriter font is gone! It's now a pleasure for the eyes as well as the mind. And it's still as relevant as ever.

What they've said about Data and Reality…

An excellent, philosophical discussion of the problems inherent in describing the real world. There is nothing really similar to this work. I think that all data base researchers should read this document. It might also be assigned as supplementary reading in general graduate and undergraduate courses in data base systems.

–Mike Senko (1978)

I expect the book to be one of the most frequently quoted ones for the next few years. It is unique in being an almost exhaustive, condensed rendition of the typical problems encountered. The most striking strong point is its penetration into major data base technology headaches… Many well chosen examples and the lucid style make it easy to read.

–Reiner Durcholz (1978)

…highly recommended and even required reading for all DP people…

–G.M. Nijssen (1978)

Kent has produced a rather remarkable and highly readable short work… the most important things he has to say are philosophical and go right to the heart of the key concepts that must be understood if a system is to be "successful" (whatever that may mean!)…This is a serious book but not a heavy one. Kent writes easily and without hiding behind the semantics of the data base specialists. The ideas are presented in a straightforward manner with no attempt to preach.

–Datamation, March 1979

This excellent study of the problems inherent in describing the real world is unique in (1) being an almost exhaustive, condensed rendition of the typical problems encountered, (2) not offering an own solution as remedy for all evils, and (3) penetrating into the mists of conceptual ambiguity… This book is of important value to all those in the field of data bases and information systems who are concerned with developing a deeper understanding of this matter. It is of equal importance to the systems analyst, to the data base designer, and to the database system designer.

–Current Engineering Practice, July 1979

Data and Reality illustrates extensively the pitfalls of any simplistic attempts to capture reality as data in the sense of today's database systems. The approach taken by the author is one which very logically and carefully delineates the facets of reality being represented in an information system, and also describes the data processing models used in such systems. The linguistic, semantic, and philosophical problems of describing reality are comprehensively examined… The depth of discussion of these concepts, as they impact on information systems, is not likely to be found elsewhere.… the value of this book resides in its critical, probing approach to the difficulties of modeling reality in typical information systems… it is very well written and should prove both enjoyable and enlightening to a careful reader.

–ACM Computing Reviews, August 1980

By page eight one has been exposed to an incredible number of philosophical ideas, all cast as concrete data-representation problems… this is basically a book that poses problems and exposes contradictions… A very stimulating read.

–Quantitative Sociology Newsletter, Spring 1981

Kent attacks the pseudo-exactness of existing data models in a very neat and clear (and often humorous) manner… This book is for everyone who thinks about or works on data files and who wants to understand the reasons for his disenchantment.

–European Journal of Operations Research, November 1981

I am using *Data and Reality* as research material for my current project. It is on my desk right now.

–Joe Celko, 1998

The book is still quoted quite often and has a message even – or especially – for today's jaded information scientists.

–Prof. Dr. Robert Meersman, Vrije Universiteit Brussel (1998)

Your book focuses attention on many issues that are still, embarrassingly, not being dealt with in our formalized information systems. It provides an important reference point not only in identifying these problems, but in pointing out origins and the long-standing practice of simply ignoring them. When I reopened your book… I found lots of issues that seem as fresh as ever.

–Roger Burkhart, John Deere (1998)

A small number of computing and information management books are of foundational nature, not oriented towards a particular technology, methodology or tool. *Data and Reality* is such a book. The concepts and approach described there are as valid now as they were in 1978, and are still often ignored resulting in systems that are not what we want them to be. Doing better than that requires *Data and Reality* to be an essential component of our intellectual foundation.

–Haim Kilov, Genesis Development Corporation (1999)

I remember my first exposure to the work of Edward Tufte. The richness of detail that could be presented simply was almost a physical shock. Were it not for Bill Kent I might have forgotten that the data represented by that richness was only a representation of reality, and not the reality itself. In a world which reinvents the Perfect Semantic Representation Language to End All Semantic Representation Languages every ten years or so, it is a pleasure to have Bill's calming influence in print in the form of *Data and Reality*.

–Richard Mark Soley, Ph.D., Chairman and CEO,
Object Management Group, Inc. (1999)

Contents

Preface to the Second Edition

Despite critical acclaim, outside of a small circle of enthusiastic readers this book has been a sleeper for over twenty years. Publishers have recently offered to market and distribute it with more vigor if I would provide a new revised edition, but I've resisted. Laziness might be seen as the excuse, but I'm beginning to realize there's a better reason.

A new revised edition would miss the point of the book. Many texts and reference works are available to keep you on the leading edge of data processing technology. That's not what this book is about. This book addresses timeless questions about how we as human beings perceive and process information about the world we operate in, and how we struggle to impose that view on our data processing machines. The concerns at this level are the same whether we use hierarchical, relational, or object-oriented information structures; whether we process data via punched-card machines or interactive graphic interfaces; whether we correspond by paper mail or e-mail; whether we shop from paper-based catalogs or the web. No matter what the technology, these underlying issues have to be understood. Failure to address these issues imperils the success of your application regardless of the tools you are using.

That's not to say the technical matrix of the book is obsolete or antiquated. The data record is still a fundamental component of the way we organize computer information. Sections of the book exploring new models including behavioral elements are precursors of object orientation.

The scope of the book extends beyond computer technology. The questions aren't so much about how we process data as about how we perceive reality, about the constructs and tactics we use to cope with complexity, ambiguity, incomplete information, mismatched viewpoints, and conflicting objectives.

You can read the book for those reasons, or for other reasons as well. A few years back, almost twenty years after the book was published, I began to notice that the book is also about

something else, something far more personal. The scope of the book doesn't only extend beyond computer data processing into the realm of how we perceive the world. It also extends into our inner domain. I've come to recognize that it touches on issues in my own inner life that I, like most of us to some degree or other, have been grappling with for decades.

Consider the key topics: existence, identity, attributes, relationships, behavior, and modeling.

Existence: Is cogito ergo sum *sufficient? To what extent am I really present and engaged in the process of life around me? How real are the physical things I experience? To what extent do I exist in some spiritual realm independent of the physical context?*

Identity: The old "Who am I?" bit. What is the true nature of the kind of person I am? What sorts of needs, goals, outlooks define who I really am?

Attributes: What kind of person am I? What are my values, my assets, my limitations?

Relationships: This is the core of it all. What is the quality of my interaction with parents, lovers, spouses, children, siblings, friends, colleagues, and other acquaintances? What are my connections with things material, social, spiritual, and otherwise? What are my needs here? What are the issues and problems? How can they be improved?

Behavior: What should I plan to do in various situations? How? What might be the consequences, both intended and otherwise? What contingencies need to be anticipated?

Modeling: How accurate and useful are the constructs I use to explain all these things? How effective are these kinds of explanations in helping me change what needs to be changed?

This book certainly shouldn't be classified in the social sciences, but it is remarkable to observe how technology issues can resonate as metaphors for our inner lives. This perspective seems to explain why I've engaged so intimately with these ideas, why I've argued so passionately about them at standards committee meetings and in the hallways at conferences.

I repeat the invitation, made in the book's original preface, to discover for yourself what you might think the book is about.

It just might be about you. But if that's too much pop psychology for your comfort, if that's too invasive of your personal space, then just read it for its insights into data processing and reality.

Preface

A message to mapmakers: highways are not painted red, rivers don't have county lines running down the middle, and you can't see contour lines on a mountain.

For some time now my work has concerned the representation of information in computers. The work has involved such things as file organizations, indexes, hierarchical structures, network structures, relational models, and so on. After a while it dawned on me that these are all just maps, being poor artificial approximations of some real underlying terrain.

These structures give us useful ways to deal with information, but they don't always fit naturally, and sometimes not at all. Like different kinds of maps, each kind of structure has its strengths and weaknesses, serving different purposes, and appealing to different people in different situations. Data structures are artificial formalisms. They differ from information in the same sense that grammars don't describe the language we really use, and formal logical systems don't describe the way we think. "The map is not the territory" [Hayakawa].

What is the territory really like? How can I describe it to you? Any description I give you is just another map. But we do need some language (and I mean natural language) in order to discuss this subject, and to articulate concepts. Such constructs as "entities", "categories", "names", "relationships", and "attributes" seem to be useful. They give us at least one way to organize our perceptions and discussions of information. In a sense, such terms represent the basis of my "data structure", or "model", for perceiving real information. Later chapters discuss these constructs and their central characteristics — especially the difficulties involved in trying to define or apply them precisely.

Along the way, we implicitly suggest a hypothesis (by sheer weight of examples, rather than any kind of proof — such a hypothesis is beyond proof): there is probably no adequate

formal modeling system. Information in its "real" essence is probably too amorphous, too ambiguous, too subjective, too slippery and elusive, to ever be pinned down precisely by the objective and deterministic processes embodied in a computer. (At least in the conventional uses of computers as we see them today; future developments in artificial intelligence may endow these machines with more of our capacity to cope.) This follows a path pointed out by Zemanek, connecting data processing with certain philosophical observations about the real world, especially the aspects of human judgment on which semantics ultimately depend ([Zemanek 72]).

In spite of such difficulties (and because I see no alternative), we also begin to explore the extent and manner in which such constructs can and have been incorporated into various data models. We are looking at real information, as it occurs in the interactions among people, but always with a view toward modeling that information in a computer based system. The questions are these: What is a useful way to perceive information for that purpose? What constructs are useful for organizing the way we think about information? Might those same constructs be employed in a computer based model of the information? How successfully are they reflected in current modeling systems? How badly oversimplified is the view of information in currently used data models? Are there limits to the effectiveness of any system of constructs for modeling information?

In spite of my conjecture about the inherent limits of formal modeling, we do need models in order to go about our business of processing information. So, undaunted, I have assimilated some of my own ideas about a "good" modeling system, and these appear toward the end.

Keep in mind that I am not talking about "information" in a very broad sense. I am not talking about very ambitious information systems. We are not in the domain of artificial intelligence, where the effort is to match the intellectual capabilities of the human mind (reasoning, inference, value judgments, etc.). We are not even trying to process prose text; we are not attempting to understand natural language, analyze

grammar, or retrieve information from documents. We are primarily concerned with that kind of information which is managed in most current files and databases. We are looking at information that occurs in large quantities, is permanently maintained, and has some simplistic structure and format to it. Examples include personnel files, bank records, and inventory records.

Even this modest bit of territory offers ample opportunity for misunderstanding the semantics of the information being represented.

Within these bounds, we focus on describing the information content of some system. The system involved might be one or more files, a database, a system catalog, a data dictionary, or perhaps something else. We are limiting ourselves to the information content of such systems, excluding such concerns as:

- Real implementations, representation techniques, performance.
- Manipulation and use of the data.
- Work flow, transactions, scheduling, message handling.
- Integrity, recovery, security.

A caution to the lay reader in search of a tutorial: this book is not about data processing as it is. As obvious as these concepts may seem, they are not reflected in, or are just dimly understood in, the current state of data processing systems. “We do not, it seems, have a very clear and commonly agreed upon set of notions about data — either what they are, how they should be fed and cared for, or their relation to the design of programming languages and operating systems. This paper sketches a theory of data which may serve to clarify these questions. It is based on a number of old ideas and may, as a result, seem obvious. Be that as it may, some of these old ideas are not common currency in our field, either separately or in combination; it is hoped that rehashing them in a somewhat new form may prove to be at least suggestive” [Mealy]. That opening paragraph of a now classic paper, some ten years old, is still distressingly apt today.

There is a wonderful irony at work here. I may be trying to overcome misconceptions which people outside the computer business don't have in the first place. Many readers will find little new in what I say about the nature of our perceptions of reality. Such readers may well react with "So what's new?" To them, my point is that the computing community has largely lost sight of such truisms. Their relevance to the computing disciplines needs to be re-established.

People in the data processing community have gotten used to viewing things in a highly simplistic way, dictated by the kind of tools they have at their disposal. And this may suggest another wonderful irony. People are awed by the sophistication and complexity of computers, and tend to assume that such things are beyond their comprehension. But that view is entirely backwards! The thing that makes computers so hard to deal with is not their complexity, but their utter simplicity. The first thing that ought to be explained to the general public is that a computer possesses incredibly little ordinary intelligence. The real mystique behind computers is how anybody can manage to get such elaborate behavior out of such a limited set of basic capabilities. The art of computer programming is somewhat like the art of getting an imbecile to play bridge or to fill out his tax returns by himself. It can be done, provided you know how to exploit the imbecile's limited talents, and are willing to have enormous patience with his inability to make the most trivial common sense decisions on his own. Imagine, for example, that he only understood grammatically perfect sentences, and couldn't make the slightest allowance for colloquialisms, or for the normal way people restart sentences in mid-speech, or for the trivial typographical errors which we correct so automatically that we don't even see them. The first step toward understanding computers is an appreciation of their simplicity, not their complexity.

Another thought, though: I may be going off in the wrong direction by focusing so much concern on computers and computer thinking. Many of the concerns about the semantics of data seem relevant to any record keeping facility, whether computerized or not. I wonder why the problems appear to be

aggravated in the environment of a computerized database. Is it sheer magnitude? Perhaps there is just a larger mass of people than before who need to achieve a common understanding of what the data means. Or is it the lost human element? Maybe all those conversations with secretaries and clerks, about where things are and what they mean, are more essential to the system than we've realized. Or is there some other explanation?

The flow of the book generally alternates between two domains, the real world and computers. Chapter 1 is in the world of real information, exploring some enigmas in our concepts of "entities". Chapter 2 briefly visits the realm of computers, dealing with some general characteristics of formally structured information systems. This gives us a general idea of the impact the two domains have on each other. Chapters 3 through 6 then address other aspects of real information. Chapters 7 through 11, dealing with data processing models, bring us back to the computer. We top it all off with a smattering of philosophical observations in Chapter 12.

This has been an approximate characterization—one view—of what the rest of the book contains. Please read on to discover what you might think the book is about.

* * * *

I want to thank the people who took the time to comment on (and often contribute to) earlier versions of this material, including Marilyn Bohl, Ted Codd, Chris Date, Bob Engles, Bob Griffith, Roger Holliday, Lucy Lee, Len Levy, Bill McGee, Paula Newman, and Rich Seidner. George Kent, of the Political Science Dept. at the University of Hawaii, provided a valuable perspective from a vantage point outside of the computing profession. Karen Takle Quinn, our head librarian, was immensely helpful in tracking down many references. I thank Willem Dijkhuis of North Holland for his substantial encouragement in the publication of this book.

And very special thanks go to my wife, Barbara, who helped make the book more readable, and who coped and sacrificed more than anyone else for this book.

1 Entities

"Entities are a state of mind. No two people agree on what the real world view is." [Metaxides]

An information system (e.g., database) is a model of a small, finite subset of the real world. (More or less — we'll come back to that later.) We expect certain correspondences between constructs inside the information system and in the real world. We expect to have one record in the employee file for each person employed by the company. If an employee works in a certain department, we expect to find that department's number in that employee's record.

So, one of the first concepts we have is a correspondence between things inside the information system and things in the real world. Ideally, this would be a one-to-one correspondence, i.e., we could identify a single construct in the information system which represented a single thing in the real world.

Even these simple expectations run into trouble. In the first place, it's not so easy to pin down what construct in the information system will do the representing. It might be a record (whatever that means), or a part of one, or several of them, or a catalog entry, or a subject in a data dictionary, or .... For now let's just call that thing a *representative*, and come back to that topic later. Let's explore instead how well we really understand what it is that we want represented.

As a schoolteacher might say, before we start writing data descriptions let's pause a minute and get our thoughts in order. Before we go charging off to design or use a data structure, let's think about the information we want to represent. Do we have a very clear idea of what that information is like? Do we have a good grasp of the semantic problems involved?

Becoming an expert in data structures is like becoming an expert in sentence structure and grammar. It's not of much value if the thoughts you want to express are all muddled.

The information in the system is part of a communication process among people. There is a flow of ideas from mind to mind; there are translations along the way, from concept to natural languages to formal languages (constructs in the machine system) and back again. An observer of, or participant in, a certain process recognizes that a certain person has become employed by a certain department. The observer causes that fact to be recorded, perhaps in a database, where someone else can later interrogate that recorded fact to get certain ideas out of it. The resemblance between the extracted ideas and the ideas in the original observer's mind does not depend only on the accuracy with which the messages are recorded and transmitted. It also depends heavily on the participants' common understanding of the elementary references to "a certain person", "a certain department", and "is employed by".

1.1 One Thing

What is "one thing"?

That appears at first to be a trivial, irrelevant, irreverent, absurd question. It's not. The question illustrates how deeply ambiguity and misunderstanding are ingrained in the way we think and talk.

Consider those good old workhorse database examples, parts and warehouses. We normally assume a context in which each part has a part number and occurs in various quantities at various warehouses. Notice that: various quantities of one thing. Is it one or many? Obviously, the assumption here is that "part" means one kind of part, of which there may be many physical instances. (The same ambiguity shows up very often in natural usage, when we refer to two physical things as "the same thing" when we mean "the same kind".) It is a perfectly valid and useful point of view in the context of, e.g., an inventory file: we have one representative (record) for each kind of thing, and speak loosely of all occurrences of the thing as collectively being one thing. (We could also approach this by saying that the representative is not meant to correspond to any physical object, but to the

abstracted *idea* of one kind of object. Nonetheless, we do use the term "part", and not "kind of part".)

Now consider another application, a quality control application, also dealing with parts. In this context, "part" means one physical object; each part is subjected to certain tests, and the test data is maintained in a database separately for each part. There is now one representative in the information system for each physical object, many of which may have the same part number.

In order to integrate the databases for the inventory and quality control applications, the people involved need to recognize that there are two different notions of "thing" associated with the concept of "part", and the two views must be reconciled. They will have to work out a convention wherein the information system can deal with two kinds of representatives: one standing for a kind of part, another standing for one physical object.

I hope you're convinced now that we have to go to some depth to deal with the basic semantic problems of data description.

We are dealing with a natural ambiguity of words, which we as human beings resolve in a largely automatic and unconscious way, because we understand the context in which the words are being used. When a data file exists to serve just one application, there is in effect just one context, and users implicitly understand that context; they automatically resolve ambiguities by interpreting words as appropriate for that context. But when files get integrated into a database serving multiple applications, that ambiguity-resolving mechanism is lost. The assumptions appropriate to the context of one application may not fit the contexts of other applications. There are a few basic concepts we have to deal with here:

- Oneness.
- Sameness. When do we say two things are the same, or the same thing? How does change affect identity?

- What is it? In what categories do we perceive the thing to be? What categories do we acknowledge? How well defined are they?

These concepts and questions are tightly intertwined with one another.

Consider "book". If an author has written two books, a bibliographic database will have two representatives. (You may temporarily think of a representative as being a record.) If a lending library has five circulating copies of each, it will have ten representatives in its files. After we recognize the ambiguity we try to carefully adopt a convention using the words "book" and "copy". But it is not natural usage. Would you understand the question "How many copies are there in the library?" when I really want to know how many physical books the library has altogether?

There are other connotations of the word "book" that could interfere with the smooth integration of databases. A "book" may denote something with hard covers, as distinguished from things in soft covers like manuals, periodicals, etc. Thus a manual may be classified as a "book" in one library but not in another. I don't always know whether conference proceedings constitute a "book".

A "book" may denote something bound together as one physical unit. Thus a single long novel may be printed in two physical parts. When we recognize the ambiguity, we sometimes try to avoid it by agreeing to use the term "volume" in a certain way, but we are not always consistent. Sometimes several "volumes" are bound into one physical "book". We now have as plausible perceptions: the *one* book written by an author, the *two* books in the library's title files (Vol. I and Vol. II), and the *ten* books on the shelf of the library which has five copies of everything.

Incidentally, the converse sometimes also happens, as when several novels are published as one physical book (e.g., collected works).

So, once again, if we are going to have a database about books, before we can know what one representative stands for,

we had better have a consensus among all users as to what "one book" is.

Going back now to parts and warehouses, the notion of "warehouse" opens up another kind of ambiguity. There is no natural, intrinsic notion of what constitutes "one warehouse". It may be a single building, or a group of buildings separated by any arbitrary distance. Several warehouses (e.g., belonging to different companies) may occupy the same building, perhaps on different floors. So, what is "one warehouse"? Anything that a certain group of people agrees to call a warehouse. Given two buildings, they might agree to treat them as one, two, or any number of warehouses — with all perceptions being equally "correct".

IBM assigns "building numbers" to its buildings for the routing of internal mail, recording employee locations, and other purposes. One two-story building in Palo Alto, California is "building 046", with the two stories distinguished by suffixes: 046-1 and 046-2. Right next door is another two-story building. The upper story is itself called "building 034", and the lower story is split into two parts called "building 032" and "building 047". IBM didn't invent the situation. The designations correspond to three different postal addresses: 1508, 1510, and 1512 Page Mill Road are all in the same building.

Another IBM location in Santa Teresa, California, is apparently one building, since it has one building number. The structure has eight distinct towers. Signs inside direct you to "building A", "building B", etc. How many buildings are there?

"Street" is another ambiguous term. What is one street? Sometimes the name changes; that is, different segments along the same straight path have different names. Based on a comparison of addresses, we would probably surmise that people on those various segments lived on different streets. On the other hand, different streets in the same town may have the same name. Now what does an address comparison imply?

Sometimes a street is made up of discontinuous segments, perhaps because intervening sections just haven't been built yet. They may not even be on a straight line, because the ultimate street on somebody's master plan curves and wiggles all around.

And sometimes I can make a right turn, then after some distance make a left turn and be back on a street with the same name as the first. Is that one street with a jog? When do we start thinking of these as different streets having the same name?

Is a street terminated by city, county, state, or national boundaries? Suppose the street just ran right across the boundary, same name and all. Would you be inclined to say that people living in different countries lived on the same street?

Does the term "street" imply that motor vehicles can drive on it? Some are narrower than alleys, and some are pedestrian malls.

Does the term "street" include freeways, highways, thruways, expressways, tollways, parkways, autobahns, autopistes, autostradas, autoroutes, dual carriageways, motorways, ... (I'm really just trying to convey one idea — what do they call it in your neighborhood?) Very often, one highway will coincide with portions of many different streets along its route. Does a highway name count as a street name? Along some segments, the highway name might be the only street name. Various street segments will have various multitudes of names ("look at all the highway markers on that pole!"). And, after I make a turn, whether or not I'm on the "same street" may depend on my own state of mind: which street name did I think I was following? Finally: if I drive from New York to California on Highway 66, have I been on the same street all the way?

Thus, the boundaries and extent of "one thing" can be very arbitrarily established. This is even more so when we perform "classification" in an area that has no natural sharp boundaries at all. The set of things that human beings know how to do is infinitely varied, and changes from one human being to another in the most subtle and devious ways. Nonetheless, the "skills" portion of a personnel database asserts a finite number of arbitrary skill categories, with each skill being treated as one discrete thing, i.e., it has one representative. The number and nature of these skills is very arbitrary (i.e., they do not correspond to natural, intrinsic boundaries in the real world), and they are likely to be different in different databases. Thus, a "thing" here is a very arbitrary segment partitioned out of a

continuum. This applies also to the set of subjects in a library file or information retrieval system, to the set of diseases in a medical database, to colors, etc.

This classification problem underlies the general ambiguity of words. The set of concepts we try to communicate about is infinite (and non-denumerable in the most mind-boggling sense), whereas we communicate using an essentially finite set of words. (For this discussion, it suffices just to think about nouns.) Thus, a word does not correspond to a single concept, but to a cluster of more or less related concepts. Very often, the use of a word to denote two different ideas in this cluster can get us into trouble.

A case in point is the word "well" as used in the data files of an oil company. In their geological database, a "well" is a single hole drilled in the surface of the earth, whether or not it produces oil. In the production database, a "well" is one or more holes covered by one piece of equipment, which has tapped into a pool of oil. The oil company had trouble integrating these databases to support a new application: the correlation of well productivity with geological characteristics.

1.2 How Many Things Is It?

A single physical unit often functions in several roles, each of which is to be represented as a separate thing in the information system. Consider a database maintaining scoring statistics for a soccer team, both on a position basis and on an individual basis. The database might have representatives for 36 things: 11 positions and 25 players. When Joe Smith, playing halfback, scores a goal, the data about two things is modified: the number of goals by Joe Smith, and the number of goals by a halfback. That human figure standing on the field is represented as (and is) two things: Joe Smith and a halfback.

Consider the question of "sameness". Suppose Joe switches to fullback, and scores another goal. Did the same thing make those two goals? Yes: Joe Smith made both. No: one was made by a halfback, the other by a fullback.

Why is that human figure perceived and treated as two things, rather than one or three or ninety-eight? Not by any

natural law, but by the arbitrary decision of some human beings, because the perception was useful to them, and corresponded to the kinds of information they were interested in maintaining in the system.

If the file only had data about player positions, then the same physical object would be treated as being different things at different times. Joe is sometimes a halfback and sometimes a fullback. From the perspective of this file, his activities are being performed by two different entities.

Also consider two related people (e.g., husband and wife) who work for the same company. When considering medical benefits, each of these people has to be considered twice: once as an employee, and once as a dependent of an employee. How many people are involved?

Or suppose a person held two jobs with the company, on two different shifts. Does that signify one or two employees? Shipping clerk John Jones and third-shift computer operator John Jones might be the same person. Does it matter? Sometimes.

The notion is also applicable to warehouses. From the point of view of another application, the thing involved is not a warehouse at all, but a building or property on the assessment rolls.

It is plausible (bizarre, perhaps, but plausible) to view a certain employee and a certain stockholder as two different things, between which there happens to exist the *relationship* that they are embodied in the same person. There would then exist two representatives in the system, one for the employee and one for the stockholder. It's perfectly all right, so long as users understand the implications of this convention (e.g., deleting one might not delete the other).

Transportation schedules and vehicles offer other examples of ambiguities, in the use of such terms as "flight" and "plane" (even if we ignore the other definitions of "plane" having nothing to do with flying machines). What does "catching the same plane every Friday" really mean? It may or may not be the same physical airplane. But if a mechanic is scheduled to service the same plane every Friday, it had better be the same physical airplane. And another thing: if two passengers board a plane

together in San Francisco, with one holding a ticket to New York and the other a ticket to Amsterdam, are they on the same flight?

Classification, e.g., of skills, impacts the notion of "sameness" as much as the notion of "how many". The way we partition skills determines both how many different things we recognize in this category, and when we will judge two things to be the same. Consider a group of people who know how to do such things as paint signs on doors, paint portraits, paint houses, draw building blueprints, draw wiring diagrams, etc. One classifier might judge that there is just *one* skill represented by all of these capabilities, namely "artist", and that every person in this group had the *same* skill. Another classifier might claim there are *two* skills here, namely painting and drawing. Then the sign painter has the same skill as the portrait painter, but not the blueprint drawer. And so on.

The same game can be played with colors. Two red things are the same color. What if one is crimson and the other scarlet?

The perceptive reader will have noticed that two kinds of "how many" questions have been intermixed in this section. At first we were exploring how many *kinds* of things something might be perceived to be. But occasionally we were trying to determine whether we were dealing with one or several things of a given kind. If you can't apply that distinction to the preceding discussions, then please don't become a database administrator. I fear your database may well become a minefield of semantic traps.

For another example of the latter kind, consider program problem reports (known as APAR's in IBM). Considerable effort is often expended in determining that the symptoms reported in two APAR's are caused by the same programming error; thereafter, the two APAR's are considered to be the "same". (The correctness of this view depends on whether you think the entity involved is the programming error or the problem report.)

And analogously, much of the fuss in many insurance claims and court battles revolves around determining whether several things relate to the "same" illness or injury.

1.3 Change

And then there's change. Even after consensus has been reached on what things are to be represented in the information system, the impact of change must be considered. How much change can something undergo and still be the "same thing"? At what point is it appropriate to introduce a new representative into the system, because change has transformed something into a new and different thing?

The problem is one of identifying or discovering some essential invariant characteristic of a thing, which gives it its identity. That invariant characteristic is often hard to identify, or may not exist at all.

We seem to have little difficulty with the concept of "one person" despite changes in appearance, personality, capabilities, and, above all, chemical composition. (The proportions and structure — i.e., the chemical formulas — may not change much, but the individual atoms and molecules are continually being replaced... again illustrating an ambiguity between "same kind" and "same instance": how rapidly is the chemical composition of your body changing?) When we speak of the same person over a period of time, we certainly are not referring to the same ensemble of atoms and molecules. What then is the "same person"? We can only appeal to some vague intuition about the "continuity" of — something — through gradual change. The concept of "same person" is so familiar and obvious that it is absolutely irritating not to be able to define it. Definitions in terms of "soul" and "spirit" may be the only true and humanistic concepts, but, significantly, we don't know how to deal with them in a computer-based information system. It is only when the notion of "person" is pushed to some limit do we realize how imprecise the notion is. This is the basis of some legal issues.

Modern medicine is dissecting our concept of "person" via transplanted and artificial limbs and organs. The Hopi Indians consider mental activity to be in the heart [Whorf]; they might argue that the recipient of a heart transplant becomes the person who the donor was — the donor has merely acquired a new

body. (Is it a heart transplant or a body transplant?) We are more likely to take that position with respect to the brain, rather than the heart. A number of legal issues will have to be resolved when brain transplants begin to be performed (and the issues may get more complex if just portions of the brain are transplanted).

In an information system maintaining data about people, we will have to decide which information gets interchanged between two representatives. Which information is to be associated with the body, and which with the brain? A name? A spouse? Other relatives? How is the medical history rearranged? Who has which job? Skills? Financial obligations?

We also have some issues regarding the beginning and ending of a person. It makes sense in the context of some medical records to treat an unborn fetus as an unborn person; observations during pregnancy become a part of that person's medical history. A recent court case considered the question of whether an unborn fetus was eligible for welfare benefits, which would have made the fetus representable in the welfare database. After death, a person ceases to exist for many legal purposes, but the data about him (or his body) continues to be relevant to a cemetery, or a coroner, or a medical researcher.

An analogous situation exists with automobiles. Suppose you and I start trading parts of our cars — tires, wheels, transmissions, suspensions, etc. At some point we will have exchanged cars, in the sense that the Department of Motor Vehicles must change their records as to who owns which car — but when? What is the "thing" which used to be my car, and when did you acquire it? The Department of Motor Vehicles (at least in California, I believe) has made an arbitrary decision: the "essence" of a car is the engine block, which is (they assume) indivisible and is uniquely numbered. Owning and registering a car is defined to mean owning and registering the engine block. All the other parts of the car can be removed or replaced without altering the identity of the car.

What would happen if another state had a different convention for establishing the identity of a car? Could their two databases be integrated?

The same kinds of questions apply to organizations, such as companies, departments, teams, government agencies, etc. Is it still the same company after changes in employees? (Of course.) Management? (Yes.) Owners? (Maybe.) Buildings and facilities? (Yes.) Locations? (Probably.) Name? (Probably). Principal business? (Maybe.) State and country of incorporation? (Maybe.) The answers are significant to the handling of old contracts and other obligations, the determination of employee vacation and retirement benefits, etc.

And political boundaries. A database of population statistics must have some definition of what is meant by India, Pakistan, Germany, Czechoslovakia, etc., over time. There's more involved than a change of name; the things themselves have been created, destroyed, merged, split, re-partitioned, etc. In some other database it may have to be understood that two people born at different times in the same town might have been born in different countries.

There are some kinds of change which result in the existence of two copies of the thing, corresponding to the states before and after the change. There are several ways to deal with this situation: (1) Discard the old and let the new replace it, so that it is really treated as a change and not as a new thing; (2) Treat the old and the new as two clearly distinct things; and (3) Try to do both.

The significance of differences between copies shows up in books and other textual matter. The document you are reading now is one book. It has been and will be the "same book" throughout a series of changes, and may even appear published in several forms with various changes in wording, punctuation, etc.

A whole spectrum of concepts. There is the "one book" containing the ideas expressed by an author, which is the same book regardless of which language it is translated into, or how it is edited, abridged, condensed, revised, etc.

Then there are "editions", which differ from each other by some arbitrary amount, due either to changes in the content or to the correction of significant amounts of error. On the other hand,

some minor amount of difference (erroneous or deliberate) is permitted between reprints of a single edition.

A condensation or abridgment may be grossly different from the original, but for some purposes it is treated as being the same book.

This topic is most painfully familiar to us in relation to "versions", e.g., of such things as programs. There is some arbitrary threshold up to which minor changes can be made without creating a new version. The old copy is discarded, there may or may not be a record of the modification, and the representative (e.g., catalog entry) of the old copy now serves to represent the new copy.

Beyond a certain (arbitrary) point, we decide to keep the old and new copies as different versions. We now enter a metaphysical realm in which we manage to merge the concepts of "one" and "many", as in the expression "these *several* things are different versions of the *same* thing". In some contexts we mean to refer to all versions collectively (as in the property: this is a FORTRAN compiler), in some we refer to a particular copy, and in some we refer to one copy — whichever one happens to be the "current" version.

A user who invokes the FORTRAN compiler several times probably believes that he is invoking the "same thing" each time even if he gets different versions. From this point of view, there should be one representative for this thing ("the current version") even though it represents different things at different times. Each version should also have its own permanent representative, and there probably should also be one representative for the collective concept of "FORTRAN compiler" independent of version. The representatives for the current copy and the collective concept may or may not be the same; is the property "required memory size" applicable to both?

1.4 The Murderer and the Butler

Combining the ideas of our last two sections: sometimes it is our perception of "how many" which changes. Sometimes two distinct entities are eventually determined to be the same one,

perhaps after we have accumulated substantial amounts of information about each.

At the beginning of a mystery, we need to think of the murderer and the butler as two distinct entities, collecting information about each of them separately. After we discover that “the butler did it”, have we established that they are “the same entity”? Shall we require the modeling system to collapse their two representatives into one? I don’t know of any modeling system which can cope with that adequately.

1.5 Categories (What Is It?)

We have so far been focusing on the questions of “oneness” and “sameness”. That is, given that you and I are pointing to some common point in space (or we think we are), and we both perceive something occupying that space (perhaps a human figure), how many “things” should that be treated as in the information system? One? Many? Part of a larger thing? Or not a thing at all?

And: do we really agree on the composition and boundary of the thing? Maybe you were pointing at a brick, and I was pointing at a wall.

And: if we point to that same point in space tomorrow (or think we are), will we agree on whether or not we are pointing at the same thing as we did today?

None of this focuses on what the thing *is*. I don’t mean its properties, like is it solid, or is it red, or how much does it weigh, but what *is* it? I had to use the phrase “human figure” above because I didn’t think you would follow my point if I kept using the indefinite word “thing” — I had to convey some kind of tangible example. But that phrase is just one possible perception of the “thing” we pointed to. You might have said it was a mammal, or a man, or a solid object, or a bus driver, or your father, or a stockholder, or a customer, or ... ad nauseam.

I will refer to what a thing is — or at least what it is described to be in the information system — as its “category”, agreeing with the usage in, e.g., [Abrial]. The same idea is also often called “type”, or “entity type”. Like everything else, the

treatment of categories requires a number of arbitrary decisions to be made.

There is no natural set of categories. The set of categories to be maintained in an information system must be specified for that system. In one system it might be employees and customers, in another it might be employees and dependents, or enrolled computer users, or plaintiffs and defendants, and in an integrated database it might include all of these. A given thing (representative) might belong to many such categories.

Not only are there different kinds of categories, but categories may be defined at different levels of refinement. One application might perceive savings accounts and loan accounts as two categories, while another perceives the single category of accounts, with "savings" or "loan" being a property of each account. In another case, we might have applications dealing with furniture or trucks or machines, while another deals with capital equipment (assigning everything a unique inventory number). Thus, some categories are, by definition, subsets of others, making a member of one category automatically a member of another. Some categories overlap without being subsets. For example, the category of customers (or of plaintiffs, in a legal database), might include some people, some corporations or other businesses, and some government agencies.

It is often a matter of choice whether a piece of information is to be treated as a category, an attribute, or a relationship. (Which raises the question of how fundamental such a distinction really is.) This corresponds to the equivalence between "that is a parent" (the entities are parents), "that person has children" (the entities are people, with the attribute of having children), and "that person is the parent of those children" (the entities are people and children, related by parentage).

It's often difficult to determine whether or not a thing belongs in a certain category. Almost all non-trivial categories have fuzzy boundaries. That is, we can usually think of some object whose membership in the category is debatable. Then either the object is arbitrarily categorized by some individual, or else there are some locally defined classification rules which probably don't match the rules used in another information

system. Just as an example, consider the simple and "well understood" category of "employee". Does it include part-time employees? Contract employees? Employees of subsidiary companies? Former employees? Retired employees? Employees on leave? On military leave? Someone who has just accepted an offer? Signed a contract but not yet reported for work? Not only do the answers have to be decided according to how the company wants to treat the data, but perhaps the questions can't even be answered consistently within the company. A person on leave may not be an employee for payroll purposes, although he is for benefits purposes. Then the notions of category and property have to be reexamined again, to arrive at a set meaningful to all users.

As another example, consider the category of "cars", and decide if the following are included: station wagons, micro-buses, ordinary buses, pickup trucks, ordinary trucks, motor homes, dune buggies, racing carts, motorcycles, etc. What about a home-made contrivance in which a short pickup truck bed is hung out of the trunk of a sedan? An old bus converted to a motor home?

As long as we are traveling, answer this question: what's the difference between a motel and a hotel? (If you have an answer, you haven't traveled much lately.)

"A more amusing example is to imagine a continuum of physical objects between some given chair and table, constructed by letting the chair back shrink while its seat expands and flattens, and its legs become higher. There will be some strange objects in this continuum which cannot clearly be assigned to either class" [Goguen]. Does the distinction between a bench and a table depend on your height?

The editor of a collection is often listed as the "author" of the book. Did he "author" anything?

The category of a thing (i.e., what it is) might be determined by its position, or environment, or use, rather than by its intrinsic form and composition. In the set of plastic letters my son plays with, there is an object that might be an "N" or a "Z", depending on how he holds it. Another one could be a "u" or an "n", and still another might be "b", "p", "d", or "q".

The purposes of the person using an object very often determine what that object is perceived to be (cf. [Stamper 77]). I can imagine the same hollow metal tube being called a pipe, an axle, a lamp pole, a clothes rack, a mop handle, a shower curtain rod, and how many more can you name? A nail driven into a wall might be designated a coat hook.

You may think you are carrying the inventory file under your arm. But the customs agent perceives a quantity of magnetic tape, and randomly snips off a sample.

Now consider some physical objects. One is a vertical rod mounted on the center of a circular stone. The second is a set of metal pointers driven around a common axis by a system of mechanical gears. The third is a marked cylinder of paraffin, with a burning cotton core. The fourth has two chambers, with a fluid flowing between them. The fifth is a flashing digital display driven by solid state circuitry. Are these all the same kind of object? Yes — if you happen to perceive them as clocks.

"Basically, we're all trying to say the same thing."

On the other hand, is a watch a clock? Of course it is — but try asking someone if he has a clock with him.

In part, these observations illustrate the difficulty of distinguishing between the category (essence) of a thing and the uses to which it may be put (its roles).

There are also interesting questions having to do with fragments of things, and imitations. Is it still a donut after you've taken a bite out of it? Did you ever call a stuffed toy an animal?

And, like everything else, the category of an object can change with time. A dependent becomes an employee, and then a customer, and then a stockholder. A slab of marble becomes a

sculpture. A piece of driftwood becomes a work of art — just by being found and labeled! An ingot of steel becomes a machined part.

The number of entities changes, too. One ingot becomes many parts. Cutting a work of art in pieces may be vandalism — or it may create many works of art.

Perhaps the easiest way out is to ignore the principles of continuity and conservation that we have learned since earliest childhood. It simply is no longer the same object. The sculptor does not "modify" the marble. He destroys the slab, and creates a sculpture.

The fundamental problem of this book is self describing. Just as it is difficult to partition a subject like personnel data into neat categories, so also is it difficult to partition a subject like "information" into neat categories like "categories", "entities", and "relationships". Nevertheless, in both cases, it's much harder to deal with the subject if we don't attempt some such partitioning.

For a closing amusement, do you remember "Who's On First"? Well, here's a variation:

"Which is bigger, a baseball team or a football team?"

"A football team, of course."

"Why's that?"

"A football team has eleven players, and a baseball team has nine."

"Name a baseball team."

"The San Francisco Giants."

"How many players do they have?"

"About twenty five."

"I thought you said a baseball team has nine players."

"I guess it's twenty five."

"Any twenty five baseball players?"

"No, just the twenty five on one roster."

"If they trade a player, does that change the team?"

"Of course."

"You mean they're not the San Francisco Giants any more?"

And so on.

1.6 Existence

In a record processing system, records are created and destroyed, and we can decide with some certainty whether or not a given record exists at any moment in time. But what can we say about the existence of whatever entities may be represented by such a record?

1.6.1 How Real?

It is often said that a database models some portion of the real world. I've said so in this book.

It ain't necessarily so. The world being modeled may have no real existence.

It might be historical information (it's not real now). We can debate whether past events have any real existence in the present.

It might be falsified history (it never was real) or falsified current information (it isn't real now). Fraudulent data in welfare files: is that a model of the "real" world?

It might be planning information, about intended states of affairs (it isn't real yet).

It might be hypothetical conjectures — "what if" speculations (which may never become real).

One might argue that such worlds have a Platonic, idealistic reality, having a real existence in the minds of men in the same way as all other concepts. But quite often the information is so complex that no one human being comprehends all of it in his mind. It is not perceived in its entirety by any agency outside of the database itself. Or, although not overly complex, the information may simply not have reached any human mind just yet. The computer might have performed some computations to establish and record some consequence of the known facts, which no person happens to be aware of yet. It happens all the time: computers often record accounts as being overdrawn some time before any people are told about it. And even more obviously: that is precisely the point of doing hypothetical

simulations by computer. The computer figures out who wins a simulated war game; in the interval between the computation and a person's reading of the output, this result is in the computer — but what person "knows" it?

Where is the reality that the database is modeling?

And what about fiction? The subjects of some databases are the people, places, and events occurring in fiction (literature, mythology). This again stretches the concept of the "real world" being modeled in a database. (Isn't fiction the opposite of reality?) But beyond that, it challenges certain premises about certain kinds of entities.

It is sometimes held that there are certain "intrinsic attributes" which all entities of a certain type must possess. For people, such attributes include birthdate, birthplace, parents, height, weight, etc. Does Hamlet have these attributes? Cities have a geographic location, an area, a population, etc. Does Camelot have these attributes?

Or shall we say instead that Hamlet is not a person, and Camelot is not a city?

Note that this situation is very different from a simple lack of information. It is not uncommon to say that we don't know a certain person's birthday, and to record it as "unknown" in the database. That implies the possibility of eventually discovering and recording what it is. Instead, we are questioning whether such characteristics exist at all.

To conclude, if we can't assert that a database models a portion of reality, what shall we say that a database does in general? It probably doesn't matter. Once again, it seems that we can go about our business quite successfully without being able to define (or know) precisely what we are doing.

If we really did want to define what a database modeled, we'd have to start thinking in terms of mental reality rather than physical reality. Most things are in the database because they "exist" in people's minds, without having any "objective" existence. (Which means we very much have to deal with their existing differently in different people's minds.) And, of the things in the database which don't exist in any person's mind,

whose mental reality is that? Shall we say that the computer has a mental reality of its own?

1.6.2 How Long?

Some kinds of entities have a natural starting and ending, and others have an "eternal" existence; creation and destruction aren't relevant concepts for them. The latter tends to be true of what we call "concepts" — numbers, dates, colors, distances, masses.

We could be perverse and wonder in what sort of Platonic sense such concepts have "always" existed. Did zero exist before some ancient Arab thought of it? Did gravity exist before Newton? Did the concept of television exist 50 years ago?

It doesn't really matter, for our purposes. We are not going to have to worry about creating and destroying such conceptual entities. Unless.... you are a cosmetic company, "inventing" new colors every day... or a number theorist, computing certain numbers (e.g., the primes, or perfect numbers), and adding each one to a list as you "discover" it.

There are, at the other extreme, tangible physical objects that have a well defined finite period of existence, a beginning and an end. Creation and destruction are very relevant concepts here.

But notice that I hesitate to list examples. Beginnings and endings are often processes, rather than instantaneous events. We get tied up in our definitions of what entities are in the first place. Is it the whole thing when it's partially formed? The whole abortion controversy centers on this: does a person become a person at conception, or birth, or somewhere in between? Does a car stop being a car when it enters the junkyard? Or after it's been deformed into a solid cube?

The entity concept enters in some other ways, too. Depending on what entity categories we choose, a certain process may or may not create an entity. Hiring merely alters the attributes of a person, but it creates an employee (but be careful — it might be a re-hire!). And, did the sculpture always exist in the marble? Recall the old vaudeville directions for sculpting an elephant: just cut away the parts that don't look like an elephant.

In spite of all of this, we can entertain a notion that tangible objects have a finite existence, a beginning and an ending.

Not that we always really care. For most of our practical purposes, we prefer to treat certain objects as eternal, those whose "finite" existences appear virtually infinite: the continents, the planets, the sun, the stars. The creation and destruction of these are real only to astronomers, and to science fiction fans (real???).

But suppose that we had neatly defined tangible objects, with instantaneous beginnings and endings. Does that solve all the important problems?

We are, of course, not interested primarily in the objects themselves, but in the information we have about them. Does our handling of this information mimic the creation and destruction of such objects? Do we start having information about such objects at the instant of their creation, and stop having the information at the instant of their destruction? Of course not. We often become aware of things long after their creation (the people we deal with, the things we buy). And we're sometimes aware of them before their creation. Data are kept about children before their birth. Unborn — and unconceived — children are mentioned in wills. Data may kept about ordered merchandise long before manufacture begins.

And we certainly keep information about things long after they have ceased to exist.

So, does the creation and destruction of information have any direct relationship to the beginning and ending of objects? Almost never. "Create" and "destroy", when applied to information, really instruct the system to "perceive" and "forget".

Once more: we are not modeling reality, but the way information about reality is processed, by people.

2 The Nature of an Information System

For the most part we are looking at the nature of information in the real world. But our ultimate motivation is to formulate descriptions of this information so that it may be processed by computers. In this chapter we briefly explore how this goal shapes our view of information. Among other things, we touch on the need for having data descriptions.

At a fundamental level, there are certain characteristics of computers that have a deep philosophical impact on what we do with them. Computers are deterministic, structured, simplistic, repetitious, unimaginative, unsympathetic, uncreative. These notions I leave as background; that's a different plane from the one I want to be on. (Some may argue with those characterizations. Some artificial intelligence experiments have simulated more elegant computer behavior. But it remains an adequate description of the computers that will be processing our data in the near term.)

We take "information system" to be more or less synonymous with the term "integrated database". We mean to deal only with information that can be perceived as some formal structure of relatively simple field values (as in computerized file or catalog processing). We thus exclude, for example, text based systems, with their capabilities for parsing, abstracting, indexing, and retrieving from natural language text.

2.1 Organization

A computer is typically described as consisting of input, processing, output, and memory. I will change the words slightly, and suggest that we need to think of three basic parts of a data processing system: a repository, an interface, and a processor.

2.1.1 Repository

The repository "contains" information, in some static sense. We have to have some mental system for imagining what is inside that repository. That's what this book is mostly about. Whether we think in terms of words written and erased on blackboards or beads resting on an abacus, we have to have some mental concept. For some people and some purposes, the right image involves magnetic fields and silicon; for others the image is in terms of files of punched cards.

I want to suggest that we try to adopt an image more in terms of the informational functions performed for us, rather than in terms of the mechanical processes and materials that perform those functions. By way of analogy, I would say that we should think of a clock as containing a "repeater", an abstract mechanism capable of actuating something at precise and uniform time intervals. We would say nothing about gears and escapements, or silicon circuits, or pivoted and balanced water pipes.

2.1.2 Interface

The interface is the medium of communication between you and the repository, or, more precisely, between you and the processor. It may actually consist of punched cards and readers, printers and printed forms, typewriter terminals, graphic displays, etc. etc. For our purposes, we need only imagine it as an opaque surface with a stream of symbols passing in and out of it.

2.1.3 Processor

The processor receives symbol streams coming in across the interface. Parts of the stream are instructions to the processor, e.g., to change information or to find answers to questions. Parts of the stream represent information which is to be put into the

repository, or which is used to find things in the repository (e.g., the name of the person about which you want information).

The processor, following instructions, alters or retrieves information in the repository. It then generates an outgoing stream across the interface, containing either requested information or status about the operations.

2.2 Data Description

2.2.1 Purpose

In a totally generalized system, there might be a universal naming convention uniformly applicable to all things. For example, one might postulate that a name is any string of characters, of unlimited length; every thing has one or more such names (if several, the names are interchangeable and synonymous). Conventional systems don't support such generality, and we rarely want it. In most cases, there are restrictions on the kinds of names that are acceptable. There may be limits on length (perhaps a certain fixed length for a certain kind of thing), and restrictions on acceptable characters and syntax (only digits, only letters, must start with a letter, hyphens in certain positions, rules about blanks and commas and periods, etc.). A thing often has different kinds of names, which are not synonymous and interchangeable (social security number and employee number; license number and engine number). To enforce such constraints, we have to notify the information system, in advance, which naming conventions will apply to which kinds of things (to employees, departments, parts, warehouses, cities, cars, etc.).

Similarly, an information system might be totally permissive, imposing no constraints at all on the semantic sensibility of information. The system would accept such information as "the accounting department has a shipping weight of 30 pounds, and has two children named 999-1234 and 12.50". While it is possible to build such totally generalized systems, it is customary, in all current data processing systems, to exclude

such absurdities. Provision is needed to specify which things can sensibly have which properties, and which relationships make sense between which things.

Pre-definition of information is also needed in order to specify security constraints, to specify validity criteria for information, and to specify how representations are to be interpreted (data type, scale, units, etc.).

There are also economic implications. Known limitations on the lengths of various information, and a predictability of which pieces of information will or won't occur together, make it possible to plan much more efficient utilization of computer storage. In fact, if the constraints are strict enough, very efficient repetitions of simple patterns can be employed. Furthermore, if formats are rigid enough, and the number of combinations of things that might occur is limited, then programs and procedures can be kept simple and efficient. This is precisely why data processing is currently done in terms of records.

Such rules and descriptions should be assertable before information is loaded into the system, and obviously can't be expressed in terms of individuals. ("Tom, Dick, and Harry must have 6-digit employee numbers.")

At the semantic level, we have adopted (in section 1.5.) the term "category" to label the intrinsic character of a thing ("man or mouse"). It also offers an attractive way of specifying rules about things without referring to the individual things. One simply asserts that certain rules apply to all things in a certain category; one only has to name the category, not the individuals.

Categories are at the foundation of almost all approaches to the description of data, and we will also adopt such an approach for the time being. But we will have some critical things to say about it later.

2.2.2 Levels of Description

Real systems are not so monolithically simple as our idealized system organization of interface, processor, and repository. A single system typically supports a myriad of interfaces and processors, each with its own capabilities,

protocols, and languages. A large variety of programs are developed to serve a variety of application areas. For our purposes, it doesn't matter whether we think of such programs as interface or processor.

The various people and applications using a database are likely to have different perceptions of the entities and information they are dealing with (employees vs. stockholders; employer implied by record type vs. employer as a field value). Different applications use different facts about entities, so that an employee record may look quite different in the personnel application and in the medical benefits application. It is also possible for these applications to use different data processing disciplines, i.e., different file types, access methods, and data structures. These generally provide different ways of representing relationships and different interfaces for manipulating the data.

Thus there is a level of description corresponding to the perceptions and expectations of various applications, specifying such things as record formats, data structures, and access methods. For some kinds of question answering systems, or systems with graphical displays, the descriptions might not even be couched in terms of record formats.

All these applications may be supported by a common pool of data, an integrated database. One significance of integration is that common attributes are synchronized; e.g., changing an employee's address also changes his address in the stockholder file, if he happens to be one. Synchronization may be achieved by maintaining the address in only one place, or by the system's recognizing that a change in one place must automatically be propagated to another place. The method doesn't matter, as long as the information appears synchronized to users.

Another significance of integration is that a new application may "borrow" data already in the database for the benefit of other applications. The new application's requirements can be mapped directly to the integrated database. Without integration, it can be difficult and often impossible to extract the data from several physically unrelated files and then merge it into a form useful to the new application.

The integrated database is the system's analog to the real world: it is that ongoing persistent thing of which different applications may have different perceptions.

Unlike the real world, however, we don't have the luxury of merely saying "it's there — make of it what you will, with your own eyes and ears and mind". The database has to be described to the system.

We have a choice of describing the integrated database in "physical" terms, or in both "physical" and "logical" terms. Physical descriptions specify the location, format, and organization of the data on disks, tapes, or other storage media; the locations of key fields in records; the kinds of pointers used to reference related records; the criteria for physical contiguity of records, and the handling of "overflow" records; the kinds of indexes provided, and their locations; etc. Logical descriptions are more in terms of the information content of the database: the kinds of entities, the attributes, and the relationships among them.

We have identified three levels of description:

- The multiple views held by a variety of applications, each employing their own variations on record formats, structures, and access techniques. This level is variously referred to as "user", "application", "external", "program", and "logical".
- The physical layout of data in storage, including implementation techniques for various paths and linkages. The common names for this level are "internal", "storage", and "physical".
- The specification of the information content of the database, employing concepts equivalent to entities, attributes, and relationships. Names for this level include "conceptual", "information", and "entity" (and, sometimes, "logical").

There is growing recognition of a need to provide and maintain these three levels of description ([ANSI], [GUIDE-SHARE]).

This separation into multiple levels of descriptions is necessary to cope with change. Experience has shown that the way data is used changes with time. Application programs change the way they use the data. They change record formats, and they change the combinations of records they need to see in a single process. New applications need to see records containing data that had previously been split among several records. Other new applications need extensions to existing data (e.g., additional fields in old records), without perturbing the old applications. Applications sometimes change the data management technique which they use to access the data.

Below the interface seen by these applications, the physical layout of the data changes. The grouping and sequencing of data, redundancy, and various kinds of access paths combine to provide certain performance tradeoffs for the various applications. Such parameters are often "tuned" to vary these performance characteristics, and such tuning is not supposed to affect the logical operations of the applications.

As an increasing number of applications interact with an increasingly large integrated database, the effects of such changes become much more complex, more difficult to predict and control. Davies described the problem some ten years ago:

"...nothing stands still or, conversely, everything is subject to change and does... When IBM changed from five to six character man numbers, all programs referring to a data set containing man numbers had to be recompiled. In addition, all data sets in active use had to be copied to change their formats. This was by no means the end of the work. What about the data sets containing man numbers that had become history? The man number field most certainly did not change length in these data sets. Now we have two definitions of the field to contend with forever, unless of course all history is copied and the format changed to reflect the latest definition of the field wherever it appears. In the case of applications like inventory control for fifty thousand items or so, this would have been in excess of two thousand reels of tape!

"This is still not the end. All programs referring to history would have to be recompiled. Another example of the kind of change that is punitive to current application programs is the

case of the field that changes from one data set to another. It must be understood that a data set represents that collection of fields most frequently needed the majority of the time. While this, of course, is absolutely true, frequency of access changes with time.

"Unfortunately, a program written today that did not require a certain field may, when changed tomorrow, require that field. If we physically move the field from one data set to another, although we satisfy the program requiring the field, we would have rendered useless the programs previously referencing the original data set from which we removed the field. These programs could not be recompiled to recover our losses but rather their logic would have to be changed in order to address the field, because it is now in a different physical data set.

"The preceding are but two occurrences of changes to a data set that did not affect the logic of most programs using the data set. The author has been a witness to no less than five hundred such changes, affecting over fifteen hundred data sets and five thousand programs. The logic was not being changed. It is frightening to say the least, but worse, it reduces the already too small qualified programming staffs to fire fighters instead of allowing them to enlarge the application" [Davies].

A need is emerging to manage the data in a manner that is insensitive to such changes. A new role is emerging — the Database Administrator. A large part of his job consists of defining and managing this mass of information as a corporate resource ([ANSI] splits out this part of the job into the role of "enterprise administrator"). He needs a way to describe this information purely in terms of "what kinds of information do we maintain in the system". With this description (the conceptual model) as a reference, he can then separately specify the various formats in which this data is to be made available to application processes (the external models), and also the physical organizations in which the data is to exist in the machine (the internal model).

Besides its role in an operational database system, a conceptual model is also needed in the planning process. It provides the basic vocabulary, or notation, with which to collect

the information requirements of various parts of the enterprise. It provides the constructs for examining the interdependencies and redundancies in the requirements, and for planning the information content of the database.

This book is essentially concerned with the conceptual model, i.e., the descriptions of the information content of the database. It reflects a perception of reality held by one person or group, in the role of database administrator. This administrator decides what portion of the real world is to be reflected in the database, and which constructs, conventions, models, assumptions, etc., are to be used. Although it is a single perception of reality, it must be broad and universal enough to be transformable into the perceptions of all the applications supported by the database.

2.2.3 The Traditional Separation of Descriptions and Data

In traditional record processing systems, constraints on information are implicitly enforced by the rigid discipline of record formats. The birthday of a car cannot be recorded anywhere, if there is no defined record format putting two such fields into one record. Alphabetic employee numbers are excluded by specifying the data type of the field as numeric; the defined field length determines the length of acceptable employee numbers.

Out of this practice emerged a “type” concept, referring to record formats. A set of records of type X all conform to the described format for type X records. And the systems require record descriptions. (But the level of discipline varies considerably. A system might only require specification of the length of a record, to know how to fit it into available storage. In such a minimal system, all kinds of junk might still be crammed into a record.)

Such practices have two consequences in data processing systems: the emergence of a type concept, and the partitioning of the repository into two disjoint parts (often with distinct interfaces and processors): one for the data, and one for the descriptions.

We have thus far described a very simple system organization, consisting of interface, processor, and repository. The descriptions, and constraints, have to be somewhere.

Historically, the repository has been partitioned into two unconnected regions, the data files and the descriptions (contained in system catalogs or data dictionaries). The descriptions had to be formatted specially because system code looked at it, and it had to do that efficiently. Also the system had to be protected from anyone tampering with the descriptions, as they might with ordinary data, because the system fell apart if the descriptions didn't match the data. If a file description is changed, that's more than just a change in information — the system has to do something, usually traumatic, to make the file fit the new description.

These concerns far overshadowed any possible need by any application to get at data that happened to be in the catalog or dictionary. The separation is so entrenched in the thinking of most data processing people that they don't even understand what I'm talking about. Catalogs and files are so "obviously" different things that they can't fathom any commonality. One of them is encoded information used by the system, and the other is the data used by applications.

There are some parallels between the two. Files and catalogs both contain descriptions (or data) about things. We can easily switch the traditional uses of some terms: an employee file contains *descriptions* of employees, while catalogs contain *data* — about data. Consider the parallels between the *conceptual* forms of such information:

- A record type contains a number of fields, each of which has a name and some attributes.
- A department has a number of employees, each of which has an identifier and some attributes.

These statements are perfectly symmetrical in form. There is an important functional difference: with data about data, modifications need to be carefully controlled; they have consequences that must be carried out by the system. E.g., if

someone says the number of fields in a record type is changed, then the file containing those records had better actually be reformatted.

The distinction between information needed by the system and by applications isn't so sharp either; we have examples in section 8.2.2. And, finally, I would point out that systems can be implemented with the data and descriptions in the same repository, and in the same form (e.g., System R [Chamberlin 76b]).

2.3 What is "In the System"?

The perverse nature of information touches everything: we can't even clearly define what information is "in the system".

For the purpose of this discussion, we distinguish between "raw" and "deduced" information. Raw information is that which the system has no way of knowing unless it is asserted to the system, e.g., the names of the employees in a department. Deduced information is then anything that can be computed or otherwise derived from the raw information, e.g., the number of employees in the department. Under what conditions shall we consider deduced information to be "in the system"? Consider the following cases:

1. The only way to determine the number of employees in the department is to ask for the names and count them yourself.
2. There is a count field in the department record. Whenever an application adds or deletes an employee, the application is also supposed to update the count field.
3. Declarations for the information system define a computed count field. Whenever an employee is added or removed, the system updates the count and stores it with the other information about the department.
4. Declarations for the information system define a count field as part of the data needed by your application. The field is never stored (this is sometimes called a "virtual" data field). Whenever your application retrieves a

department record, the system counts the number of employees and inserts the count into the record produced for your application.
Your application can't tell the difference between cases 3 and 4, except perhaps in the time it takes to retrieve the information.

5. You interface with a query processor, and ask it how many employees are in the department. The query processor extracts the list of names, counts them, and tells you the answer. (Sometimes a query processor can be considered part of the information system. I am thinking more of the case where a query processor can be purchased and installed separately, after the underlying system has been installed and loaded with data. Does installing the query processor increase the information content of the system?)
6. The manipulative interface of the information system provides a count function. For example, it may be possible to request "the next department record for which the count of employees exceeds ten".

In which of these cases does the system "contain" the information? The only unambiguous cases are 2 and 3, where the data is literally stored in the system. The other cases can all be debated, and have been. Consider case 1 from the security point of view. Suppose the size of a department is sensitive information. If you release a list of a department's employees, have you violated security? The security officer would take a dim view of your protest "I didn't tell them how many!".

We will have more about implicit relationships in sections 4.6 and 10.5.

There is another sense in which information may implicitly exist in the system. One often tends to think of information in the system as the contents of various fields in records. A fact in a database is sometimes defined as an association between two fields: one giving the value of an attribute (weight = 200 lb.) and one identifying the entity having that attribute (name = Henry Jones). However, the mere existence of a record in a file in itself

bears information. In the employee file, there is no field giving the employer's name; the presence of the record in the file implies that the corresponding person works for this company. In effect, the employee file serves as an "existence list" (cf. sections 2.4, 10.6.).

Another common form of implicitly represented information has to do with data which depends on some kind of continuous variable, such as time or weight. Examples: the departments to which an employee was assigned at various times, or the postal rates applicable to various shipping weights. Although one can determine what an employee's department was at any point in time, the data is not stored that way. The only things which are actually stored are "breakpoints", i.e., points (of time or weight) at which the information changes. Extracting the data values for a given time or weight involves a combination of table lookup and computation. These are examples of procedural existence tests, essentially of a range test variety (section 2.4), combined with computed relationships (section 4.6).

[Folinus] offers a unifying approach to many of these questions which equates "data in the system" with what can be extracted, rather than with what is physically stored. The system is modeled as a set of named functions, which are capable of returning certain values when invoked with certain arguments. An update presumably modifies the function, so that it subsequently returns different results for the same argument. The implementation of the functions is masked from the user; it might involve simple access to stored data, complex traversals of data structure, and/or computations. Thus the information content of the system is defined by this set of functions, rather than in terms of physically stored data. (It may not be possible to completely hide the implementation; see section 4.6.) This description of information content is still incomplete, of course. We can always mentally infer other information from the values returned by functions. [Hall 75] presents a somewhat similar notion of representing a relation as a procedural mapping. Similar concepts occur in the "accessor" mechanism of [Abrial], and also in inferential systems (e.g., [Ash], [Levien]).

Case 6 does illustrate another important point. The kinds of information that a system is capable of handling are as much determined by its manipulative interface as by its declarative facilities. There is a very thin line between the ability to declare something about a data item and the ability to dynamically request a data item with the same characteristics. Someone has said that "structure is process slowed down". The more you declare, the less your applications have to do procedurally — and the effect is more stable.

In terms of the abstract organization of an information system we introduced earlier: the distinction between processor and repository is not so clear after all.

Still another form of implicit information concerns the *meaning* of the data items in the system. In almost all current data systems, you would be able to retrieve a record containing "Joe Smith" and "95" in adjacent fields, with no clue from the *system* as to whether the "95" was his age, weight, commuting distance, hours worked last week, or something else. If such information is in the system at all, it is traditionally in a catalog or dictionary, quite segregated from the data you are processing, and having to be accessed by entirely different means (section 2.2.3). It is customary to expect that you, the user, know what the fields signify. The manner in which a multiplicity of users get to know, and agree about, what these data items mean is the central point of data description.

Such information could be perceived as part of the "normal" information content of the system if we expect to get answers to such questions as:

- What attributes do we maintain about employees?
- Which attributes of Joe Smith have undefined (or null) values?
- In what relationships is Joe Smith involved?
- What are the relationships between Joe Smith and department Z99?

2.4 Existence Tests In Information Systems

Suppose you said to the processor, across the interface, "Put this in your repository: John Smith lives in Poughkeepsie." How does the processor know there is such a place as Poughkeepsie? Does it care?

Depending on the kinds of entities involved, and often on the whims of the people who set up the system, the processor might go to various kinds of efforts to verify existence. At bottom, they are really acceptance tests, which may or may not have some correspondence with existence. What is being tested is the acceptability of a symbol.

2.4.1 Acceptance Tests: List and Non-List

We can broadly classify the tests as list tests and non-list tests. For list tests, there is an explicit list of existing elements with which the symbol can be compared. If it matches, then the corresponding entity can be presumed to exist.

In practice, such lists tend to be either static or dynamic. A static list is permanently defined, and is usually incorporated into the data description rather then occurring in the data itself. They often occur in the form of rules, e.g., "sex" may be "male" or "female". Such rules, occurring in the data description, tend to be strictly enforced. These static lists can actually be modified, but that's exceptional and traumatic.

Dynamic lists occur in the data itself, and they can be modified by the normal update activity on the data. That is, part of the normal activity on the data includes inserting and removing such entities. A dynamic list might simply be a set of symbols; or it might be a set of objects representing entities, to which the symbols are associated as names. In conventional systems, records often play the role of such lists. E.g., a reference to an employee is acceptable if and only if there is a record for him in the employee file. Actual practice varies widely in this respect; various systems enforce such rules to varying degrees.

Non-list tests involve some procedure other than list checking. The most common forms of these are range checks and syntactic checks. Range checks require valid values to lie between specified limits. Syntactic checks are based entirely on rules governing the composition of the symbol itself; no other indication of existence or meaning is involved.

Syntactic checking is by far the most common, being the essential idea behind the specification of data types. Quite often the only constraint on the acceptability of a symbol is that it must be numeric, or it must be alphabetic, and often must contain a specified number of characters. For numeric quantities, such acceptability is usually an adequate assurance of existence: there actually does exist a real quantity corresponding to each possible sequence of digits. But for alphabetic symbols, there is no such assurance. Symbols are all too often accepted as the names of people, companies, addresses, states, countries, and so on, with no test at all for their existence.

For many entity types, any of the existence tests might be employed in a real system. In practice, tradeoffs are made between the cost of performing the tests and the cost of misrepresenting the existence of the entities. The vast majority of data items in today's files are subjected (in the information system) only to syntactic tests, leaving open the possibility of nonsense references to non-existent entities. While information systems are supposed to be modeling some aspect of reality, there does seem to be a very mixed bag of techniques for synchronizing the system's perceptions with the actual existence of things.

2.4.2 An Act of Creation

Entities whose existence is modeled by a list test require an explicit act of creation. Some overt act is required to establish the existence of such an entity before other things can be said about it. E.g., it has to be included in the list of acceptable values in the entity type definition, or something like the insertion of an employee record must occur before any other reference to that employee is permitted.

(Real systems vary in the degree to which such semantic consistency is enforced, especially in regard to deletion. Many systems will allow references to such an employee to persist in the data even if the employee record is deleted.)

In contrast, entities defined by non-list tests have a kind of "eternal" existence. Once the procedure is defined, the entire set of things acceptable to it exist implicitly. Such entities do not require an overt act of creation prior to being referenced.

2.4.3 Existence by Mention

I biased things a bit by equating existence with acceptance. There is a much simpler sense of existence. We could simply be asking about what things are known to exist at the moment, rather than the acceptability of a certain symbol. In that case, Poughkeepsie exists if it happens to be mentioned as someone's residence, while London doesn't if none of our people lives there.

There are three very different notions of "the domain of cities" (or "the set of cities") operative here:

1. The real set of existing cities (ignoring, for the moment, arguments about historical or fictional cities).
2. The set of cities whose names are acceptable as input (which includes ZZZZZZZ, together with every other alphabetic sequence of acceptable length).
3. The set of cities currently mentioned in the computer's data (which, in our example, excludes London).

In general, this ambivalence will be true wherever the acceptance test is limited to a loose syntactic check.

2.4.4 Existence By Implication

If the computer knows the date you were hired and the date you were fired, it can list all the dates on which you were an employee. Do those dates "exist" in the machine? We'll come back to that in section 2.3.

2.5 Records and Representatives

An attempt to provide a regular modeling of the existence of entities leads to the notion of "representatives".

The traditional construct that represents something in an information system is the record. It doesn't take much to break down the seeming simplicity and singularity of this construct. What is a record? In manual (non-computerized) systems, it could be one sheet of paper, or one card, or one file folder. It might sometimes have a formal structure and boundaries, like a printed form (perhaps several pages long, or extending over several cards). Sometimes it doesn't have much structure, but runs on to several pages or cards (consider a library catalog that has continuation cards), yet with some recognizable convention for distinguishing one record from another.

The concept of "record" is equally muddy in computer systems. The term sometimes refers to:

- A geometrically defined piece of storage medium: card, record within a track on disk, area between blank portions of tape.
- A quantity of data transferred as one chunk between external storage media and main storage (sometimes called a block). This chunk of data generally goes into a buffer area in main storage, managed by an access method.
- A quantity of data transferred as one chunk by an access method between a buffer and an application program.
- Several such chunks, whose contents have some logical relationship (as in [IMS]).

Sometimes a record is associated with a piece of physical medium: the first 80 characters on one track are not the same record as the first 80 characters of another track. Sometimes a record is associated with its contents: "one" employee record may exist simultaneously on a punched card, in a spool file on a disk, in a buffer, and in an application program's work area.

By various rules and conventions, we somehow know how to call a collection of data "one record" even though:

- It may physically exist in several copies (in main memory, on one or more auxiliary storage devices — e.g., a primary copy and a backup copy);
- It may not be physically contiguous (it may be stored in fragments, e.g., span tracks, on auxiliary storage);
- Its location and content change over time.

Thus, even at this level, we do not have a truly tangible, physical construct called "record", but rather we have to deal with it abstractly. We try to get by with some concept like "a record is that data which appears in my buffer whenever I submit a certain key to a certain access method using a certain index" (and even that is full of holes).

In some uses of the term "record", its characteristics are constrained by the processing system (device and media characteristics, access method) rather than by the information content appropriate to an application. This might include such constraints as:

- absolutely fixed length records (e.g., 80-byte card image).
- declarable but fixed lengths per "record type" or per "file".
- upper limits on record lengths.
- fixed number of fields per record.
- fixed length fields.

In general, the record concept grew out of data processing technology, and reflects many things besides the desire to represent an entity in a model of information. (More on this in chapter 8.)

Thus, something that an application might want to deal with as a "medical record" may not correspond to a single "access method" record, but to some kind of structured collection of such

records. In this sense, the IMS concept of a record (consisting of a variable number of structurally related segments) is a fairly good approximation to an application's concept of a record.

We are after some single construct that we can imagine to exist in the repository of an information system, for the purpose of representing a thing in the real world. Beyond grappling with the definition of "record", we have another traditional problem to contend with. In many current information systems, we find that a thing in the real world is represented by, not one, but many records. A library book is represented in the catalog by at least a title card, an author card, and a subject card. A person may be represented by a personnel record, a health record, a benefits record, an education record, a stockholder record, etc. In this latter case, of course, we are viewing all the files maintained by one company as constituting a single information system.

One objective of a coherent information system (i.e., an integrated database) is to minimize this multiplicity of records, for several reasons. In the first place, these various records usually contain some common information (address, date of birth, social security number); it takes extra work to maintain all the copies of this data, and the information tends to become inconsistent (sooner or later, somebody will have different addresses recorded in different files). Secondly, new applications often need data from several of these records.

So, we integrate these various records into one "pool" of data about an individual — and thereby introduce several new concepts of "record". On the one hand, it might be this pool of data. On the other hand, it is often used to mean that data which an application sees, which might bear no simple physical resemblance at all to the underlying pool of data. A "medical record" would consist of some subset of data out of this pool, perhaps collected from scattered physical locations, and formatted to the requirements of some application.

Well, then. If we can't pin down "records" to represent things in the real world, could we somehow use this underlying pool of data as a representative? Maybe. The problem is that we would like the representatives of two things to somehow be cleanly disjoint, to be distinctly separate from each other.

Unfortunately, much of the data about something concerns its relationships to other things, and therefore comprises data about those other things as well. The enrollment of a student in a class is just as much a fact about the student as about the class. So, we can't draw an imaginary circle around a body of information and say that it contains everything we know about a certain thing, and everything in the circle pertains only to that thing, and hence that information "represents" the thing. Even if we could, the concept is just too "smeared" — we need some kind of focal point to which we can figuratively point and say "this is the representative of that thing".

We won't try to solve this problem. We will simply skirt the whole issue and continue to use the term "representative" (borrowing it from [Griffith]; [Hall 76] uses the term "surrogate"). We need the terminology to develop some concepts of information representation, without getting too tangled up in machine processing constructs. In some situations a representative may correspond to a record, or to a segment (IMS), or to a row in a relation; sometimes none of these constructs quite fits the concept of a representative.

Another reason for introducing the term "representative" is that our topic is broad enough to include systems that don't even use the term "record". In computer catalogs and directories, we have "entries", and in data dictionaries we have "subjects".

Although it is an abstraction, related to a theoretical view of data and data description, the representative has some definite properties, some of them reflecting the computer environment which is its ultimate motivation. The characteristics of a representative in an idealized repository might include these:

- A representative is intended to represent one thing in the real world, and that real thing should have only one representative in an information system. There may be some controlled redundancy in the physically stored data, such as duplicate copies of records in order to optimize different access strategies. That doesn't violate this principle, if there is some provision for keeping the contents of such records acceptably synchronized. Note

that we have the corollary concept of information systems themselves as bounded, disjoint collections. Something in the real world may have several representatives in several information systems, but should have no more than one representative in each. Note further that this last constraint is a matter of intent, not definition. Something in the real world may in fact have several representatives in one information system, due to that system's failure to detect the duplication. This is an error situation that nonetheless can occur. It's a familiar headache to welfare agencies: a person fraudulently drawing benefits under several different names.

- Representatives can be linked. This is the fundamental basis for representing information (in addition to the fact that the representatives exist).
- The information expressed by linking representatives includes such things as relationships, attributes, types, names, and rules.
- The kinds of rules that generally need to be specifiable about representatives include conventions governing their type, names, existence tests, equality tests, and general constraints on their relationships to other things.
- For representatives with explicit existence tests, the representative must be created by an overt operation on the information system. It does not exist simply because its counterpart in the real world exists. An information system may or may not be able to detect the creation of two representatives for the same thing. It will assume that two representatives represent two different things.
- The information associated with a representative must be asserted explicitly to the information system. The system is not omniscient. (We exclude here information that can be computed or derived from other asserted information.) The accuracy and currency of the information is determined by the assertions.
- It would help to have some mechanism to clearly and unambiguously indicate what is meant by "one" and "the

same" representative. A numbering system would suffice, wherein each representative is assigned a number unique within an information system, and numbers are not re-used after a representative is destroyed. Whether or not it is provided in a real implementation, this scheme defines the concepts of "one" and "same" representative in an information system. A representative always has exactly one and the same number, and no other representative ever has that number. Two references (e.g., names, relationships) refer to the same representative if and only if they refer to the same representative number.

3 Naming

3.1 How Many Ways?

The purpose of an information system is to permit users to enter and extract information — about entities. Most transactions between user and system require some means of designating the particular entity of interest. In order to design or evaluate the naming facilities of an information system, it helps to be aware of the variety of ways in which we designate things.

How do we indicate a particular thing we want to talk about? Let me describe just a few of the ways I can think of:

- You point your finger. By itself that's generally ambiguous, unless there's something in the context or conversation to indicate whether you are pointing to a button, a shirt, the man's chest, the man, the horse and rider, or the whole regiment.

  Is this relevant? How about pointing a light pen at a display screen? If you are editing text, there has to be some way to establish whether you are erasing a letter, a word, a line, a sentence, a paragraph, or a letter (lovely ambiguity).

- If it's a person, you might use his name. Did I say "name", in the singular? There are many different sequences of letters and punctuation that are recognizable as his "name". There's his full name (with or without Ms. or Dr. or Capt. in front and Jr. or II or MD or Ph.D. in back); you might omit his middle name(s), or use only initials for either his first or middle names, or use only his initials altogether (monogram); you might use a nickname, or just the initial of his nickname (I sometimes get memos addressed to B. Kent); you might address him only by his first name or

nickname, or by his last name only; and in some cases you have to give his last name first. And then we have the curious custom of addressing married women by their husband's names: Mrs. Henry Smith.

These are all, of course, ambiguous. People's names are generally non-unique. Whether you address a person by full name, first name, or nickname, there is always some chance that someone else will respond. We employ a variety of techniques and assumptions to insure that we are addressing the right person; sometimes they don't work. A newspaper editor in France was the victim of an assassination plot aimed at a political figure with the same name.

And what is the syntax of a person's name? Certainly far more complex than first name, blank, last name. There may be periods, commas, blanks, hyphens, apostrophes (anything else?) in certain places (what are the rules?). A last name may have embedded blanks, and may not even start with a capital letter (van den Berg). With a name like that, do we always know where the split occurs between first, middle, and last names (especially if the name is printed in all capitals)? People can have any number of middle names. Is there any limit on the length of people's names?

- Notice that I started that last discussion with "if it's a person". In identifying something, a name may be meaningless unless you also establish the category of the thing. What does the name "Colt" identify? It might be a person, a gun, a car, a beer, a football player, or perhaps even a city or county somewhere. (Check an atlas?) Of course, I'm being a little careless here. "Colt" is not the name of one gun or one can of beer or one football player. And, if it's not even clear that I am using a name, I might just be talking about a horse.

Sometimes you can't tell whether an entity or a category is being named. Try asking an operator for the phone number of "The Restaurant" or "The Movie".

- A thing can have different kinds of names. A person might be identified by a social security number, employee number, membership number in various organizations, military service number, various account or policy numbers (strictly speaking, these latter don't identify him, but something he's related to; on the other hand, you might also say that about a social security number). A car might be identified by license number or by engine number. A department may have a name (Accounting) and a number (Z99). A book has a title, a Library of Congress number, an ISBN (International Standard Book Number), not to mention various Dewey decimal identifiers in local library catalogs. And each copy of a book may have an "accession number", assigned locally by a library for their overall inventory management.

  So, to be complete, we may sometimes have to indicate the kind of identifier being used, in addition to the identifier itself and the category of the thing. Very often, but not always, the kind of identifier is implicitly understood (a social security number is generally recognizable by its format). Of course, in a data processing system, there has to be some convention for indicating which field in the record is to be matched.

- You don't always have all these options. Very often you have to know who you're talking to; that will determine how you have to identify the thing being referenced. In addressing mail, you had better include the last name. For the IRS or a stockbroker, you might have to use social security number. The personnel file might only be keyed on employee numbers. If the personnel file can be addressed by name, there are probably some very

specialized rules, e.g., all capitals, last name first followed by a comma, truncate all names longer than 25 characters, etc. To log on as a user of a teleprocessing system, you may have to present a special identifier assigned within that system.

- You might even have a choice of several different names of the same "kind" for the same thing. A married woman often retains her maiden name for professional purposes. People have stage names, pen names, aliases, and sometimes several nicknames (Chuck and Charley).

  A person's name may have several *correct* spellings, especially if it is transliterated from a foreign language. Look up the composer of "Swan Lake" in several catalogs.

  In Hawaii, last names are often so long that people just use the first few syllables.

  When a book is published or distributed by different publishers (perhaps in different countries), then the book may bear several International Standard Book Numbers (ISBN). [Douque] is an example.

  Vacuum tubes often have several numbers designating the same type of tube.

  If you will accept a phone number as the "name" of a telephone, then we include the possibility of several names for the same instrument. The phone may also respond to several "kinds" of names: outside numbers and internal extension numbers.

  Sometimes the alternative names (synonyms) can be predicted by a rule (algorithm) rather than requiring an explicit list. Many command systems allow the names of commands to be truncated from the right. Thus PR, PRI, and PRIN might all be recognized as synonyms for the command named PRINT, even if they aren't explicitly listed anywhere in the system.

The term "synonym" sometimes refers to the existence of several kinds of names (e.g., employee number and social security number), and sometimes to the existence of several names of the same kind (e.g., a person's aliases).

- You might refer to someone or something by its relationship to another identified thing: Charley's aunt, Harry's car, the owner of a certain bank account or charge account. (How about "Mrs. Henry Smith"?) Such references may or may not be unique.
- Or by the role currently being played by the thing: the mailman, the bus driver, the third baseman.
- Or by its attributes: the red car, the highest-paid employee.
- And certainly by combinations of these: the red car's owner's lawyer. Again, these references may or may not be unambiguous.
- We often address a letter to a person when we really want to deal with his role (e.g., manager of a certain department). If someone else now has his job, we really want the matter handled by his replacement.
- A name sometimes describes the thing being named. Sometimes it doesn't. Main Street may or may not be the main street in town. It may not be a street at all (restaurant? clothing store? movie or book?). Does Scotch Tape come from Scotland? How many blackboards are black?

  When my daughters were very young, they had a toy they called "Blue Car". It was a yellow donkey. (Or was it a toy, and not a donkey? Shall we debate whether the category of animals includes toys, pictures, statues, and other imitations of animals? Are you going to insist that the toy was not a donkey?)

On television one night, the "8 O'Clock Movie" started at 7 o'clock.

And then of course there are code names, which are deliberately uninformative or even misleading.

- Some names have embedded in them information about the thing being named. In some states, you can determine from a license plate the county in which it was issued, or the fact that the car belongs to a rental or leasing agency, or to a government agency. An account number often has the bank branch number included (this also relates to qualification — section 3.3.2). Prefixes very often have special meaning, as in the names of modules of computer programs.
- When dealing with ambiguity, we sometimes employ a complex strategy of reducing the number of candidates to one. Sometimes it is a pre-established strategy, e.g., specify name and address, or name and date of birth. Sometimes we do it in dialog fashion: "Is this the John Smith who works for IBM?" "Yes, but there are three." "Were you at the ACM meeting last week?" And so on.

  Some other strategies:

  - Take the first one encountered, according to some ordering. This is the common treatment of "non-unique keys". Sometimes this ordering is determined by an ordering of "scopes" (discussed below), e.g., catalogs.
  - When things are versioned, default reference is to some "latest" version.

  Qualification is an especially important technique, which will be discussed later.

- We also refer to things by pronouns (you, her, it, that), which depends on some convention to establish the object of reference. A common convention is to assume it to be a previously identified thing. This, too, has its counterpart in data processing systems. In IMS, for example, the "replace" function is assumed to refer to

the last record that had been retrieved (that's accurate enough for this discussion; the precise rule is a bit more complex).

- Sometimes we refer to something without knowing yet exactly which thing we are talking about. A mystery novel refers to "the murderer"; a contest announcement mentions "the winner". This is analogous to using variables in algebra and in programming. (It also bears some resemblance to roles, and to pronouns. In the case of roles, we are less likely to care about which individual is actually playing the role than when we use variable reference. The distinction between variables and pronouns does not so readily come to mind.)
- In programming, an important special case of reference by relationship involves some ordering. With respect to such an ordering, one can refer to the first item, or the last, the next, the third, etc. "Next" also involves pronoun reference, since it implies "next" relative to whatever item was last referenced.
  We make ordinal references to footnotes, bibliographies, pages, chapters, volumes, editions, etc.
- In programming, "pointing" often means naming some location in the machine. It is something like pronoun reference ("that") in that it involves some convention to establish what is being referenced, i.e., what is assumed to be at that location (both its nature and its extent: character, field, record, control block, etc.).

Which of these phenomena shall we call "naming"? No answer. It doesn't matter.

Can we distinguish between naming and describing?

On one hand there is a pure naming or identification phenomenon: a string of characters serves no other purpose than to indicate which thing is being referenced. On the other hand we have information about the attributes of a thing and its relationships to other things. Of course, the two overlap.

There are very few "pure" identifiers, containing no information whatsoever about things. A person's name suggests

possible relationships to other people; a first name can indicate a person's sex; a name often conveys ethnic clues; it may suggest something about age or social status; in some forms, it may indicate profession, or level of education.

The serial number of a part often implies something about the date or place of its manufacture, or something about the presence or absence of certain features. Vacuum tube numbers encode much information about electrical and mechanical specifications. An International Standard Book Number (ISBN) encodes publisher, author, title, and type of publication.

3.2 What is Being Named?

Which entity is being named? Consider telephones and telephone numbers (analogous to message handling in a message processing system). If, as before, we consider a phone number to be the name of a telephone, then:

- a telephone may have several names (several numbers ringing the same phone);
- a given number could ring several phones: the several extensions in your home, or the phones of a manager and his secretary;
- phones can change names (numbers): the phone company replaces a defective telephone, or the phone company assigns new numbers, or you transfer your number when you move.

Alternatively, we could invent a new abstract entity, e.g., a "message destination" (in teleprocessing systems, a "logical unit"). We then consider one phone number to be the name of one message destination, and we deal with a *relationship* between message destinations and telephones (in teleprocessing, logical units and physical units). This relationship could be many-to-many, and can be changed. And it now requires some method for identifying (naming) the physical telephones involved.

A familiar message again: you the observer are free to *choose* the way you apply concepts to obtain your working model of reality.

3.3 Uniqueness, Scope, and Qualifiers

Whether a name refers to one thing or many frequently depends on the set of candidates available to be referenced. This set of candidates comprises a "scope", and it is often implicit in the environment in which the naming is done. A reference to "Harry" is often understood to mean the Harry present in the room. A letter addressed to Menlo Park (without naming the state) will probably be delivered in California if mailed on the West coast, and to New Jersey if mailed on the East coast. The boundaries of a scope, and the implicit default rules, are often fuzzy: I don't know where the letter would go if it was mailed in Illinois.

Qualification, the specification of additional terms in a name, is often used to resolve such ambiguities by making the intended scope more explicit. In this case, adding the state name would (partially) resolve the ambiguity.

Scopes are often nested, and we often employ a mixed convention: a larger scope is left implicit, but a sub-scope within it is explicitly specified. This is partial qualification. There are cities named "San Jose" in Costa Rica and in the United States. Let's imagine that the one in Costa Rica is within a "district" named California. Then the address "San Jose, California", although qualified, is still ambiguous. Whether the letter gets to its intended destination depends on the "default scope" (i.e., country) implied by the point at which it is mailed.

Even the city name is a scope, resolving the ambiguity of a street address — University Avenue exists in many cities. And the street name selects a scope of house numbers. A complete address is a whole chain of scope qualifiers.

Telephone numbers provide familiar examples of qualification. A (7-digit) phone number is certainly not unique; it may exist within many different area codes. Here the boundaries of the scopes, and the default rules, are well defined. If you don't

dial the area code, the destination is assumed to be within the area you are dialing from; otherwise you must dial the area code explicitly. Of course, this is again only an imperfect model of reality; I can reach many destinations in nearby area codes without dialing the area code. The first three digits of the seven-digit number seem to imply something in relation to area codes.

When dialing from an office phone, one often has to select first from a larger set of scopes: whether you want an outside line (dial 9), a "tie line" connection (dial 8), or a local extension (in some places, dial 6; in others, the default in the absence of an initial 9 or 8).

Incidentally, phone numbers illustrate some kinds of anomalies that may occur in real naming conventions:

- Different forms of names are valid within different scopes: for local extensions, they are four digits; for outside numbers, they are seven digits plus optional area code; a tie line number could have still another form.
- Form and content (syntax and semantics) are mixed together. You can't specify the naming rules independent of the numbers involved. Certain initial digits are reserved for certain functions. If the first digit you dial is zero, then you are addressing the operator, not selecting a scope (you could fudge that by confusing a scope with its single member). Certain three digit numbers are valid destinations, and not part of a seven-digit number (like 411 for information).
- The naming conventions can depend on the scope *from* which the naming is done: the phones at another location may have a different convention for getting outside lines, local extensions, etc.

If you consider a company's internal switchboard operator to be part of the addressing mechanism, then you could think of a very completely qualified name as consisting of a 9 for an outside line, an area code and seven digits, followed by a four digit office extension. Two points:

- If you think of the destination switchboard as another scope, then here again different scopes may accept different names: some companies have 3-digit extensions, some use four, etc. Furthermore, many *kinds* of names and descriptions are acceptable within the switchboard's scope: extension numbers, people's names or roles ("personnel manager, please"), names or descriptions of departments ("shipping, please"), etc.
- The scope itself may have multiple synonymous names: the company may have several numbers that ring at its switchboard.

To elaborate on an earlier point: the same thing might have different names when referenced from different scopes:

- Phone numbers, as mentioned above.
- Programs, files, etc. might be registered in different catalogs with different names (don't know if any current systems support this).
- Users of the IBM Virtual Machine Facility (VM) will recognize this: I can get access to the disk you call 191 and call it 193 myself (when I have another disk which I call 191).

In this case, the name does not go with the entity, but is an "attribute" of the relationship between the entity and the scope (i.e., it goes with the directory entry).

3.3.1 Deliberate Non-Uniqueness

Quite often, things don't have individual unique names. This poses no problem when the things aren't individually represented in the system. In the case of parts, for example, we have one named representative for a type of part; the existence of individual instances is reflected only in the "quantity on hand" attribute.

Consider, however, something like a table of organization for a military unit. There may be several slots for clerks, with each slot having the same job description and skill requirements. We want them separately represented; they are the permanent entities

in this structure. One of the *attributes* (or relationships) we want to record for them is the name of the person currently holding the position. When the positions are vacant, the information associated with the entities is identical. When we want to address one of them, e.g., to assign someone to a job, it is sufficient to refer to "any one of the vacant clerk positions". For this kind of information, the entities do not require unique identification.

It is sometimes asserted that each entity represented in the system must have a unique identifier. I contend that this is a requirement imposed by a particular data model (and it may make many things easier to cope with), but it is not an inherent characteristic of information.

3.3.2 Effective Qualification

A scoping object does not have to have any intuitive connotations of "scope". It need not be a physical region, or a catalog, or an area code. Quite often, the technique for giving something a unique qualified name is simply based on an arbitrary relationship to some other object. In effect, the scope becomes the set of things having a particular relationship to a particular object.

Consider, for example, the naming of employees' dependents by the two fields consisting of the employee identification number plus the dependent's first name (the example is taken from [Chen]). In order for such a convention to be effective, a number of conditions must be satisfied.

Uniqueness Within Qualifier

The relationship must confer uniqueness of simple name within relative (i.e., the employee must not have two dependents with the same simple name). Curiously enough, even this might not hold for the given example. A pathological case would occur if the employee had several children with the same name (or is that in fact plausible with adopted children? or after remarriage?). More reasonably, his wife and daughter might have

the same name, or his father and son (and grandson, if he was an eligible dependent).

Singularity of Qualifier

The relationship does not actually have to be one-to-many for naming purposes, so long as the previous constraint on uniqueness holds for each relative. Thus a person could be a dependent of several employees, and still be uniquely identifiable, so long as no employee has two dependents with the same first name.

However, this situation does give rise to synonyms: a given dependent could be identified by qualification by any of his related employees. This could lead to a number of problems, such as determining when two references to dependents were really references to the same person. And also: when a new employee lists his dependents, how shall we know if any of those dependents are already recorded as dependents of other employees? (Do we add new dependent records, or add synonyms to existing records?)

To avoid such problems, one could require that the identifier have no synonyms. Then dependents could no longer be identified via their related employees — unless we wanted to deny the reality that a person might be a dependent of several employees.

Another alternative is to require that one of the synonyms be designated the "primary" identifier, being the only one permitted to be used in referring to that dependent. With this constraint we lose usefulness and naturalness. How do we know which employee to use in referring to a dependent? If an employee asks me to add one of his dependents to some list, I first have to find out whether I might have to use some other employee's number to form the dependent's identifier. If I have to do that lookup in the dependent's record, I might as well be getting some arbitrary identification number out of it instead of bothering with qualified names. And this convention doesn't solve the problem of change. If a dependent's "primary" employee leaves the company, and another relative still works there, then all references to the

dependent will have to be modified to reflect a different primary employee in his qualified name.

Existence of Qualifier

A qualifier must exist for each entity occurrence. Therefore the relationship must not be optional; each dependent must have a corresponding employee. If the benefits program were expanded, let's say as a charitable community service, to cover needy people unrelated to any employee, then this system of entity identification would no longer work.

Invariance of Qualifiers

Such a relationship must really be invariant (unmodifiable). The relationship constitutes information that is redundantly scattered about everywhere that this entity is referenced, with the potential for enormous update anomalies if the information can change. (Qualified names thus violate the spirit, if not the letter, of relational third normal form [Codd 72], [Kent 73].) Even this requirement might not be satisfied by the example cited. For tax purposes, two married employees might wish to change which one of them claims which children as dependents; such a change would have to be propagated into the qualifiers in every single reference to those children.

3.4 Scope of Naming Conventions

The oil well problem: some oil wells, but not all, have "API" codes assigned by the American Petroleum Institute. Oil companies assign their own names to the wells they own, using their own conventions and formats. Some wells are jointly owned, with each owning company naming the well according to its own rules.

In a database to be used for correlating data on all wells in some area, no single naming convention would apply to all the wells. The API code works for those wells that have them. Otherwise, you have to know who the owner is (or which

owner's convention is being used, for jointly owned wells) before you know the applicable name format. When one company writes an application looking only at its own wells, it would like to see and use its own names. A second company's application would like to see and use that company's names, even when some of the same wells are involved.

The common solution: develop a brand new naming system (keys) for all wells represented in the database. Now everybody has to learn a new set of names, and correlate them with the ones they already know. And the headache will recur when several such databases are integrated.

(Lots of people don't have social security numbers — such as the employees of a multi-national corporation.)

3.5 Changing Names

Names do change: people, streets, cities, nations, companies, divisions, departments, programs, files, projects, books, other publications. Part numbering systems change once in a while. Mistakes get corrected.

In the references at the end of this book, SIGFIDET and SIGMOD are the old and new names of the same organization. Did you know that?

How (and how long) do you detect and handle references to the old names? Is this similar to synonyms?

The common solutions: either disallow name changes (pretend they don't happen), or generate a new naming scheme for the data system and treat the other (changeable) names as attributes. The latter solution has a price, of course: increased space required for storing and indexing the additional names, learning and processing problems in dealing with new, "unnatural" names; possible loss of "key" facilities of some access methods (e.g., if secondary indexing weren't available).

Systems that depend on symbolic associations for paths (e.g., the relational model), as opposed to internal "unrepresented" paths between entities, cannot readily cope with changing names [Hall 76]. That is a fact; we might, however, debate whether it is a fault or a virtue.

When name changes are disallowed by the system, one can trick the system by deleting the entity, and then inserting it again as a "new" entity under its new name. Unfortunately, it is sometimes very difficult, if not impossible, to discover all the attributes and relationships associated with the old entity, so that they may be re-established for the new entity. And sometimes deletion and insertion might have undesirable semantic implications of their own, enforced by the system and perhaps unknown to the application that is trying to change a name. This technique for altering an employee's identifier could enter a spurious firing and re-hiring into his employment history.

3.6 Versions

Quite often several versions of a thing are available, reflecting the status of the thing after various changes. The thing might be a document (e.g., various printings or editions of a book), a program, or a set of data records. (When data records are kept on magnetic tape, the traditional way to update the data is to rewrite it all to a new tape, incorporating the desired changes in the process. It is common practice to retain several "generations" of such tapes, for backup and error recovery purposes.)

The central problem with the version concept is that we can't decide whether we are dealing with one thing or several. "The payroll program" is a singular concept, and a command to execute it is implicitly understood to refer to "the current version". On the other hand, one sometimes refers explicitly to an old version; for example, in order to reconstruct how a certain error occurred last month, one may want to rerun the version of the program that was current then. In this context, we are explicitly aware of the several versions as distinct entities, and have to specify the desired version as part of the naming process.

3.7 Names, Symbols, Representations

What is a name but a symbol for an idea? What essential difference is there between "Kent" and "25" and "blue", other than that they name different things?

3.8 Why Separate Symbols and Things?

3.8.1 Do Names "Represent"?

In linguistics, a symbol is itself a representative of the thing it names. We have no choice; there isn't anything else. In the conventional linguistic view of verbal communication (written and spoken), including our normal communications with computers, we have nothing else except character strings to represent the things we are communicating about. This leads some people to conclude that we must use such symbols as the representatives of entities.

But in a modeling system, we do have an alternative. We *can* postulate the existence of some other kind of object inside a modeling system that acts as the representative (surrogate) for something outside the system. There "actually" is something in the system (a control block, an address in virtual memory, or some such computer-based construct) which can stand for a real thing. Once we've done that, we can talk about the symbols that name a thing separately from the representative of that thing.

Does this have any counterpart in our own experiences? Do we ever use anything besides words for communicating? Do we ever use pictures?

Consider the way we often use graph-like diagrams to supplement verbal communication, to help cope with synonyms and ambiguities in symbols. Even the authors who want us to use symbols exclusively use such diagrams in their own papers. Our thing object is essentially a node on a graph, before any label has been written in it. We can decide what that node stands for before we write any labels; we then have a variety of options for

choosing the label, and we may even change the label at various times. The same label might also occur on another node, but then we know it stands for something else. Or we might not write any label, because we can refer to it by its relationships to other nodes. But through all this, it is the node that has constantly been the representative of a certain thing, independent of the labeling considerations.

This is not to say that we can do without character strings. They are absolutely indispensable in describing and referring to what is being represented and linked. What we have done is to shift the primary responsibility for representing things away from character strings and onto a system of objects and links. Then we use character strings for description and communication. This shift of responsibility gives us greater freedom in how we use the character strings, and helps us escape a multitude of problems rooted in the ambiguity and synonymity of symbols.

This idea of taking the label out of the node, of treating an object separately from the various symbols with which it might be associated, should be exploited for a number of reasons:

- We can cope with objects that have no names at all (at least in the sense of simple labels or identifiers). We can support other ways of referring to an object, e.g., via its relationships with other objects.
- The separation permits symbol objects to be introduced and described (constrained) in the model, independent of the objects that they might name. One can thus introduce the syntax of data types, social security numbers, product codes, etc. (This is relevant to a certain level of information validation, independent of questions of implementation or internal representation.)
- Naming rules can be expressed simply in the form of relationships between thing types and symbol types.
- Other useful relationships might be expressed among symbols: synonyms, abbreviations, encodings, conversions.

- Various kinds of relationships might exist between things and strings:
  - Present name vs. past.
  - Legal name vs. pseudonym, alias, etc.
  - Maiden name vs. married name.
  - Primary name vs. synonym.
  - Name vs. description.
  - Which name (representation) is appropriate for which language (or other context). This could be useful in multi-lingual environments, such as the UN, the EEC, multinational corporations, and countries such as Canada, Switzerland, and Belgium.
- The structures of names can be distinguished from the structure of an object. For example, a particular day, such as the day on which you were born, is a single concept, a single entity. Its names, however, come in various forms. Most of the conventional notations take three fields; in Julian notation, however, it occupies one field. (And something else to think about: is the representation of a date in years, months, and days really all that different from representing a length in miles, feet, and inches?) Thus we should generally avoid confusing the structure of an object with the structures of its names.
- The separation permits differentiating between different *types* of names for a given thing, e.g., person name, employee number, social security number. Such types are themselves a normal part of the information structure available from the model.
- By distinguishing sets of things from sets of signs, we can avoid confusing several kinds of assertions:
  - Assertions about real things: “every employee must be assigned to exactly one department”.
  - Assertions about signs: “a department code consists of a letter followed by two numbers”.
  - Assertions relating things and signs: “a department has exactly one department code and one department name”.

3.8.2 Simple Ambiguity

"It all depends on what you mean by ambiguity."

We mustn't neglect the plain and familiar ambiguities, which make their own large contribution to our communication confusion. Most words simply do have multiple meanings; we can't escape that. Some comments and corollaries:

- As evidence of the multiplicity of meanings, simply consider the average number of definitions per word in a dictionary. Then extend that to include all kinds of dictionaries, e.g., glossaries of specialized terms. Then add in the undocumented varieties of jargon used in various specialties. And include all the times a technical article begins by defining the terms it will use. And allow for variations in usage in different parts of the country, and in different countries. And slang, and metaphor.
- Ambiguity appears to be inevitable, in an almost mathematical sense, if we consider the relative magnitudes of the set of concepts and the set of words. The set of concepts that might enter our minds appears to be quite infinite, especially if we count every shade of meaning, every nuance and interpolation, as a separate concept. On the other hand, the number of words of a reasonable length (say, less than 25 letters) which can be formed from a small finite alphabet is quite small in comparison. It seems inevitable that many of these words would have to be employed to express multiple concepts.
- "...fuzziness, far from being a difficulty, is often a convenience, or even an essential, in communication and control processes. It might be noted that in ordinary human communications, the ability to stretch and modify word meanings is essential. There are many more situations occurring in life than we have ready-made tags for. Even so simple a word as 'chair' has all kinds of readily visible complexities in its use. It has *ambiguity*,

in that it has more than one distinct area of application (in addition to the usual, we have 'Would the chair recognize my motion now?' and 'Would you like to chair this meeting?'). *Vagueness* (or fuzziness) is closely related to *generality*, the possibility of referring to more than one object. In fact, without generality, language would be almost impossible. Imagine if we had to give each chair a new proper name before we could talk about it! As far as *'stretchiness'* is concerned, note that some people make a living designing objects they call 'chairs', but in which other people might sit with only the greatest reluctance. The concept of 'chair' is constantly evolving, in fact" [Goguen].

- The complexity of legal jargon testifies to the difficulty of being precise and unambiguous.
- Observe the number of puns and jokes that depend on ambiguity ("walk this way").
- If you listen carefully, you will discover all kinds of ambiguities occurring continuously in your daily conversations. If you listen too carefully, it could drive you out of your mind. Consider:
  - When a receptionist directs you to "go through the same door as you did yesterday", she refers to doorway, not the door. Would you care if carpenters had replaced the door in the meantime? Or the doorframe?
  - "Turn left at the second traffic light" means you should turn left at the second intersection that has traffic lights. The first such intersection probably has two traffic lights itself.
- Why should we expect the language which describes a customer's business to be any better understood or less ambiguous than the language which describes our own? Data theorists are ready to argue about any of the following words and phrases: data, database, data bank, database administrator, information system, data independence, record, field, file, user, end user, performance, navigation, simplicity, naturalness, entity,

logical, physical, model, attribute, relationship, relation, set, integrity, security, privacy, authorization, ...

3.8.3 Surrogates, Internal Identifiers

Some alternative models suggest that some sort of an internal construct be used to represent an entity, acting as a "surrogate" for it ([Hall 76]). This surrogate would occur in data structures wherever the entity is referenced, and naming problems would at least be isolated by keeping structured or ambiguous identifiers off to one side, outside the structures representing attributes and relationships.

Since these surrogates must eventually be implemented inside the computer in some form of symbol string, it is sometimes held that such surrogates are themselves nothing but symbols.

It is useful to be aware of some fundamental differences between surrogates and ordinary symbols:

- A surrogate need not be exposed to users. Only ordinary symbols pass between user and system. In concept, models involving surrogates behave as though a fact (e.g., the assignment of an employee to a department) was treated in two stages. First, the surrogates corresponding to the employee and department identifiers are located (i.e., name resolution). Then the two surrogates are placed in association with each other, to represent the fact.
- Users do not specify the format, syntax, structure, uniqueness rules, etc. for surrogates.
- Surrogates are globally unique, and have the same format for all entities. The system does not have to know the entity type before knowing which entity is being referenced, or before knowing what the surrogate format will be.
- Surrogates are purely information-free. They do not imply anything about any related entities, nor any kind of meaningful ordering.

- A surrogate is intended to be in one to one correspondence with some entity which it is representing. In contrast, the correspondence between symbols and entities is often many-to-many.
- Surrogates are atomic, unstructured units. E.g., there is never a question concerning how many fields it occupies.

3.9 Sameness (Equality)

3.9.1 Tests

A counterpart of the existence test of section 2.4 is the equality test. When shall two symbol occurrences be judged to refer to the same entity? (We mean "symbol" broadly in this context to include phrases, descriptions, qualified names, etc.) In general, different modes are applicable to different entity types. It is as much a specifiable characteristic as the naming conventions themselves.

We can describe several kinds of equality tests: match, surrogate, list, and procedural.

A match test is based on simple comparison between the symbols. They are judged to refer to the same entity if and only if the symbols themselves are the same (by whatever rule sameness is judged, with regard to, e.g., case, font, size, color, etc.). Addresses are typically treated in this way; any variation in the character sequence implies unequal addresses.

In a surrogate test, each symbol is interpreted to refer to some surrogate object (e.g., a record occurrence). If both symbols refer to the same surrogate, the symbols are judged equal. (Following [Abrial]: "Equality always means identity of internal names.")

A list test involves a simple list of synonyms. E.g., they might indicate which color names are to be considered synonymous (crimson and vermilion might occur together in one company's list, but not in another's), or give the various forms of

abbreviation for a given term. If the two symbols occur in the same list, they are judged equal.

A procedural test involves some other arbitrary procedure by which the two symbols are judged equal. These are most often performed in relation to numeric quantities.

It is not generally acknowledged that equality tests for numeric quantities exhibit much the same characteristics as equality tests for non-numeric symbols. For numeric quantities, a number of factors are generally involved:

- The quantities are more likely to be judged equal if they were initially named by the same "conventions", i.e., measured and recorded with the same precision.
- The quantities need to be "converted" into common units of measure, data types, representations, etc. These are, in effect, replacing the original symbols with procedurally determined synonyms.
- Compare the two symbols. In many cases, the quantities only have to match within a certain tolerance ("fuzz") to be judged equal. This is another procedure for recognizing synonymous symbols, effectively similar to explicit lists of synonyms (considering crimson and vermilion to be equal is really a form of fuzz; to some people the difference in those two colors is significant).

There is certainly some interaction between the forms of the equality tests and the existence tests. Not all of the equality tests are applicable to entities subject to each of the existence tests.

3.9.2 Failures

When equality is based on symbol matching, several kinds of erroneous results can arise.

- If things have aliases, then equality will not be detected if two different names for the same thing are compared.

- If symbols can be ambiguous (name several things), then spurious matches will occur. Different things will be judged to be the same, because their names match.

(When qualified names are involved, another kind of spurious match can occur — see section 8.8.3.)

These concerns are especially relevant when attempting to detect implicit relationships based on matching symbols.

In general, when aliases are supported, we have to know:

- When two symbols refer to the same thing.
- Which symbol(s) to reply in answer to questions.
- Whether use of a new symbol implies a new object or a new name for an existing object.

4 Relationships

Relationships are the stuff of which information is made. Just about everything in the information system looks like a relationship.

A relationship is an association among several things, with that association having a particular significance. For brevity, I will refer to the significance of an association as its "reason". There is an association between you and your car, for the reason that you own it. There's an association between a teacher and a class, because he teaches it. There's an association between a part and a warehouse, because the part is stored there.

Relationships can be named, and for now we will treat the name as being a statement of the reason for the association (which means we will sometimes invent names which are whole phrases, such as "is-employed-by"). As usual, we have to be careful to avoid confusion between kinds and instances. We often say that "owns" is a relationship, but it is really a *kind* of relationship of which there are many instances: your ownership of your car, your ownership of your pencil, someone else's ownership of his car. I will often (but not consistently) use the unqualified term "relationship" to mean a kind, and add the term "instance" if that's what is meant. So, to be precise, our opening definition was of a relationship instance. A relationship then becomes a collection of such associations having the same reason.

Note that the reason is an important part of the relationship. Just identifying the pair of objects involved is not enough; several different relationships can exist among the same objects. Thus, if the same person is your brother, your manager, and your teacher, these are instances of three different relationships between you and him.

4.1 Degree, Domain, and Role

We have so far looked only at relationship instances involving two things. They can also be of higher "degree". If a certain supplier ships a certain part to a certain warehouse, then that is an instance of a relationship of degree three. If that supplier uses a certain trucking company to ship that part to that warehouse, then we have a fourth degree relationship.

We must distinguish between "degree" and a confusingly similar notion. If a department employs four people, we might view that as an association among five things. If another department employs two people, we have an association among three things, and we couldn't say in general that the "employs" relationship has any particular degree.

We proceed out of this dilemma in several steps. As a first approximation, think of a relationship (not an instance) as a pattern, given as a sequence of categories (e.g., departments and employees). An instance of such a relationship then includes one thing from each category (i.e., one department and one employee). The degree of such a relationship would then be the number of categories in the defining pattern. What we have done is to reduce the "employs" relationship from being an association between one department and all of its employees to being an association between one department and one of its employees. Although the former is certainly a legitimate relationship, it is difficult to subject it to any definitional discipline. We will only deal with relationships in the latter form.

It is also possible to think of the relationship between a department and all its employees as a relationship between two things, where the second thing is the set of employees in the department. This introduces a new construct, namely the *set* of employees as a single object, and the relationship is now indirect: employees belong to the set, and the set is related to the department. We will not pursue this alternative.

Specifying the pattern of a relationship as a sequence of categories is sometimes too restrictive. There are many relationships that permit several categories to occur at the same "position", as is the case when one can "own" many kinds of

things. We therefore introduce the term "domain" to designate all the things that may occur at a given position in the relationship. A domain may include several categories. Thus we might describe an "owns" relationship as having two domains, with the first domain including such categories as employees, departments, and divisions, while the second domain included such categories as furniture, vehicles, stationery supplies, computers, etc.

"Domain" and "category" could be treated as the same concept if (1) we are dealing with a system which permits overlapping categories, e.g., unions and subsets; (2) the system does not impose intolerable performance or storage penalties for maintaining many declared categories; and (3) it doesn't bother our intuitions to think of all owners of things as a single kind of entity, and all owned things as another single kind.

One final improvement in the specification of relationships makes the specification more informative and less formally structured. Instead of assigning a domain to a sequential position in a pattern, we can give it a unique "role" name describing its function in the relationship, such as "owner" and "owned". Thus a relationship can be specified as an unordered set (rather than a sequential pattern) of unique role names. The number of role names is the degree of the relationship. A domain is specified for each role.

Role names are especially useful when several roles draw from the same domain. A "manages" relationship would be defined over the roles "manager" and "managed", both drawing from the domain of employees.

4.2 Forms of Binary Relationships

Much of the information in an information system is about relationships. However, most data models (e.g., the relational model, IMS hierarchies, DBTG networks) do not provide a direct way to describe such relationships, but provide instead a variety of representational techniques (record formats, data structures). Implicit in most of these, and in the accompanying restrictions in the data processing system, is the ability to support

some forms of relationships very well, some rather clumsily, and some not at all.

In order to assess the capabilities of a data model, it would help to have some systematic understanding of the various forms of relationships that can occur in real information. In the next few paragraphs I will discuss some significant characteristics of relationships. A particular "form" of a relationship is then some combination of these characteristics. A method for assessing a data model would include a determination of which forms it supported well, poorly, or not at all. Note the emphasis on combinations. In most data models you can probably manage to find a way to obtain most of the following features, taken one at a time. The challenge is to support relationships having various combinations of these features.

By "support", I mean that

- the system somehow permits a constraint to be asserted for the relationship (e.g., that it is one-to-many), and
- the system thereafter enforces the constraint (e.g., will not allow the recording of an employee's assignment to more than one department at a time).

Such support is often implicit in the data structure (e.g., hierarchy), rather than being declared explicitly.

The set of characteristics listed below is probably incomplete—I imagine it will always be possible to think of additional relevant criteria. For simplicity, we are now only considering "binary" relationships, i.e., those of degree two. Most of the concepts can be readily generalized to "n-ary" relationships (those of any degree).

4.2.1 Complexity

Relationships might be one-to-one (departments and managers, monogamous husbands and wives), one-to-many (departments and employees), or many-to-many (students and classes, parts and warehouses, parts assemblies). The relationship between employees and their *current* departments is

(typically) one-to-many, whereas the relationship between employees and all the departments they have worked in (as recorded in personnel history files) is many-to-many.

Another way to characterize complexity is to describe each direction of the relationship separately as simple (mapping one element to one) or complex (mapping one element to many). The terms “singular” and “multiple” are also used. Thus “manager of department” is simple in both directions; “manager of employee” is simple in one direction and complex in the other. Relative to the number of “forms” of relationships, this would count as four possibilities, since a given relationship might be simple or complex in each direction.

One advantage to this latter view is that it corresponds well with certain aspects of data extraction. Very often a relationship is being traversed in one direction (e.g., find the department of a given employee); the data processing system usually has to anticipate whether the result will contain one element or many (e.g., whether an employee might be in more than one department). The complexity of the reverse direction is of little concern (i.e., whether or not there are also other employees in the department).

Thus, if a given direction is complex, it doesn’t matter much whether the relationship is 1:n or m:n. If the direction is simple, the distinction between n:1 and 1:1 may be immaterial.

It’s amusing to note that the relationship between postal zip codes and states in the USA is *almost* many-to-one, so that the zip code directory is organized hierarchically as zip codes within states. The relationship is really many-to-many, but there are only about four zip codes that actually span state boundaries. The post office copes with that by listing the exceptions at the front of the directory.

4.2.2 Category Constraints

Either side of a binary relation might be constrained to a single category, constrained to any of several specified categories, or unconstrained (three possibilities on each side, for a total of nine combinations). Constraint to a single category is

probably the most common situation, as in the examples above under "Complexity".

Constraint to a set of categories occurs, for example, when a person can "own" things in several different categories, or when the owner might be a person, department, division, company, agency, or school. This case might be avoided by defining one new category as the union of the others — if you're dealing with a data model which permits overlapping categories.

It is hard to think of a relationship that is naturally unconstrained as to category (i.e., one that applies to every kind of thing), but it often makes sense to handle a relationship that way in a real data processing system. Perhaps the relationship does happen to apply to all of the things represented in this particular database, or to so many of them that it isn't worth checking for the few exceptions. Perhaps the installation doesn't want to incur the overhead of enforcing the constraint, and trusts the applications to assert only sensible relationships. Or, the system simply may not provide any mechanism for asserting and enforcing such constraints.

4.2.3 Self-Relation

Three possibilities:

- The relationship is not meaningful between things in the same category.
- Things in the same category may be so related, but a thing may not be related to itself.
- Things may be related to themselves.

The first case is again probably the most common. The second occurs, for example, in organization charts and parts assemblies. Examples of the third are our representatives in government (the representative is one of his own constituents), and canvassers for fund drives (the canvasser collects from himself).

Incidentally, I am thinking here of the simple case where categories are mutually exclusive. When categories overlap, as in subsets, things may be more complicated.

4.2.4 Optionality

On either side of the binary relationship, the relationship might be optional (not everybody is married) or mandatory (every employee must have a department). I will count this as four combinations (two possibilities on each side), although there could conceivably be more: one of the domains may include several categories, with the relationship being optional in some categories and mandatory in others.

4.2.5 The Number of Forms

Even with this limited list of characteristics, we already have 432 forms (4 x 9 x 3 x 4). This number might include some symmetries, duplicates, and meaningless combinations, but after subtracting these we still have a sizable checklist.

4.2.6 Multiplicity of Relationships

Another important criterion concerns whether and how the system permits more than one relationship over the same pair of domains. The nature of the support often varies according to the forms of the relationships involved.

4.2.7 Examples

A sampling of a few forms and how they are handled in some data models.

For instance: certain cases can only be implemented in IMS using logical relationships (intersection records), e.g., self relation, or multiple relationships over the same domains. These cases cannot also be constrained to be one-to-many relationships, since they are no longer part of a single hierarchic structure.

Consequently, that most elementary of structures, the homogeneous hierarchy (like an organization chart), cannot be represented in IMS (or DBTG, for similar reasons).

Also, there is no way to enforce a one-to-one relationship in IMS, except by representing both entities within the same segment. Then it becomes difficult to change the relationship, or to make it optional — and you can't have an application that looks at one entity without the other.

Some systems (e.g., [IMS], [DBTG]) require 1:n relationships to have only a single category on the "parent" or "owner" side of the relationship. Consider a 1:n relationship that naturally has parents in several categories (e.g., suppose that items of capital equipment may be owned by departments or divisions, but not both). It is sometimes suggested that such a relationship can be modeled as the composition of several 1:n relationships, one for each parent category (e.g., one relationship for things owned by departments, and another for things owned by divisions). This doesn't usually work, however, because it is difficult to prevent an item from having a parent in *each* of these relationships (i.e., that data structure would erroneously permit a piece of equipment to be owned by a department and a division at the same time). Furthermore, this approach creates an unnatural situation by replacing one natural relationship with two artificial ones. One can no longer ask "Who owns this equipment?" One now has to engage in a stilted dialogue: "Which department owns this equipment? None? Oh, then which division owns it?"

4.3 Other Characteristics

There are a number of other characteristics of relationships that might be worth describing to an information system. (We are still looking only at binary relationships.)

4.3.1 Transitivity

For some relationships, if X is related to Y and Y is related to Z, then X is automatically related to Z. This is true of ordering

relationships (less than, greater than) and equivalence relationships (equal to, has same manager as). This characteristic is only meaningful when both domains of the relationship include the same category.

4.3.2 Symmetry

For some relationships, X being related to Y implies that Y has the same relationship to X. This is true of equivalence relationships and also, for example, "is married to". (In the latter case, both domains are "people". The relationship "is the husband of" between the categories of men and women is not symmetric.) Again, symmetry is only meaningful when both domains include the same category.

It's worth confessing that purely symmetric relationships only fit awkwardly into this general structure of relationships. In the first place, the two roles (as well as the two domains) are identical. In the "is married to" relationship, the role on both sides is "spouse", just as the domain on both sides was "people". Thus we can no longer equate "degree" with the number of (distinct) roles. Also the "pattern" notion we used earlier doesn't fit quite as neatly. That was based on a concept of ordered pairs, where each position had some significance. Here we are really dealing with unordered pairs; the information is identically the same no matter which way the pieces are ordered. Saying that A and B are married is identically the same as saying that B and A are married. Few systems really support symmetric relationships; any that do are likely to require both pairs to occur, even though they are redundant.

Another difficulty is that the concept of "degree" is less clear. If the relationship is not limited to being between two people (and "sibling" isn't), then we can't really appeal to an intuitive notion of "pattern" to establish the notion of degree. The relationship might more naturally be viewed as one of varying degree, depending on the number of siblings in a given family. Nonetheless, we find it much more convenient to consider the relationships between people two at a time, and regularize this as a binary relationship.

4.3.3 Anti-symmetry

For some relationships, if X is related to Y, then Y cannot have the same relationship to X. Examples include "is manager of", "is parent of", and total orderings. ("Less than or equal" is a *partial* ordering, which permits some symmetries; "less than" is a *total* ordering, which is anti-symmetric.)

4.3.4 Implication (Composition)

A relationship may be defined as the composition of two others, i.e., the occurrence of two relationships implies a third. For example, if an employee works for a certain department and that department is in a certain division, then that employee belongs to that division. Or, one relationship may imply another: "is the husband of" implies an "is the wife of" relationship. The converse implication may or may not hold.

4.3.5 Consistency (Subset)

A certain kind of consistency between relationships might be obtained by defining one to be a subset of another. For example, the relationship between employees and their current departments is a subset of the relationship between employees and all their departments, as recorded in the personnel history file.

4.3.6 Restrictions

A variety of restrictions might be specified (cf. [Eswaran], [Hammer]). There may be a limit on the number of things that one thing can be related to (maximum department size). One relationship might require another to be true (an employee's manager must be in the same division, or must have a higher salary). It may be invalid to "close a path" (i.e., a part can't be a component of any of its sub-assemblies).

4.3.7 Attributes and Relationships of Relationships

An instance of a relationship might have attributes of its own, such as when it was established (date of assignment to department). And it can itself be related to other things. This will come up again later.

4.3.8 Names

Relationships have names, and could be subject to the general variety of naming conventions.

The names of relationships comprise valid information. An information system should be able to answer questions like:

- What relationships exist between x and y?
- In what relationships is x involved?

4.4 Naming Conventions

I tend to use one convention for naming relationships, but there are several others in use as well, and each of them seems to be more natural in certain cases. The conventions involve the use of zero, one, or two names for the relationship.

4.4.1 No Name

If we are speaking of an employee and mention "department", it can be recognized as a reference to the department to which the employee is assigned.

The convention is that, from a given entity, one traverses a relationship (selects a path) by naming the domain at the other end.

That works whenever (1) the relationship is binary, (2) the two domains are distinct, and (3) there is only one relationship between those two domains, or there is a convention for selecting one of them as a default.

This convention could be viewed as a degenerate form of the two-name convention (below), where each path has a name

derived from the target domain. E.g., “department of” is the name of the path from employee to department.

4.4.2 One Name

The relationship may be given a single, neutral name such as “assignment” or “inventory”. If we want to find a person’s department, we ask about “assignment of person”; if we want to find the people in a department, we ask about “assignment of department”.

This is a common convention, and one that I tend to use, but it doesn’t correspond well with most of our language habits; we usually tend to use different words for the two directions of the relationship. Furthermore, if the two domains are the same, then the convention only works if the role names are mentioned.

This convention could include the no-name case, if a defaulting mechanism were provided whenever the relationship name was omitted.

This convention is the one that extends best to n-ary relationships. Instead of getting involved with all the combinations of pairwise directions between the domains, one simply names the relationship and values in some set of domains, and expects as an answer all the combinations of values which exist with them in the other domains. For example, if a ternary relationship between parts, warehouses, and suppliers is given a single name such as INVENTORY, then questions can be written symmetrically using a form such as

INVENTORY (PART=PIN, WAREHOUSE=WEST,
SUPPLIER=?)

or

INVENTORY (SUPPLIER=?, PART=?,
WAREHOUSE=WEST)

4.4.3 Two Names

A binary relationship can be traversed in two directions, and each is sometimes given its own name. (The two directions are sometimes described as two distinct paths.)

A hybrid between one and two names consists of the practice of giving the relationship one name, but requiring that it be modified in some way to indicate direction (e.g., by prefixing a minus sign for the direction considered to be "reverse").

This convention could eliminate the need for role names, but it does not extend well to n-ary relationships.

4.5 Relationships and Instances Are Entities

Instances of relationships are things themselves, about which we may have information in the system.

They have attributes. Just as you have an age, so does your association with your department, your spouse, and your car. For a given part (type) stocked in a given warehouse, there is a certain quantity on hand.

They can be related to other things. The storing of a certain part in a certain warehouse is approved by a certain manager.

Instances of relationships can be related to each other (illustrated in section 10.2).

Instances of a relationship can be identified (named) by identifying the relationship and the entities being related: "employed-in, John Jones, Accounting". In real systems these composite names of instances are usually not represented explicitly, but are implied by the definition and organization of the records in a file. A record in an employee file will explicitly contain "John Jones" and "Accounting"; "employed-in" is implicitly understood by users of the file, or may be factored out into a file or record description somewhere. (Are there real examples where an instance of a relationship has a single simple name of its own, rather than a composite name?)

4.6 "Computed" Relationships

We've talked so far about relationships that get established and broken, which we might naturally visualize in terms of links between objects.

There are other kinds. Some are permanent, and are detected by some sort of computational process rather than by traversing links. And there is an enormous number of non-permanent relationships that are not represented by direct links either.

Every computable procedure represents relationships between its inputs and its outputs. The correspondences between angles and their sines can be computed, and so can the correspondences between diameters and areas of circles.

Perhaps the simplest such "computed" relationships are the orderings. The fact that one employee earns more than another (clearly a relationship between the two) is determined by performing comparisons, not by traversing links. We aren't going to think in terms of explicit links between each employee and everyone who earns more than he does.

We might be tempted to dismiss computed relationships as an essentially different kind of phenomenon. After all, why should we get all tangled up between linkages and computations? They just don't feel like the same sort of thing at all. But consider:

- The way we talk about the two is not all that different: "list all the employees *who are assigned to* this department", and "list all the departments *which have smaller budgets than* this department".
- We're never quite sure that there really aren't explicit links lurking under the covers. Sometimes there really are tables that get looked up, instead of performing computations. Do trig tables, log tables, tax tables, etc. look all that different from intersection records?

Another kind of non-explicit relationship exists in enormous quantities. These are all the implied, computed, compound, composite, derived (I use these all synonymously for the

moment) relationships which arise out of the existence of other relationships. We can't avoid them. As soon as any two things are each related in any way to a third thing in common, they have some relationship to each other. Some of these are meaningful and interesting to us, others are not. If your father and my brother are the same person, then I am your uncle. If your birthdate and mine are the same, we are of the same age. If your were born on the same date that I visited my grandmother, then that too is a relationship — but who cares. (But then, who cares that we are the same age? Both relationships are there, whether we care or not.)

If some such relationships are of special interest to us, then it makes sense to name them, and to define how they arise. We can define "in-law" as an appropriate composition of "spouse" and "sibling".

Now we have relationships which are not necessarily permanent — they can be established and broken — but they are not represented by direct links.

And then there's another thing that can happen: we might not be sure which were the "direct" relationships and which were the "derived" ones. Then all we can do is to define all of them as direct links, with appropriate specifications of derivation and consistency. Consider this: we always tend to think of "uncle" as derived from "father" and "brother" — but sometimes the only thing we know about two people is that one is the uncle of the other. We don't know yet who their common relative is (we might or might not care), but we'd like to record their relationship directly. On the other hand, if we know that other people have a common brother and father, then we want "uncle" derived for us. And when a whole bunch of such relationships might be asserted, we might like the system to perform some reasonableness checks for us.

Ideally, we shouldn't have to know much about the way relationships are represented internally, as long as we have a name to refer to them with. However (and you knew there was going to be a however), certain differences in behavioral characteristics are likely to be visible.

- The relationship might not be modifiable. Depending on the implementation, you may or may not be able to say "change the cosine of ninety degrees to .12345".
- The number of instances might not be finite. It might not be possible to list all instances.
- The instances might not be able to have attributes of their own, or be related to other things (except perhaps via other computations).
- They might not be bi-directional. Procedures might not be provided for following the reverse direction.

Such behavioral characteristics of relationships ought to be describable in an information model.

5 Attributes

5.1 Some Ambiguities

Lots of things have lots of attributes. People have heights and birthdays and children, my car is blue, and New York is crowded. Much of the information in an information system records the attributes of things.

But as common as the term "attribute" may be, I don't know what it means. The fact that I've been using the term is totally irrelevant.

The term is used to mean different things at different times, and I have trouble distinguishing the idea from others we've already discussed. Don't be fooled by the fact that I can rattle off a few examples. As you'll see later on, I really think they are examples of something else.

There are several ambiguities in the way the term is used. In order to explain that without getting tangled up in other ambiguities, let me temporarily introduce three new terms, so that we can get a better handle on what we're talking about. Every attribute has a *subject*: what it is an attribute of. People, my car, and New York were the subjects of the attributes in the examples above. Then there are *targets*, which are at the other end of the attribute, such as heights, blue, and crowded. Thirdly, there are *links* between subjects and targets. In the last example, it isn't "New York" or "crowded" which are important in themselves; what is being expressed is a connection between the two: New York *is* crowded.

(Later I'll show that the three new terms are quite imperfect. They still retain two ambiguities: type vs. instance, and thing vs. symbol.)

First ambiguity: "attribute" sometimes means the target, and sometimes the link. "Blue", "salary", "height" are sometimes referred to as attributes. On the other hand, "color of car" and "height of person" are also sometimes called attributes. If you don't make the distinction, you get trapped into believing that a

single construct can represent the idea of "blue" and the set of all things that are blue. If you do make the distinction, then you had better use the term very carefully. About half the people you meet will use it in the opposite sense from you.

I tend to favor using the term "attribute" in the sense of the link itself, between the subject and the target. But I'm not sure I am always consistent in my usage (or that anyone else is).

The second ambiguity has to do with type and instance, and my new terms haven't helped that ambiguity one bit. Some people say that "blue" (or "my car is blue") is an attribute. Others will say that the attributes in this case are "color" (or "color of car"), and that the first two things were "values" (or instances, or occurrences) of the attribute. I have no preference. I tend to use the terms carelessly in either sense. Other people are sometimes careful to define the sense they intend, and sometimes they aren't.

The third ambiguity has to do with thing and symbol, and my new terms didn't help in this respect either. When I explore some definitions of the target part of an attribute, I get the impression (which I can't verify from the definitions given!) that the authors are referring to the representations, e.g., the actual four letter sequence "b-l-u-e", or to the specific character sequence "6 feet". (Terms like "value", or "data item", occur in these definitions, without adequate further definition.) If I were to take that literally, then expressing my height as "72 inches" would be to express a different attribute from "six feet", since the "value" (?) or "data item" (?) is different. And a German describing my car as "blau", or a Frenchman calling it "bleu", would be expressing a different attribute from "my car is blue". Maybe the authors don't really mean that; maybe they really are willing to think of my height as the space between two points, to which many symbols might correspond as representations. But I can't be sure what they intend.

To summarize: any of the following might be an example fitting the concept of "attribute", although each exemplifies a different thing:

- The concept of color.
- The concept of blue.
- One of the character strings “blue”, “bleu”, “blau”, etc.
- The general observation that cars have colors.
- The fact that my car is blue.

Perhaps these ambiguities can be resolved with some careful definitions, and some authors do make a commendable effort. Most definitional efforts I’ve seen, however, leave other crucial terms undefined or ambiguous, so that we don’t really have a working basis for applying the concept.

5.2 Attribute vs. Relationship

I’m really not very concerned about the ambiguities. For me, these problems are overshadowed by a larger concern. I don’t know why we should define “attribute” as a separate construct at all. I can’t tell the difference between attributes and relationships. (The astute reader may have noticed that I have, in two earlier comments, identified both attributes and relationships as constituting the bulk of the information managed in the system.)

The fact that “Henry Jones works in Accounting” has the same structure as the fact that “Henry Jones weighs 175 pounds”. “175 pounds” appears to be the name of an entity in the category of “weights” just as much as “Accounting” is the name of an entity in the category of “departments.” Both facts are relationships between entities. Both facts (relationships) are capable of themselves having attributes: Henry Jones has worked in Accounting since 1970; Henry Jones has weighed 175 pounds since 1970. Both facts are answers to a symmetric set of questions:

- Where does Henry Jones work?
- How much does Henry Jones weigh?
- Who works in Accounting?
- Who weighs 175 pounds?

Both facts can be “traversed” in symmetric fashion to answer questions like:

- Who works in the same department as Henry Jones?
- Who has the same weight as Henry Jones?

Sundgren tries to make the distinction on the basis of whether the target is an object in the system — without defining what that means: “At any point of time every object in [the system] S possesses a set of properties. Some of the properties of an object are local, i.e., they are independent of the existence and properties of other objects in S. Other properties of an object are relational, i.e., they depend upon the object’s relations to other objects in S” [Sundgren 74]. Then he confesses, in the ensuing discussion, that “...there are no formal criteria. However, I am convinced that useful informal rules of thumb can be given. Moreover, it is my experience that it is not a big problem for the user to make a satisfactory intuitive distinction between objects and properties.”

Berild and Nachmens write: “We store information about objects ... of two kinds, namely attributes of an object and an object’s relations to other objects. Note that this distinction between attributes and relations is only of logical interest, as both attributes and relations are stored as associations ...” [Berild].

There really does always seem to be an entity lurking behind the scenes somewhere, to which there separately corresponds an assortment of symbols exhibiting ambiguity and/or synonyms. We just have to learn to think of them properly. To accept the equivalence between attributes and relationships, we may have to acquire new habits of thought. My height really is not the string of characters “6 feet”. A height (or other length) is a certain interval in space (any good reason not to think of it as an entity?); its measurement can be written down in many ways. A day is just that — a day on which you can think of something as having happened; there’s a large assortment of ways to write the dates that are the names of that day. A color is something which

you can see, and maybe has a definition in terms of light wave frequencies; it is not the word “blue”.

Even with numbers, we have to distinguish between the abstract quantity and the various symbols that might represent it. When it comes to measured quantities, there are really two steps from entity to symbol:

1. From entity to abstract number, via a unit of measure. A unit of measure establishes relationships between mass entities and abstract numbers. The rule named “yards” maps my height into a number that is the same as the number of hands I have (which was mapped by a “count” relationship).
2. From abstract number to symbol, via data type, precision, base, notational system, etc. The symbol for my height in yards is “2” in decimal integers, “10” in binary integers, “II” in Roman numerals, and “two” in English words.

The target of an attribute is rarely a symbol directly. There is almost always a target entity distinct from the symbols. There are some notable exceptions to this rule, but then I wouldn’t call the phenomenon an “attribute”. If the target is really a pure symbol, then I prefer to call this “naming” and deal with it in another chapter. (It’s confusing. Some people do prefer to say that name, employee number, and social security number are “attributes” of people. It’s perfectly good jargon, but it does get some underlying distinctions muddled.)

In real practice, of course, dates, heights, managers, departments, etc. do get treated in diverse ways. But rather than classifying that in terms of attributes vs. relationships, I think it is more helpful to distinguish them on the basis of the kinds of existence (and equality) tests employed for the entities involved.

Incidentally, I do share with you the intuitive inclination to distinguish between relationships and attributes. For some facts the term “attribute” seems appropriate, and others seem to be “relationships”. It’s just that I can’t find any really objective distinguishing criteria to support my intuitions consistently.

Sometimes some of us might subconsciously picture it in terms of data records. If a fact is pictured just as a value in a field, we are inclined to call it an attribute, but if it has the effect of linking two records together, then it's a relationship. But that's an unsatisfactory basis for defining the distinction. First of all, we can conjure up many examples running counter to our intuitions. Secondly, the same fact can be represented inside the machine either way at various times. We want to define our basic information constructs in real world terms; the implementation in data processing mechanisms comes after we model the enterprise, not before.

Let me suggest a way to satisfy our intuitions. Let us build a modeling system that only supports one basic linking convention, which we are free to call either "attribute" or "relationship". The terms will be synonymous; we can use whichever one feels better at the moment.

If such a system doesn't satisfy you, I hope that you will tell me exactly what the system should do differently when it sees the terms "attribute" and "relationship".

5.3 Are Attributes Entities?

If one really wanted to develop a rigorous notion of attributes (which I don't), then this is another nasty question to be faced. Intuitively, some might say that attributes aren't themselves entities (regardless of whether one had in mind the links or the targets).

But if you think that relationships are entities, and you can't distinguish attributes from relationships, then where are you left?

And again: do you think that the subject of an attribute is necessarily an entity? I'm inclined to think so. But it turns out that some attributes are themselves the subjects of other attributes (which would make them entities after all). Examine carefully the structure of the following information, which is likely to be found in some databases:

- The percentage of an object's surface which is a given color.
- The date an employee began receiving a certain salary.
- The ages of an employee's children.

Those appear to be attributes of attributes.

And what about dates? They could have attributes, like day of week, or scheduled events. An illustration in [Sharman 75] shows a relation whose columns are month, day of month, and day of week.

5.4 Attribute vs. Category

We can say something *is* a car, and we can say that something *is* red. Intuitively, I feel that the first assertion is about the intrinsic nature of the thing (hence, its category), while the second asserts additional information about its characteristics (i.e., attributes). At one time I wanted to believe in a definable difference between category and attribute, but I didn't know how to articulate it. Some assertions fall in a middle ground ("that is an employee"), diminishing hopes for an effective distinction.

I've abandoned my hope of defining that distinction, too.

5.5 Options

In the area of attributes, just as with the other topics, we can apply the constructs to the data in a number of arbitrary ways, all of which make some sense to some people some time.

We can refine the structure of attributes to varying degrees. We tend to treat hair color as an attribute of a person, although a strict rendition perceives that color is an attribute of hair, which in turn is an entity related to a person. So also with date of hire, which is really the "starting time" attribute of the relationship between an employee and employer. We are often inconsistent, letting date of hire be an attribute of a person in the employee file, while treating it as the attribute of a relationship in the employment history file.

It sometimes makes sense to say that all colored things draw their attribute values from the same "domain". On the other hand, hair colors and car colors may not have many values in common. The list for the existence test may be different in the two cases!

A given set of things might be treated as the names of distinct fields (attributes?), or as the set of allowable values for a single field. We have all seen two kinds of forms for indicating, e.g., marital status. One has a heading "marital status", under which you are expected to fill in "married", "single", etc. The other kind has "married", "single", etc. as *headings* under which you are expected to make some mark (in this case the field values correspond to yes/no).

The same phenomenon might be an attribute, a categorization, or a relationship. Consider a person employed by a certain company:

- In a banking database in which companies are "non-entities", a person's employer is simply an attribute of the customer.
- In that company's database, that person falls in the category of "employees".
- In a more generalized database, "employed by" may be just one of several possible relationships between people and companies. Others might be "stockholder of", "sells to", "is covered by benefits of", etc.

And the view of the phenomenon will often change with time. That is, different perspectives become appropriate as the information processing needs of an enterprise change, and as the scope of interest changes.

Examples:

- If the databases of several companies are merged (e.g., for more efficient payroll processing), then the "employee" entity becomes a "person" entity with an explicit relationship to his company.

- Then, also, date of hire changes from an attribute of an employee to being an attribute of the relationship between person and company.
- When a company starts automating its personnel history records, the relationship between employee and department changes from 1:n to m:n.
- "Address" changes from a simple attribute to a complex one when residence histories are kept.
- Instead of address being an attribute of a person, it could become an attribute of a "place" entity. A "resides" relationship could be introduced between people and places.
- Some states generalize a driver's license into a general-purpose identity card. Then the attribute "is licensed to drive", which was implied for all cardholders in the old construct, now must be made an explicit attribute. Something similar probably happens when social security numbers are extended to serve as taxpayer identification numbers; it may no longer be true that a social security account exists for each of these numbers.

5.6 Conclusion

I will not formally distinguish between attributes and relationships, or between those two and categories. Even so, I do continue to use the terms "attribute" and "category" when they seem more natural, but I won't be able to say why they feel more natural at the time. Most likely, it will correlate well with my implicit assumptions about the existence tests for the entities involved.

6 Types and Categories and Sets

6.1 "Type": A Merging of Ideas

Three ideas seem to have gotten combined:

- The urge to classify things according to "what they are".
- A need to express the semantic characteristics of things, by specifying which attributes and relationships and names are relevant and valid for them. The easiest paradigm: certain rules and constraints apply to certain *classes* of things.
- A tradition of data description, based on record types. These often tend to be identified as the same phenomena. As a result, the concepts of "entity type" and "record type" are held to coincide. And they are often considered to represent a special kind of information, somehow distinct from other kinds.

6.1.1 Guidelines

One common denominator is the notion of grouping. We assume that things can be divided up into groups, where the groups are expected to satisfy a number of guidelines:

1. The groups correspond to our intuitive ideas of what things are, i.e., classification.
2. The groups serve as the scopes over which naming conventions apply. E.g., name syntaxes, uniqueness rules.
3. The groups serve as the scopes over which validity constraints apply.
4. The groups correspond to the domains of relationships.
5. Things don't move from one group to another.

6. The groups are mutually exclusive (nothing belongs to more than one such group) — an enormously bad hangover from the record type heritage, but still required in almost all definitions.

In any discussion of "type", it would be useful to establish which of these guidelines were to be assumed.

6.1.2 Conflicts

Unfortunately, these guidelines are generally quite incompatible.

As we've already seen, notions of "entity categorization" are very variable, subjective, and dependent on local purpose. We have "categories" for which naming conventions aren't uniformly applicable, for which attributes aren't universally applicable.

Some people don't have social security numbers; some don't have maiden names. If a category is defined to be the union of several sub-categories, a rule for one sub-category may not apply to another. Further, we may not want to formally define sub-categories corresponding to the scope of every rule, e.g., just for married female employees.

Books may have "International Standard Book Numbers" (ISBN) and Library of Congress numbers. Some books have both, some have neither, some have one or the other. The category of things covered by Library of Congress numbers includes photographs, movies, tapes, recordings, etc. Those don't get ISBN's.

Record types are probably the only concept to which the guideline of being mutually exclusive is applicable.

I would speculate that for each pair of guidelines in the list above, we could find some example that brought the two into conflict.

6.2 Extended Concepts

6.2.1 Arbitrary Sets

Consider arbitrary groupings: sets defined in terms of things satisfying certain predicates, e.g., having certain relationships to certain things, or Boolean combinations of such conditions. Such conditions could be based on attribute values, relationships to other things, names, etc. It's not clear why "type" is a different idea from these, or which of these is to be thought of as "type". There are some good reasons not to make type quite so distinct.

For example, there should be some way to present categories as properties (e.g., field values) to applications. E.g., if someone is both an employee and a stockholder, then

- An application dealing with stockholder records should be able to see employer name as a field value, and
- An application dealing with employee records should be able to see a field indicating stockholder status.

Conversely, properties may be used to define "apparent" categories to applications, probably as subsets of "real" categories. For example, a new application may want to deal with a file of managers (perhaps with department records also occurring in the file). "Manager" appears to be the category (i.e., file name or record type) to the application, but it is defined to the system as that subset of the "employees" category which has "manager" as the value of the "job" attribute.

6.2.2 General Constraints

We haven't lost sight of the original objective, namely to be able to specify rules about "groups" of things. But now the groups need not be so explicit; we can speak in terms of how we'll recognize the individuals to which the rules apply. Rules and constraints can be generalized to the form: "the following rule applies to all things satisfying a certain predicate", where

the predicate might be “all things having relationship X to object Y”. For traditionalists, X might be “has type” and Y could be “employee”. For set theorists, X would be “is member” and Y would be “employees”. For others, X might be “is employed” and Y might be “IBM”.

It is thus a matter of viewpoint as to whether the fundamental constructs involved here are sets and membership or entities and relationships.

This general form solves the “partial applicability” problem: we can specify that “maiden name” is applicable to all things which are employees of IBM and which are of the female sex and which have been married.

Some of the rules might govern the interaction of the sets themselves: two sets may not overlap (equivalently: if one relationship holds for a given entity than another relationship can’t, and vice versa); one set may be a subset of another (the defining relationship of one implies the other), and so on.

In making types non-exclusive, we come closer to reality — and suffer the penalty of facing more of the complexities of real life. We now have to deal with the interaction of rules that apply to overlapping sets. Sometimes they can get inconsistent: employees might be required to do something stockholders are forbidden to do. It would take some complex analysis to insure that a large set of specified constraints was entirely consistent. But saying that this is a disadvantage of overlapping types is the view of the ostrich. Exclusive sets don’t solve the problem; they avoid it by pretending that employees and stockholders don’t overlap.

Problems of overlap and consistency can even occur with respect to specifying existence and equality tests. This can arise when there is an overlap of “types”, where some members have explicit surrogates and some don’t. Consider a personnel database that has explicit surrogates (records) for employees, but not for dependents. This is fine so long as we are willing to treat employees and dependents as disjoint categories. But suppose we needed to know which dependents are also employees? Dependents are usually listed by name only, and that is not an unambiguous key to the employee file. An employee might have

the same name as someone else's dependent (or his own son!). Various solutions can be devised, none of them elegant.

The same situation can occur in a banking database, in which a client's employer might be a simple attribute (field value). The bank may want to be able to determine whether a client's employer was itself a client of the bank.

6.2.3 Types, If You Want Them

Given a general mechanism for describing sets, one can try to superimpose a notion of "type" by imposing rules such as these:

- Some sets confer naming rules on their members.
- Every object must "belong" to at least one such set (which means that the object has the required relationships to appropriate objects).
- Once a member of such a set, the object may not leave the set, except when the object is deleted from the system.
- Such sets could be called "types", but underneath it all they are still ordinary sets.

If that doesn't satisfy your notion of type, just vary the rules to suit yourself. Then compare notes with your neighbor.

6.3 Sets

6.3.1 Sets and Attributes

Be cautious in equating sets with attributes (this bears on the ambiguities mentioned in chapter 5). For example, we might have an object representing the set of employees in the aggregate, and an object representing the concept of "employee" — and then be tempted to say they are the same object. We might have observed apparent redundancies. Certain patterns of relationships occur in parallel with the two objects: a person has

the attribute of being an "employee" if and only if he is a member of the set of employees. So why keep them apart?

The difficulty is that the concept of "employee" determines more than one set. The set I had in mind consisted of people who are currently employees. (That's what you had in mind too, isn't it?) But people can be related to the concept in many ways. There are people who have been employees, or are eligible to become, or have applied to be, or have pretended to be, or refused to be, and so on, together with various combinations of these sets which yield new sets. For other kinds of concepts, other relationships might also be relevant, such as "partly", or "almost", or "occasionally".

A set is determined by a predicate, whose minimal form involves a relationship to an object: the set of things having relationship X to object Y. One should not presume that the object Y determines a single set.

6.3.2 Type vs. Population (Intension vs. Extension)

A "type" is sometimes referred to as a set of occurrences (e.g., the type "employee" consists of the set of employees). This is all right as an informal concept, but several precautions ought to be observed [Durchholz].

There are two distinct notions of "set" involved here. There is the abstract idea of what the type is (e.g., the idea of "employee"), and the current population of people who happen to be employees at the moment. The former is the "intension" of the set, and the latter is its "extension". The latter tends to change often (as people get hired and fired), but the former doesn't.

Very simply, a "type" corresponds to the intension of a set, not its extension. The concept of "employee" isn't altered by hiring and firing people.

Incidentally, one ought to be very cautious about claims of various models being based on "the axioms of traditional set theory". That set theory deals entirely with extensional sets: a set is determined entirely by its population. There is simply no notion of a set with changing population; each different

population constitutes a different set. So, the relevance of such set theory to any model of data processing is, at the very least, questionable.

Another caution has to do with emptiness. The concept of "employee" continues to exist even if there are no employees. One oughtn't think that the concept has disappeared just because no space is occupied by employee records.

Again, this simply amounts to distinguishing the intension and extension of the set. And, to those familiar with set theory, it corresponds to the existence of an empty set (i.e., the set, though empty, does itself exist).

As a consequence of its extensional foundation, traditional set theory holds that there is exactly one empty set. In fact, this provides the set theoretic base for number theory: the empty set is the definition of the concept of "one". Thus, if all employees are fired, then the set of employees is the same as the set of unicorns. Not two equivalent sets, but one single solitary set to which we may give several names. Again, this doesn't correspond to our data processing models: "employee" and "unicorn" are always two distinct types, or concepts.

This distinction between extension and intension affirms that a type (set) is a distinct entity from any of its members. One could perceive set membership, or type membership, in terms of a relationships between pairs of entities: set objects and member objects.

6.3.3 Representation of Sets

Sets need not be introduced as primitive kinds of objects. They can be generalized into objects and relationships. "Belonging" can be a relationship between an arbitrary object and an object representing a set; "subset" can be a relationship between two objects representing sets. With a strong enough capability for implication and constraint on relationships (cf. section 4.6), the behavior of sets can be modeled. E.g., we can specify a derived relationship: X "belonging" to Y and Y being "subset" of Z generates X "belonging" to Z.

Thus, the basic mechanism of objects and relationships seems adequate to cover the phenomena of types and sets. It's useful, too, because types and sets share many of the characteristics of common objects. They have names (and perhaps aliases), and attributes (creation date, number of members), and relationships with other things: they are subsets of one another, people are responsible for maintaining them, they are governed by constraints, etc.

7 Models

7.1 General Concept of Models

We return now to the domain of computerized information systems. The bridge that gets us back is the "data model". It is a bridge in the sense that data models are techniques for representing information, and are at the same time sufficiently structured and simplistic as to fit well into computer technology.

We are always in trouble with words. The term "model" is so over-used as to be absurd. Out of the whole complex of meanings it might have, the following is what I have in mind at the moment.

A model is a basic system of constructs used in describing reality. It reflects a person's deepest assumptions regarding the elementary essence of things. It may be called a "world view". It provides the building blocks, the vocabulary that pervades all of a person's descriptions. In the broad arena of human thought, some alternative models might be composed of physical objects and motion, or of events seen statically in a time-space continuum, or of the interactions of mystical or spiritual forces, and so on.

A model is more than a passive medium for recording our view of reality. It shapes that view, and limits our perceptions. If a mind is committed to a certain model, then it will perform amazing feats of distortion to see things structured that way, and it will simply be blind to the things which don't fit that structure.

Some linguists have been telling us that for a while. "...language defines experience for us .... because of our unconscious projection of its implicit expectations into the field of experience... Categories such as number, gender, case, tense, mode, voice, aspect, and a host of others ... are not so much discovered in experience as imposed upon it" [Sapir]. We'll come back to that in section 11.8.5.

In much narrower terms, the data processing community has evolved a number of models in which to express descriptions of reality. These models are highly structured, rigid, and simplistic, being amenable to economic processing by computer. These models include such things as files of records, tabular structures, graphs (networks) of lines connecting points, hierarchies (tree structures), and sets.

Some members of that community have been so overwhelmed by the success of a certain technology for processing data that they have confused this technology with the natural semantics of information. They have forgotten any other way to think of information except as regimented hordes of rigidly structured data codes — in short, the mentality of the punched card.

7.2 The Conceptual Model: Sooner, or Later?

All the problems touched on in this book converge on the conceptual model (cf. section 2.2.2). It is in this medium that all the things an enterprise deals with must be reduced to crisply structured descriptions.

The conceptual model will be a very real computer-related construct, just like a program or a data file. An enterprise is going to have a large amount of time, effort, and money invested in the conceptual model.

There is the learning investment. In spite of our best efforts, any formalism we adopt as the basis of the conceptual model will still be an artificial structure. The concepts will not be perfectly intuitive to anyone; the rules, limitations, and idiosyncrasies will have to be learned. There will be a formal language to be learned, as well as operating procedures. (Interactive facilities and other design aids may help — after the bugs get ironed out — but even their use has to be learned.)

Then comes the actual modeling effort. A lot of energy will go into forcing a fit between the model and the enterprise. The correspondences won't always be obvious; there will be lots of alternatives, and it will take some iterations to recognize the best

choices. Sometimes it will take a flash of insight to perceive the real world in a new way, which better fits the model. Sometimes the enterprise itself will be altered to fit the model. (It's not unusual for a company to adopt a whole new part numbering scheme before automating their inventory control.) This is all accompanied by the gargantuan task of simply collecting and coordinating a mountainous heap of descriptions. "Many corporations will be carrying out the lengthy job over the next 10 years of defining the thousands of data-item types they use and constructing, step by step, suitable schemas from which their databases will be built. The description of this large quantity of data will be an arduous task involving much argument between different interested parties. Eventually the massive databases that develop will become one of the corporation's major assets" [Martin].

The end result will be a physically large volume of information. "It must be emphasized .... that the conceptual schema is a real and tangible item made most explicit in machine readable form, couched in some well defined and potentially standardizable language" [ANSI]. Think of it in the same orders of magnitude as a program library, or a system catalog, or a payroll file. Think of cylinders of disk space, and printouts many inches thick. Think of a small army of technical personnel who have been indoctrinated in a particular way of conceptualizing data, and who have mastered the intricacies of a new language and the attendant operational procedures.

All this time, manpower, and money will be invested by customers in any conceptual model supported in a major system. We had better be very careful about the architecture of the first one. Any attempt to replace it with a better one later will threaten that investment; customers won't accept the replacement any faster than they now accept a major new programming language, or a new operating system. And the replacements will forever be hamstrung by compatibility and migration requirements.

Unfortunately, there are some natural forces which work against our getting it right the first time.

We are just entering a transitional phase in data description. The idea of having three levels of data description (i.e., including

a conceptual model) has been much researched and written about ([ANSI], [GUIDE-SHARE]), but it hasn't yet taken serious hold in any significant commercial systems. It's still on the horizon; it's an idea whose time is just about to come. (I hope I won't still be saying that ten years from now.)

The builders and users of today's commercial systems quite justifiably want to avoid cluttering their systems with anything that might impair efficiency and productivity. The argument that this new approach will make the overall management of data more productive in the long run has yet to be convincingly demonstrated to them.

It is quite understandable that the first steps they take in that direction are small steps. They will first accept data dictionaries that are off-line, not interfering with the productive flow of their systems. They will first accept dictionaries formulated in terms of data items and records, which are the objects they can directly observe proliferating in their systems, and which are most visibly in need of management.

The need for a more sophisticated descriptive model will only gradually achieve general recognition. It will come from the headaches of trying to crunch together the diverse record formats and data structures used by growing families of applications operating on the same integrated database. The nonsense of trying to reflect all their record formats in the conceptual model, while still pretending that the conceptual model describes the entities of the enterprise, will become apparent.

The need for a more sophisticated approach to data description will also grow as the interfaces of the data systems expand to involve more people who are not trained in computer disciplines. Such people will be involved both as end users and as managers of the information resource. Someday there will be a general recognition of what it means, and what it's worth, to model entities and relationships instead of data items and records. I hope that recognition won't come too late.

7.3 Models of Reality vs. Models of Data

One thing we ought to have clear in our minds at the outset of a modeling endeavor is whether we are intent on describing a portion of "reality" (some human enterprise), or a data processing activity.

Most models describe data processing activities, not human enterprises.

They pretend to describe entity types, but the vocabulary is from data processing: fields, data items, values. Naming rules don't reflect the conventions we use for naming people and things; they reflect instead techniques for locating records in files (cf. [Stamper 77]).

Failure to make the distinction leads to confusion regarding the roles of symbols in the representation of entities (sections 2.4, 3.8, 3.9, 8.8), and some mixed ideas of "domain" (sections 2.4, 9.1).

7.3.1 Semiotics

The relevance of semiotics (a branch of philosophy dealing with the theory of signs) to data processing has been stressed by such authors as Zemanek and Stamper. It is a natural connection, since a computer deals only with the signs which represent things, and not with the things themselves.

Some authors equate signs with information, defining semiotics as "the theory of information (or of signs and signals)", and then further defining information as "the output of a mapping process, in the form of analog or digital signs or signals" [Tully].

This approach is tempting because it deals with quantifiable, measurable things. It lends itself to manageable theories, testable hypotheses, probability theorems, and other impressive mathematical paraphernalia. It lets one compute how many redundant bits have to be sent down a noisy channel to achieve a certain probability of correct reception at the other end. It doesn't ask what those bits might mean to anybody.

It is a powerful ally to the approach to information that is founded on records and data items. The focus is on recorded symbols rather than on what the symbols might represent.

This interpretation of semiotics is the wrong approach for the conceptual model. As its name implies, the conceptual model should describe concepts, not signals. Signals and concepts correspond very poorly. To illustrate: I consider a person's weight to be a single piece of information. Depending on the units and precision of measurement, this information can have many representations. Since each representation comprises a different "sign" or "signal", a narrow semiotic approach to information would treat each representation as a distinct piece of information. What is needed here is a concept of "equivalence classes" of signals which all convey the same meaning. Information as a concept would then correspond to these equivalence classes themselves, rather than to the signals contained in the classes.

Conversely, such a narrow approach to information would fail to deal with the fundamental problem of ambiguity, wherein the same signal may convey different meanings in different circumstances. In general, we might have a many-to-many relationship between signals and concepts.

Tully himself unwittingly encounters the duality between information as signs and as concepts when he says "*information* is a mapping or model of something else (which could be an object, or an event, or some other *information*)" (my emphasis).

Computers do deal only in signs, and a database is only a collection of signs. From these premises one can easily — and incorrectly — conclude that the conceptual model describes a collection of signs.

The models themselves are ensembles of signs, and in particular the signs designate *sets* of things (the external and internal models contain signs describing sets of records and fields, not individual instances). The external and internal models are appropriately constrained to describe sets of signs, since what is being described there consists of things that can be processed by programs and stored in devices.

But the conceptual model need not suffer this restriction. There is no reason why we can't introduce signs naming sets of *things* ("employees", "departments") distinct from signs for the sets of signs which name such things ("employee numbers", "social security numbers", "department codes", "department names").

With such separations, we can more clearly approach a semantic bridge to reality in the conceptual model, by explicitly relating sets of signs to sets of things.

7.4 Current Models

7.4.1 Four Popular Models

For a long time the only model for processing data looked like a file of punched cards. The record model is based on such card images.

Three other models are gaining popularity, being in various stages of acceptance in the data processing community. These are the hierarchical, relational, and network models. The hierarchical model is well established in commercial usage, e.g., in IMS. The network and relational models seem to be the main alternatives which designers of new systems are expected to consider. A number of commercial or prototype systems are based on one or the other of these two. The March 1976 issue of Computing Surveys was devoted to expositions of the hierarchical, network, and relational models.

None of these models departs very radically from the record model. Records are very much apparent as the nucleus of all three. So, chapter 8 will deal extensively with the record model and all its pervasive implications. Chapter 9 will then briefly comment on the other three.

7.4.2 An Ironic Ambiguity

I will comment on the other three models in chapter 9, but I will not describe them. My neglect in this regard might be

attributable to laziness, but I have really been avoiding it because of a supreme irony: the models themselves are ambiguous. Each model can be viewed in many ways, and means different things to different people.

The following is a *partial* list of the factors that might be considered in describing and comparing such models:

- There is an "idealized" data structure, e.g., hierarchy or graph.
- The variations on such idealized structures implied by the definitions of systems such as IMS and DBTG.
- Further variations in such structures occurring in various actual implementations (and versions, etc.).
- Various methods for the internal implementation of such structures.
- An idealized set of operations that might be associated with the structure (e.g., "get parent").
- The actual semantics of manipulation embodied in the definitions of IMS and DBTG.
- An assortment of languages in which these semantics might be embedded, at various levels of human factoring.
- Things in these languages that have nothing to do with the basic data structures, e.g., currency (position) management.
- Variations in all the various implementations of all these languages.
- And you might evaluate all of these differently when considering them for the external, internal, or conceptual models.

We often fix on some set of these characteristics as "essential" to a model, with the rest being cosmetic variations that don't really matter. The trouble is, each of us is likely to fix on a slightly different set of essentials. Unless the underlying assumptions are very carefully exposed, many debates about these models are in danger of comparing apples and oranges.

Of course, we are just being haunted again by the perverse subjectivity of perceived reality. Show three people a DBTG set and one will see a named relationship among things, another will see a set of records, and the third will see a ring of pointers through which users have to navigate.

I haven't been able to decide which view is best to take for explaining these models.

7.4.3 Graph Structured Models

There is an increasingly visible trend away from record oriented data models toward models that might generally be called semantic nets, or graph structured models. The trend is highly visible, that is, everywhere except in current commercial data processing. "Information systems technology has relied solely on fact representations which arose from card and tape media. These representations will have continued utility both at the user interfaces for human efficiency in transaction specification and at the interface to physical media for computer efficiency, but to provide further improvements in data independence we will probably have to supplement them with a compatible, more neutral form of fact representation at the system interface. We may gain clues about the form of this supplement from the freer form work on mathematics and linguistics" [Senko 75b].

The trend is visible in binary relation models such as those of Abrial and Senko. It is visible in the "idea structures" of [Griffith]. Graph structures almost invariably emerge when the primary objective is the representation of information, rather than data processing ([Bell], [Bobrow], [Heidorn], [Schank], [Shapiro]). Various efforts to provide a semantic layer around the relational model have led to graph structures ([Sowa], [Roussopoulos], [Schmid], [Sharman]).

A description of such models is beyond the scope of this book. One good place to start general research might be [Kerschberg 76a].

One might think, by the way, that the term "network" also refers to such models. Unfortunately, as it is currently used, it

does not. The term “network model” means something entirely different, as we shall see in section 9.3.

8 The Record Model

Records provide a very efficient basis for processing data. They enable us to map out very regular storage structures. They make it easy to write iterative programs for processing large volumes of data. They make it easy to partition data into convenient units for moving around, locking up, creating, destroying, etc.

In short, record technology reflects our attempt to find efficient ways to process data. It does not reflect the natural structure of information. Senko refers to "a major commitment to particular restrictive representations like the arrays of scientific computation, the extensional aspects of set notations, the n-tuples of relations, the cards, records, files, or data sets of commercial systems and the static categories of natural language grammars. Each of these representations has great merit for its original area of study, and in turn it has made major contributions to the study of information systems. Nonetheless, each provides only an approximate fit to the evolving, heterogeneous, interconnected information structures required to model real world enterprises" [Senko 75b]. Sowa observes: "Historically, database systems evolved as generalized access methods. They addressed the narrow issue of enabling independent programs to cooperate in accessing the same data. As a result, most database systems emphasize the questions of how data may be stored or accessed, but they ignore the questions of what the data means to the people who use it or how it relates to the overall operations of a business enterprise" [Sowa 76].

Record technology is such an ingrained habit of thought that most of us fail to see the limitations it forces on us. It didn't matter much in the past, because our real business was record processing almost by definition. But we want to approach the conceptual model a little differently. We want it to reflect information, rather than data processing technology. When different applications deal with the same information using

different record technologies, those differences shouldn't clutter up the conceptual model. (And we might want to consider the possibility of future data technologies that are not so record oriented.)

When I use the term "record", I have in mind a fixed linear sequence of field values, conforming to a static description contained in catalogs and in programs. A record description consists largely of a sequence of field descriptions, each specifying a field name, length, and data type. Each such record description determines one record type.

One field (sometimes a combination of several fields) is often designated as the key, whose values uniquely distinguish and identify occurrences of this type of record.

As far as the system is concerned, a field name signifies a space in the record occupied by data in a certain representation. Any other semantic significance of the field name is perceived only by the user.

Some record formats allow a certain variability by permitting a named field or group of fields to occur a variable number of times within a record (i.e., as a list of values or sets of values). I will use the term *normalized system* to refer to systems that do not permit repeating groups or fields. This follows from the relational model, which excludes such repetitions via its normalization requirements (specifically, first normal form; [Codd 70], [Kent 73]).

8.1 Semantic Implications

Much of the meaning of a record is supplied by the mind of the user, who intuits many real world implications that "naturally" follow from the data. Quite often these implications are buried in the procedures encoded in specific application programs written to process the records. But if we strip away such inferred interpretations, and look only at the semantics that inherently reside in the record construct, we find the following presumptions about the nature of information:

- Any thing has exactly one type — because a record has exactly one record type. We are not prepared for multiple answers to “What kind of thing is that?”.
- All things of the same type have exactly the same naming conventions and the same kinds of attributes — because all records of the same type have the same fields.
- The kinds of names and attributes applicable to an entity are always predictable and don’t change much — because our systems presume stable record descriptions in the catalog or dictionary, and because we’ve learned that it’s traumatic to reformat a file of records.
- There is a natural and necessary distinction between data and data descriptions. We are accustomed to having record descriptions in catalogs, and in programs, quite separate and different from data files.
- In particular, the name of the relationship occurring between two entities is not information, since it doesn’t occur in the data file. For that matter, neither does the type of an entity (i.e., the contents of a record don’t tell us that the thing represented in a certain field is an “employee”).
- A record, being the unit of creation and destruction, naturally represents one entity. Anything not represented by a record is not an entity.
- Such entities are the only things about which we have data. The key field of a record identifies one such entity; all other fields provide information about that entity, and not about any other entity. (This is the fundamental information structure implied by the format of a single record.)
- All entities have unique identifiers. Or at the very least, all entities are distinguishable from each other. I.e., for any two entities, we must know some fact that is different about them, which we can use to tell them apart. (Some systems require records to have unique keys; some do not accept duplicate records.)

- Each kind of fact about an entity always involves entities (or attribute values) of a single type. We don't expect two different kinds of entities to occur in the "employer" fields of two people's records; the record system doesn't have any way of telling us which type is occurring in that field for a particular record occurrence.
- And the entities or attribute values involved in a given kind of fact all have the same form of name (representation). We don't have self-describing records which tell us which data type or format is being used in this particular record occurrence.
- A given entity should be referenced by the same name (representation) everywhere it occurs. The only way we know if two references are to the same thing is by a match on the fields containing those references.
- There is an essential difference between entities and attribute values, and between relationships and attributes. The difference seems to correlate with the things that are or aren't represented by records. If there's a record, then the thing it represents is an entity, and a reference to it in a field comprises a relationship (as in the department field of an employee record). But if there is no separate record for the thing, then a reference to it involves neither an entity nor a relationship; it's simply an attribute value (as in the salary or spouse fields of an employee record).
- Relationships are not distinct constructs to be represented in a uniform way. Obviously; otherwise we wouldn't be provided with so confusingly many ways to represent them.
- Many-to-many relationships are (usually) entities in their own right. And the associations implied by multi-valued attributes are also entities, even though they aren't relationships. (This all follows from their being represented by distinct records.) But one-to-many relationships are (usually) not entities.

- Relationships and compound identifiers are the same phenomenon, since they can have the same representation.

8.2 The Type/Instance Dichotomy

The dichotomy between types (descriptions occurring in catalogs or dictionaries) and instances (occurring in files or databases) itself makes some limiting presumptions about information.

8.2.1 An Instance of Exactly One Type

If we intend to use a record to represent a real world entity, there is some difficulty in equating record types with entity types. It seems reasonable to view a certain person as a single entity (for whom we might wish to have a single record in an integrated database). But such an entity might be an instance of several entity types, such as employee, dependent, customer, stockholder, etc. It is difficult, within the current record processing technologies, to define a record type corresponding to each of these, and then permit a single record to simultaneously be an occurrence of several of the record types.

Note that we are not dealing with a simple nesting of types and sub-types: all employees are people, but some customers and stockholders are not.

To fit comfortably into a record-based discipline, we are forced to model our entity types as though they did not overlap. We are required to do such things as thinking of customers and employees as always distinct entities, sometimes related by an "is the same person" relationship. At most, it might be possible to model a simple type and sub-type structure, where records of the sub-type can be obtained by simply eliminating irrelevant fields from the containing type.

8.2.2 Descriptions Are Not Information

The information in a file consists mainly of field values occurring in records. Thus there is likely to be a data item answering the question "Who manages the Accounting department?" The manager's name can be found in a field somewhere. But it is not likely that the file can provide an answer to "How is Henry Jones related to the Accounting department?" There are no fields in the file containing such entries as "is assigned to", "was assigned to", "on loan to", "manages", "audits", "handles personnel matters for", etc. Depending on how the records are organized, the answer generally consists of a field name or a record type name, which are not contained in the database. To a naive seeker of information from the database (e.g., via a high-level query interface), it is not at all obvious why one question may be asked and the other may not.

It's not just that he can't get an answer; the interfaces don't provide any way to frame the question. The data management systems do not provide a way to ask such questions whose answers are field names or record type names.

Then consider the following questions:

1. How many employees are there in the Accounting department?
2. What is the average number of employees per department?
3. What is the maximum number of employees currently in any department?
4. What is the maximum number of employees permitted in any department?
5. How many more employees can be hired into the Accounting department?

If the maximum number of employees permitted is fixed by corporate policy, then a system offering advanced validation capabilities is likely to place that number into a constraint in a database description, outside the database itself. Our naive

seeker of facts will then again find himself unable to ask the last two questions. He might well observe that other things having the effect of rules or constraints are accessible from the database, such as sales quotas, departmental budgets, head counts, safety standards, etc. The only difference, which doesn't matter much to him, is that some such limits are intended to be enforced by the system, while others are not.

This suggests that we might want to seek a way to represent such constraints in the same format — and in the same database — as "ordinary" information, but with the added characteristic that they are intended to be executed and enforced by the data processing system.

It is true that descriptions and constraints are inherently different from other data with respect to their update characteristics. Changes to these imply differences in the system's behavior, ranging from changes in validation procedures to physical file reorganizations implied by format changes. But such descriptions and constraints need not be inherently different for retrieval purposes. And even with respect to update, the method need not be inherently different as perceived by users. It is only necessary that the authorization to do so be carefully controlled, and that the consequences be propagated into the system.

Some information is in the catalog rather than in the record occurrences because it is the same for all occurrences, hence factorable. Field names, types, and lengths are typically treated that way. But there have been proposals for, and probably implementations of, systems of self-describing data. A record might consist of a chain of pairs: a field name and a field value. Then fields irrelevant to a given record occurrence just don't occur. Or each field might be accompanied by its own descriptor of length, data type, units, etc., so these can vary from record to record. In such cases the names, lengths, and types of fields would be in the file and not in the catalog. Record lengths, as another example, occur in the catalog for fixed length records, but in the records for variable length records. If an application using a file of mixed record types needed to know the length of

each record occurrence, it might be able to find that information in some records and not in others.

There seem to be two real motivations for putting information into the catalog:

- It is used by the system.
- It can be factored, i.e., it applies to all occurrences of a given type.

These may be good reasons for maintaining this information in special system-usable formats, and for being especially concerned about controlling updates to it. But it is a mistake to presume this to be an inherently different kind of information, which does not need to be made available to users in the same way as file data.

8.2.3 Regularity (Homogeneity)

Record structures work best when there is a uniformity of characteristics over the population of an entity type. It is usually necessary for the entire population to be subject to the same naming conventions (e.g., there has to be something that can serve as a key field over the entire population). It is usually assumed that all instances of the entity type are eligible to participate in the same relationships.

Most fundamentally, it is presumed that the entire population has the same kinds of attributes. While exceptions are tolerated, the essential configuration is that of a homogeneous population of records, all having the same fields. The underlying assumption is that field names can be factored out of the data

The more we deviate from this norm of homogeneity, the less appropriate is the record configuration. There are certain techniques for accommodating variability among instances in a record structure, but these need to be used sparingly. If there can be considerable variation among entity instances, then the solutions become cumbersome and inefficient. Such solutions include:

- Define the record format to include the union of all relevant fields, where not all the fields are expected to have values in every record. Thus many records might have null values in many fields.
- Define the *same* field to have different meanings in different records. Unfortunately, such a practice is never defined to the system. With respect to any processing done by the system, that field appears to have the same significance in every record occurrence. It certainly has only one field name, which in these cases usually turns out to be something totally innocuous and uninformative, like CODE or FIELD1. It is only the buried logic in application programs which knows the significance of these fields, and the different meanings they have in different records.

Many entity types come to mind for which considerable variability of attributes is likely to occur, such as people, tools, clothing, furniture, vehicles, etc. For example, in a file of clothing records, consider which of the following field names are relevant (and what they might mean) in each record: size, waist size, neck size, sleeve length, long or short sleeves, cup size, inseam length, button or zipper, sex, fabric type, heel size, width, color, pattern, pieces, season, number, collar style, cuffs, neckline, sleeve style, weight, flared, belt, waterproof, formal or casual, age, pockets, sport, washable....

You can play the same game with the other entity types, or even try to extend this list.

8.2.4 Pre-definition (Stability)

Another implication of record formats, and of the file plus catalog configuration, is that the attributes applicable to an entity are pre-defined and are expected to remain quite stable. It generally takes a major effort to add fields to records.

While this may be acceptable and desirable in many cases, there are situations where all sorts of unanticipated information

needs to be recorded, and a more flexible data structure is needed.

The need to record information of unanticipated meaning or format is crudely reflected in provisions for "comments" fields or records. These consist of unformatted text, in which system facilities can do little more than search for occurrences of words. There is no way for, say, a query processor to know which words in the text name specific things (analogous to field values), which words specify their relationship to the thing being described (analogous to field names), etc. Thus, ironically, we have the two extremes of rigidly structured and totally unstructured information — but very little in between.

8.3 Too Many Ways To Represent Relationships

One way to represent relationships is to have two fields in a record containing data items which represent the two things being related, e.g., an employee number and a department number. Unfortunately, this constitutes three ways, not one: we generally may have a choice of three different records into which we specify these fields. They might be incorporated into a record representing either one of the related entities (e.g., a department number in the employee record, or an employee number in the department record), or they might be isolated into a new record defined just for the purpose of representing the relationship (so-called *intersection records*).
In the employee records:

EMP-REC:

EMP	AGE	SALARY	DEPT	…
Jones	25	20,000	Acctg	
Smith	27	22,000	Sales	

In the department records:

DEPT-REC:

DEPT	MGR	EMPLOYEES	…
Acctg	Zim	Shaw, Cap, Jones, Park	
Sales	Dun	Smith, Ho, Asp, Cole	

In separate intersection records:

EMP-DEP:

EMP	DEPT
Asp	Sales
Cap	Acctg
Cole	Sales

The employee and department fields together constitute the key (identifier) of such an intersection record. Intersection records may include additional fields bearing information about the relationship — so-called *intersection data*. A third field, containing the date of assignment, might occur as such intersection data in the intersection records shown above.

The actual set of choices available depends on whether the relationship is binary, whether the relationship is many-to-many, and whether the system is normalized.

Non-binary relationships (having degree greater than two) always seem to get handled separately in intersection records.

For binary relations in a non-normalized system (where repeating fields or groups are permitted), all three options are available.

In normalized systems, only two options are available for one-to-many relationships. Since we can't have a list of employee names in a department record, that option is ruled out. And, in actual practice, intersection records are virtually never used for this case either. The overwhelmingly predominant practice for this case is to embed the relationship into the records on the "many" side (i.e., a department name in the employee record).

Finally, for many-to-many relationships in normalized systems, there is no choice: separate intersection records must be used. For example, the employment history relationship (giving all the departments in which an employee has worked), must be kept in the intersection records shown above.

These practices lead to different representations for very similar relationships. Although current assignments and employment histories both relate employees to departments, the

current assignments are invariably embedded into the employee records, while the history gets a separate record type of its own.

8.4 But Some Relationships Can't Be Described

8.4.1 Relationships Within a Record

Ironically, while faced with such a plethora of techniques for representing relationships, we are sometimes unable to specify everything we want to about the relationships. The syntax of record descriptions often does not provide any way to express the structures that may exist within or among relationships.

To understand what we mean by formally modeling the structure of information, consider a very generalized query processor trying to deal with records like the following:

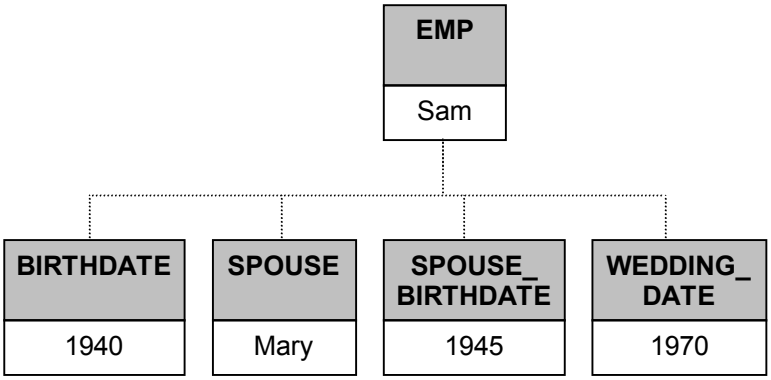
EMP	BIRTHDATE	SPOUSE	SPOUSE_ BIRTHDATE	WEDDING_ DATE
Sam	1940	Mary	1945	1970

Presumably the query processor could discover from the catalog or dictionary that EMP is a key field, and the only key field, for this record. But it is not capable of understanding the meaning of the other field names. Thus, if asked what information was available about employees, it would respond with a list of four field names, BIRTHDATE through WEDDING_DATE. If asked what information was available about a SPOUSE, the processor might be clever enough to invert the structure and let you know that it could find the corresponding employee — but that's all. It does not know that there is other information available about a spouse, such as birth date.

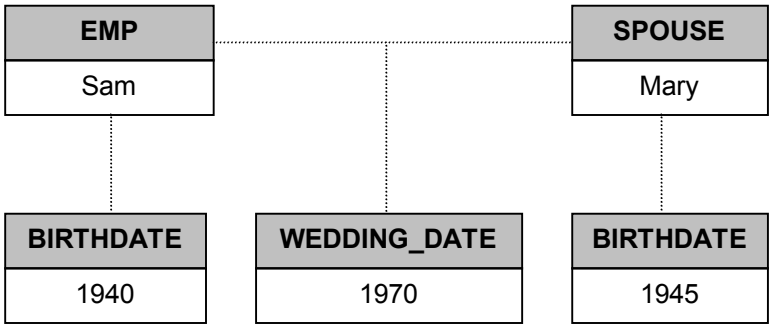
If asked about Sam's spouse, it could reply "Sam has spouse Mary." If asked about Mary's birthdate, it would reply "no data available." But if asked the right question, it might reply "Sam

has spouse_birthdate 1945", totally unaware that this was information about Mary.

In much the same way, the system would report the wedding date as a fact about the employee. There is no way to inform the system that this should not be regarded as a fact about the employee, nor even about the spouse for that matter, but as a fact about a relationship between the two of them. As far as the system is concerned, the fundamental semantic of a record format implies that the information has the following structure:



even though we might believe the information has this structure:



There are two relationships here whose existence cannot be inferred from the record structure description:

- The one between the spouse and his/her birthdate.

- The one between the wedding date and the relationship which represents their marriage.

But even when the relationship can be defined, we are limited in the kinds of information we can provide about it. In general, we might want to name three constructs:

- The relationship itself, e.g., MARRIED.
- The role of each element participating in a relationship, e.g., HUSBAND and WIFE.
- The kinds of elements permitted to play each role (the domains of the relationship), e.g., MEN and WOMEN.

For human purposes, because we understand a lot of the semantics behind such words, it isn't always necessary to be explicit about all three. If WIFE occurs as a field name, we know that the relationship is marriage and the domain is limited to women. The field name ADDRESS explicitly refers to the domain of things which may occur there; we may infer that the role is RESIDENCE, and the relationship is RESIDES. Sometimes we use compounded field names like DATE-OF-ASSIGNMENT to combine several of these names into one.

Unfortunately, however, record descriptions don't give us any regular way to name all three of these constructs, nor to indicate which ones we are naming. The only tools we can try to adapt are the record type name and the field name, but this only gives us two names for three constructs. We tend to use these two names in all sorts of combinations for relationships, roles, and domains, making it difficult to design any system facility using field names to deal with relationships.

Actually, the two names are only available in intersection records, which are exclusively used to model the relationship. In most records, only field names are available for these purposes. Because a record is often a conglomerate of many relationships, the record type name doesn't refer to any of them. Instead, the record type name often just refers to the focal object of these relationships, e.g., an EMPLOYEE record.

Consider two relationships involving the indicated roles, with the corresponding domains shown in parentheses:

- RESIDES: RESIDENT (PERSON), RESIDENCE (ADDRESS).
- EMPLOYED: EMPLOYER (COMPANY), EMPLOYEE (PERSON).

In a real file, say of a bank's clients, these might be buried within a single CLIENT record, with just three field names:

CLIENT	ADDRESS	EMPLOYER

The names of the relationships are lost altogether. One of the field names is a role name, the other is a domain name, and the third is neither. Both of the role names associated with the client are also lost.

8.4.2 Relationships That Span Records

Many relationships are represented by matching field values in two records, i.e., a symbolic linkage (a form of computed relationship — section 4.6). Sometimes this linkage must be traversed just to discover which thing is related, but sometimes it is only traversed if we need more information about the related thing.

An example of the first kind would be the case where employee records and project records both contained department numbers. It might be the policy that all employees in a department work on all projects assigned to the department. Then the way to determine which employees work on which projects is to match employee and project records on common department numbers (the relational "join" operation). I.e., you can't discover from the employee record which projects he works on. You have to go over to the project records and find the ones that have a department number matching his.

An example of the second kind is again provided by the department number in an employee record. One needn't go any further to identify the department to which the employee is assigned, but in order to find out anything else about the

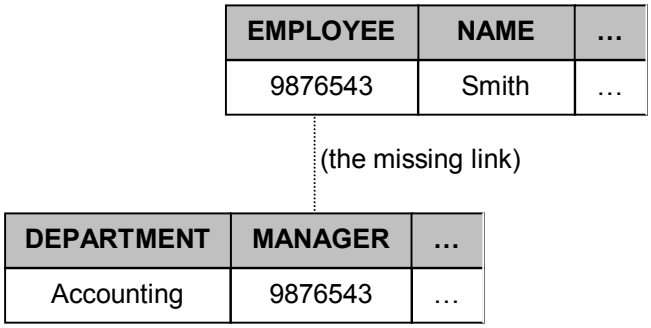
department (such as its name, or the manager) one has to get to the corresponding department record.

In a conventional record-processing environment, three things are rarely described about such linkages:

- The existence of the linkage.
- Which record types are involved.
- What the linkages signify, e.g., the name of the relationship. (This is often implicit in the field names for the second kind of linkage, but not indicated at all for the first kind of linkage.)

Thus these relationships are not really modeled, in the sense that our generalized query processor doesn't really know how to traverse them. How can this processor reply to "Which employees work on which projects?" There's nothing in any directory to identify what "work on" signifies. There's probably nothing to indicate that there's any connection at all between employees and projects.

Or consider the query "What is the name of the manager of the Accounting department?" A department record is likely to have a field named MANAGER, probably containing employee numbers. Who is going to tell the query processor that it ought to look in EMPLOYEE records to find the name of a MANAGER?



Domains

One construct that helps with some of these problems is the "domain" concept, which is used with some success in the relational model. It is used in the sense that if fields take their values from the same domain, then the fields are representing the "same kind of thing", and hence imply the basis for some kind of linkage. This requires that the record descriptions include an identifiable domain specification in each field description.

Unfortunately, the construct is not used in a very consistent way, and in any case still has its limitations.

The occurrence of common domains does not identify the nature of the relationship. Employee records, project records, and equipment records may all contain fields taken from the domain of departments; that doesn't tell us anything about the nature of the relationships among any of these entities.

A domain name may or may not provide a clue as to what record type to seek for the related item. There is generally no discipline that requires domain names to coincide with record type names for a given entity type. Thus these various fields we've mentioned might all come from a domain named "DEPARTMENTS"; the record type for these entities might well be named "DEPTS", or "DEPT-RECS". There's no help here for our query processor.

Sometimes the domain concept is shifted to refer to representations, or data types, rather than entities. Thus such things as employee numbers, or characters, or integers, might get specified as domains. If "employee numbers" is a domain for some field, no linkage will be recognized with a field whose domain is "social security numbers", even though they might refer to the same person. If "characters" is a domain, then spurious linkages will be implied between, e.g., fields containing names and fields containing addresses.

And, finally, the domain construct is always implemented, if at all, in terms of simply matching domain names. This totally fails to allow for overlapping entity types, e.g., domains and sub-domains. There is no system I know of which will recognize a

linkage between one field whose domain is “employees” and another whose domain is “people”.

Non-symbolic Linkages

Non-symbolic linkages (i.e., those implemented by some kind of file structure rather than by matching field values) offer certain advantages.

Their existence is always known (described) to the system. The description often names the relationship (but not always, as in hierarchies).

There is always a known path to the related record type.

And a certain kind of validity checking, induced by symbolic linkages, can naturally be avoided. Namely, a non-symbolic linkage simply cannot be established to a non-existent entity. With symbolic linkages, this must typically be expressed as an explicit constraint, such as “if an employee number occurs in the manager field of a department record, then there must also exist an employee record containing that employee number”.

8.4.3 When is it an Intersection Record?

Any two fields in any record represent some kind of relationship. In the records illustrated in section 8.3, we can detect such relationships as the following:

- A correlation between ages and salaries.
- A correlation between managers and employees.
- A correlation between employees and departments.

In effect, every record is an intersection record. If it has more than two fields, it is representing a multitude of relationships simultaneously. How does the system really know which of these records is “intended” to represent a relationship? How does it know that EMP-DEP is the name of a relationship between employees and departments, but EMP-REC is *not* the name of a relationship between ages and salaries?

Does the system really know anything about intersection records, or is it all in the minds of the users?

8.5 And Some Relationships Can't Even Be Represented

It's a little discomforting to find, as in the preceding section, that we have some relationships in our database that we can't adequately describe in the catalog or dictionary.

It's far worse to discover relationships that can't even be represented in record structures in the database. That is, we can't even record the data, let alone describe it.

A record type description is based on the fundamental premise that each occurrence of a given field (i.e., in each record occurrence) contains the same type of data item, and hence the field can represent exactly one entity type. It follows that binary relations can only occur between two entity types (or within one entity type, as in people to people relationships). There is no provision for (no way to represent) relationships permitting multiple entity types in one domain, especially when those entity types have very different naming conventions.

Such relationships certainly do exist. Companies, government agencies, schools, and people will usually be treated as distinct entity types — but any of these might be a person's employer. We may treat furniture and vehicles as distinct entity types, but they share a common relationship to their manufacturers. As a general example, consider an "owns" relationship: various kinds of things (employees, departments, divisions, locations) can own various kinds of things (furniture, vehicles, supplies, machines, buildings). Potentially each kind of thing might have a different identifier syntax, in terms of length, character set, variability, etc. Even worse, their names might have different qualification structure, e.g., department names are only unique within divisions, and hence a department name must always be qualified by a division name. If we start with a simple form of this owning relationship, where employees or

departments own furniture or vehicles, then we have a configuration like this:

PROPERTY	OWNER	
Furniture ID	Emp Number	
Vehicle Num	Div	Dept

Such a relationship has several interesting characteristics. For one thing, it has a variable number of fields, depending on whether the owner is an employee or a department (in relational terms, this is a relation of degree two and a half, on the average). Secondly, the terms "furniture id" and "vehicle num" would typically occur as field names; in this file, different occurrences of these "records" may have different field names associated with them. Furthermore, one does not know how to interpret a field (or even how long it is, or how many there are) without knowing the type of entity represented there. Here is a case where the *type* of an entity is itself useful information to be obtained from the database; one should be able to ask "what is the type of the owner of vehicle ABC123?" And, if you just inquire about the owner of that vehicle, you should be provided with a two-part answer: the type and the name.

In general, record formats can't accommodate the case where:

- A given kind of fact (e.g., who owns that?) might involve several types of entities.
- Each of these entity types has a different representation (length, data type, etc.) for its names. (Even if they had the same representation, the format only works if the names are unique across all the entity types. It's no good if a furniture identifier might turn out to be the same as some vehicle number.)
- Or, even worse, the different entity types need different numbers of fields to represent them.

One “solution” that record processing might force on us is to create artificial super-domains or super-types, one containing all owners and one containing all owned things. (And this presumes the system permits us to deal with types and sub-types in the first place, e.g., owners and employees.) A new and artificial identifier would have to be created applying to everything in the domain; employees and departments would have to acquire new “owner numbers”. The “owns” relationship would have to be recorded using these new identifiers, rather than more familiar employee numbers or department names. The same would apply to all owned things; they would acquire an arbitrary “property number” as another synonym. Furthermore, whenever these domains got extended (e.g., to include locations as owners), then a whole new set of owner numbers must be assigned to these other entities.

Another “solution” that fits the record oriented base is to partition this example into four relations:

- employees own-1 vehicles,
- employees own-2 furniture,
- departments own-3 vehicles,
- departments own-4 furniture.

This approach also has interesting consequences. A single relationship name (“owns”) has been replaced by four, which users have to learn to discriminate between. What used to be a simple inquiry (“who owns vehicle ABC123?”) now requires an interactive dialog, or some conditional programming statements: “which *employee* owns-1 vehicle ABC123?” “if nobody, then which *department* owns-3 vehicle ABC123?” To ask that, you now have to know in advance all the kinds of things which might be owners, and which is the appropriate form of the “owns” verb for each.

And integrity constraints get much more complex. If a thing can have at most one owner, then in the original example it was sufficient to specify that “owns” is a one-to-many relationship. Now we have to specify that for each of the four new relationships, *plus* the constraints:

- a vehicle may be owned via own-1 or own-3, but not both;
- a piece of furniture may be owned via own-2 or own-4, but not both.

Finally, if these domains are extended to include more entity types, then all these problems explode quite rapidly.

[Chen] avoids dealing with this problem by using the term "entity set" loosely, sometimes referring to an entity type and sometimes to a domain of a relation. There is always the unstated assumption of a homogeneous naming convention over the whole set.

8.6 Do Records Represent Entities? Or Relationships?

If we use records to model reality, it is fairly natural to assume that we intend a record to represent an entity. By this I mean we might expect to find a one to one correspondence between instances of records and entities (within the sphere of interest; the database does not model the whole world). I will refer to this as the *modeling assumption*.

Several questions arise. The difficulty of modeling entities that belong to several entity types has already been mentioned. Other questions follow.

8.6.1 No Record, No Entity?

A corollary of the modeling assumption is that if something is not represented by exactly one distinct record, then it is not considered an entity. How well does that jibe with our intuitions about entities?

We might have too many records. There is no discipline preventing the definition of several record types (or relations, in the relational model) corresponding to one entity type. That is, we could have several record types defined over the same key, with each record type containing different attributes of the subject entity. One might be tempted to do this for economic

reasons, e.g., to group together attributes that tend to be accessed together, or to physically segregate rarely used data. Regardless of the motivation, such a configuration is permitted in all record based systems I know of. Thus none of these systems really has a well-defined semantic establishing a one to one correspondence between entities and records.

Conversely (and ironically), there is actually no discipline which requires any record at all for an entity. This can occur (in a normalized system) if there didn't happen to be any single-valued information about the entity. Suppose one had in mind to treat projects as entities, but all the information to be maintained about them turned out to be multi-valued (in relational terms, we find no functional dependences on projects). That is, our projects can have multiple managers, multiple objectives, multiple start and stop dates, multiple budgets, and so on. Each such fact needs to be maintained in a distinct intersection record, and there might be no motivation to define a single record type or relation to represent the projects themselves. One would have record types (relations) called "project-manager", "project-objective", "project-dates", and so on, but none called simply "project".

In general, if "being the subject of information" is the criterion for thinking of something as an entity, then there are often many entities which are not represented by their own record types.

There are other, more common situations where we don't have any distinct records representing certain things. These are the things we intuitively think of as attributes of other things, such as salaries, colors, birthdates, etc. Unfortunately, apart from the listing of examples, I find it difficult to identify precise criteria for deciding whether something is an entity, and whether it is to be represented by a record (obviously, I'm still not sure if those are the same question).

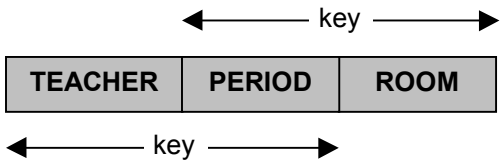
8.6.2 If It Has A Record, It's An Entity(?)

Another corollary to the modeling assumption is this: if we find some things which are in one to one correspondence with a

set of records, then the records are representing those things, and those things are entities.

By this reasoning, anything corresponding to a unique key in a record is an entity. (Those of you familiar with the relational model may recognize that these are the things that become the "subjects" of relations via an analysis of functional dependences.)

Such a rule may not always be intuitively satisfying. Consider a ternary relationship expressing an examination schedule, in the form



The combination of a given teacher and a given period can occur at most once. Hence this pair qualifies as a key, and it uniquely determines the corresponding room (i.e., the pair is the subject of a functional dependence).

Similarly, the combination of a given room and a given period can occur at most once. Hence this pair also qualifies as a key, and it uniquely determines the corresponding teacher (this pair is also the subject of a functional dependence).

In contrast, the combination of a given teacher and a given room may occur many times (in many periods). Hence this pair is neither a key nor the subject of a functional dependence.

The notions of key, and of subjects of functional dependences, are unreliable as determinants of entities. Sometimes the uniqueness of fields arises from auxiliary semantics having nothing to do with an intent to identify entities. In this case the uniqueness of teacher-period pairs and room-period pairs happens to arise from these constraints:

- A teacher can only occupy one room per period.

- Only one teacher is assigned to a room for a given period.

These constraints have nothing to do with "entity-ness". There is no reason to consider the teacher-room pair as intrinsically different (e.g., not an entity) from the other two. Either all of these pairs are entities, or none of them is.

8.6.3 Are Relationships Entities? Are Attributes?

If all records represent entities, then what entity does an intersection record represent? It must be a relationship.

Are all relationships entities? Not the ones that are embedded in the records of other entities, if by representing we really do mean a one to one correspondence. (If an employee and his relationship to a department are two distinct entities, then they shouldn't be "represented" by the same record.)

Certain arguments are sometimes advanced as to why intersection records represent distinct entities, such as:

- They represent information which is symmetrically about both related items, hence shouldn't be exclusively in one record or the other.
- There is data maintained about the relationship that is not directly data about either related item (e.g., quantity on hand for parts in warehouses, assignment dates for employment histories). That is, anything about which data is maintained must be an entity; ergo, these relationships are entities.

Unfortunately, these generally apply equally well to the one-to-many relationships which are typically embedded in other records (an employee record is likely to contain a date of assignment field).

Record-based modeling in general tends to exaggerate out of all proportion the difference between one-to-many and many-to-many relationships, especially in normalized systems. That this is a relatively minor semantic distinction is especially noticeable

when comparing such similar relationships as "current employment" and "employment history".

It's difficult to get the three concepts of "record", "relationship", and "entity" consistent with each other. We could stop claiming that all records represent entities (in a one to one fashion). Or we could arbitrarily consider some relationships to be entities and others not, based on no real criterion except how we arbitrarily chose to model them. (Remember, there is nothing to prevent a one-to-many relationship from being split out into intersection records.) Or we might consider all relationships to be entities, and split them out into distinct intersection records. This leads to very small records, and begins to approach a "binary relation" model (section 10.2), or an "irreducible" model (section 10.3).

If you want to distinguish between relationships and attributes (I don't, but many people do), there are other problems. Using more or less traditional meanings of certain terms, it turns out that some intersection records don't even represent relationships. They also have to be used for multi-valued attributes (at least in a normalized system). That is, if cars can be multi-colored, then this has to be represented in an intersection record type, with each record recording one of the colors of one of the cars. This kind of fact is traditionally called an attribute, rather than a relationship. (Because the colors are not entities, because they are not represented by records.)

Let me cope with another ambiguity at this point: I am using the term "attribute" to refer to the association between an entity and some value, and not to that value itself. (Blue is not an attribute; but the blue-ness of my car is.) Conventionally, that association is considered to be neither a relationship nor an entity. Hence those intersection records represent neither relationships nor entities.

In addition to being represented by their own records, such so-called "attributes" can exhibit another characteristic property of entities (or relationships): they themselves might be the subject of some information (like intersection data). Thus, the intersection records required to list the children of employees might include their birthdays; the intersection records listing the

colors of multi-colored cars might include the percentage of a car's surface covered by a given color.

To summarize, consider the contradictions in the following assertions:

- Every record represents an entity.
- Every entity is represented by a distinct record.
- All relationships are entities.
- Some relationships are not represented by distinct records.
- The "subject" of an attribute is an entity.
- Some attributes are the subjects of other attributes.
- Attributes are not entities.
- Some records represent attributes (and nothing else).

You play the game. See how many contradictory combinations you can find.

Then decide which of those assertions you're willing to give up in order to achieve consistency.

8.6.4 The Create/Destroy Semantic

Does insertion and deletion of records model the creation and destruction of entities?

The one characteristic of records that might be informationally meaningful for an entity concept is the create/destroy semantic. If a record represents an entity, we have an implication that the lifetime of that entity is explicitly signaled to the system by the creation and destruction of that record. If the thing is not represented by a record, then it could be referred to at any time in other records without any prior announcement of its existence (the traditional situation for such things as colors, dates, and most numeric quantities).

This semantic is not always enforced. In some systems it might be possible to mention a department name in an employee record without any verification that the corresponding department record exists. Or it might be possible to delete a

department record without regard for existing references to that department in other records. And, in fact, such "enforcement" might not always be desirable. It is plausible that an installation might wish to purge records of employees terminated more than 25 years ago, but still retain other records that happen to mention such employees.

The creation and destruction of records might have various semantic interpretations in the real world. Occasionally it might really signify the beginning and end of an entity (e.g., the birth and death of a person). More often, however, "create" or "destroy" are really instructions to the system to "notice" or "forget" an entity, quite unrelated to the beginning and end of the entity. A personnel record is created when a person is hired (which could be interpreted as "create this employee", but also as "notice this person, who was born a long time ago") — perhaps except when a former employee is re-hired (perhaps no new record is created at all). And historical records are likely to be kept long after an employee terminates, or a person dies. Thus one still has to explain somewhere what semantic is implied by the creation or destruction of a record.

In structured files, record deletion carries with it problems of cascading delete (which related records must also be deleted?). The rules dictated by the file structure have to be very carefully correlated with the semantics of "existence dependence" among the real entities.

If we want to think of relationships as entities, then the create/destroy semantic is inconsistently applied. Some relationships — namely those embedded in the records of other entities — can be modified by simple update of those records. For other relationships, however — namely those maintained in separate intersection records — the same "update" might have to be done by destroying and creating records (at least in those systems which don't permit update of key fields).

Thus, in the long run, it's probably better to specify explicitly what we intend for the create/destroy semantics of an entity, rather than relying on the behavior of the corresponding records in the system.

8.7 Distinguishability

Two questions arise here. Do records have to be distinguishable by their contents? Do they have to be distinguishable at all?

If some kind of file structure is available, such as ordering or a hierarchical structure, then that structure can be used to distinguish records which are identical in content. One can refer to the first record, or the next after X, or the parent of X, and so on, without ever mentioning anything about the content of the desired record.

Even with such capability, some systems do not permit duplicate records.

In systems without any such file structure, such as the relational model, records can only be distinguished by their content. The relational model does in fact require this distinguishability; duplicate records are not permitted.

Rather than comment on such constraints directly, let me just illustrate some behavior of real entities which doesn't conform to such constraints.

We don't always need all entities to be distinguishable, even though we want them to be modeled as distinct. That is, we may want to know there are several of them, and want to be able to say distinct things about them in the future, but at the moment we don't care which is which. There doesn't have to be any difference in the facts we know about them. One example is the table of organization of a military unit, or a duty roster. The permanent entities are the jobs, with such attributes as the job title, salary, experience requirements, etc. One of the transient facts about a job is the person currently holding it; when several identical jobs are vacant, the records may perfectly well be identical. When we hire a typist, we will simply ask for one of the unoccupied typist records; we don't care which.

To manage the circulation and inventory of a library we might have one record for each physical book, but only keyed on title and author (allowing duplicates). The library might not need to keep track of each copy individually, but still have a separate record to indicate who has currently borrowed it. (Many libraries

do give each physical book its own distinct “accession number”, but the point is that this needn’t always be the case.)

In other cases it may be inconvenient to maintain distinguishability on the basis of field contents. For entities which are primarily distinguished by order (e.g., lines of text in a text file), it can be very cumbersome to maintain a sequence field — especially if two independent processes can be doing insertions and deletions in different parts of the file. (However, there is an alternative: instead of modeling order with a sequence field, one could simulate chaining by including in each record the keys of its predecessor and successor. This does require that both adjacent records be locked and updated in order to insert or delete a record in the middle, and also that some form of null value is available for use in the first and last records.)

Sometimes identical records can be distinguished in more complex ways via structure, but still have multiple paths, or unpredictable path lengths, making it difficult to capture the distinction in a field format. (E.g., different children may have duplicate names, birthdates, etc., so long as they are unique within parent. Thus a child might be uniquely reached on a path through either parent. If the child’s record has to be distinct on the basis of its content, then an arbitrary convention has to be established to select either the mother’s or father’s identifier to be included in the child’s record.)

Thus it seems at least debatable whether records always need to be distinguishable, or distinguishable by content.

8.8 Naming Practices

8.8.1 Things and Their Names

Record structures work best when there is an exact one to one correspondence between entities and their names (representations), i.e., no synonyms or ambiguities. And they work best when all entities of a given type have the same name formats (representation). Under these conditions, it is feasible to have a single format specified for a field in which these entities

might occur. And it is easy to detect references to the same entity: just match the contents of the fields.

Real entities don't always behave so simply. The employees of a multi-national corporation might not all have social security numbers, or employee numbers (or they might be in different formats in different countries). But many employees have both, and some may have several social security numbers. Some books don't have "International Standard Book Numbers" (ISBN), others don't have Library of Congress numbers, and some have neither. But many books have both — and some have several ISBN's. And Library of Congress numbers apply to a larger class of entities than do ISBN's; they are also assigned to films, recordings, and other forms of publication, in addition to books. Oil companies have their own conventions for naming their own oil wells, and the American Petroleum Institute has also assigned "standard" names to some wells — but not all.

For all practical purposes, record systems can't cope with partially applicable names. In order to use records for an application, it is necessary that some naming convention be adopted which applies to all occurrences of the entity type.

Synonyms are not really managed at all, as far as the structure and description of data are concerned. If fields in two different record types contain employee numbers, then the system can perceive that some of these records might refer to the same person. (This is, in fact, the fundamental mechanism for expressing relationships in the relational model — matching field values imply that two records are related, and can be "joined".) But if one record type contains social security numbers instead, then this knowledge is lost. As far as the system is concerned, there are no potential relationships here. It is only in the minds of users, and in procedural logic buried in programs, that any suspicion lurks that these might in fact refer to the same people.

And in all of this, we haven't bothered to mention simple synonyms. Many skills, jobs, companies, people, colors, etc., etc., have more than one name. We might have to deal with them in multiple languages, as well. We have many ways to represent the same date. Quantifiable things are written in different ways

depending on the unit of measure, data type, number base, and so on. Our systems are usually inconsistent in handling these: they will help with such things as conversion algorithms in some cases, but not in others.

It can be very difficult to model, in a record based system, the knowledge that different representations in different records might refer to a single underlying entity (cf. [Stamper 77], [Hall 76], [Falkenberg 76b], [Kent 77a]).

Perhaps the most blatant illustration of this is our inability to manage mailing lists. I don't know how to explain to my non-technical friends why sophisticated modern computers can't eliminate the duplications in a mailing list. The most trivial variation in the way a person writes, abbreviates, or punctuates his name or address is enough to confuse the system, and prevent it from recognizing references to the same person.

8.8.2 Structured Names

Additional confusion arises when the synonyms of an entity exhibit different kinds of structure. A person's name might be structured into three fields for first, middle, and last names; his other synonyms are single fields: employee number, social security number. A date (if you will accept that as an entity) has three fields in the traditional representation, but only one in Julian notation. (A Julian date is a single integer combining year and day of year: the last day of 1977 is 77365.) Now, every relationship involving a person or a date will have an uncertainty, not only with respect to the data items the fields might contain, but also with respect to the number of fields occurring in the record. Thus a binary relationship between people and dates (e.g., birthdates) could be represented in two, four, or six fields, depending on the representations chosen. But it is still fundamentally a binary relationship. Thus there is potentially a poor (and unstable) correspondence between the degree of a relationship and the number of fields used to represent it.

Note that this differs from an earlier situation (section 8.5) where we had different kinds of entities. Here we have the same entities, but different names.

8.8.3 Composite Names and the Semantics of Relationships

Composite (e.g., qualified) names occurring in records tend to confuse the purpose and semantics (and degree) of the relationships being represented. This is especially noticeable when the composite names are themselves based on relationships. Consider, for example, the naming of employee's dependents by the two fields consisting of the employee identification plus the dependent's first name (as in section 3.3.2).

Redundancy

The dependents in this illustration might occur in any number of relationships, being related, e.g., to benefits programs for which they are eligible, histories of claims and payments, employees responsible for them as counselors, other employee records because the dependents are themselves employees, etc. From an informational point of view, the employee on whom the person is dependent comprises a distinct, independent relationship. Yet, due to the naming convention, this information is gratuitously carried around in all the other relationships. For all of the other information, there is a single well-defined relationship that must be accessed to get the facts; but for this particular information, any relationship will do. (Of course, that gratuitous information would suddenly disappear if the naming convention for dependents was switched from qualified naming to social security numbers.)

A basic information model should be able to represent dependents as individual entities in these relationships, without dragging their related employees into every such context. If it is useful for applications to see dependents so identified in various relationships, then it is appropriate to define such derived "views" for the benefit of these applications. But the underlying information model need not confuse relationships with identification. A given relationship (e.g., between a dependent and a benefit program) exists independent of the means of

identifying the dependent. That relationship should not be perturbed by problems or changes which might arise in the identification scheme.

Degree

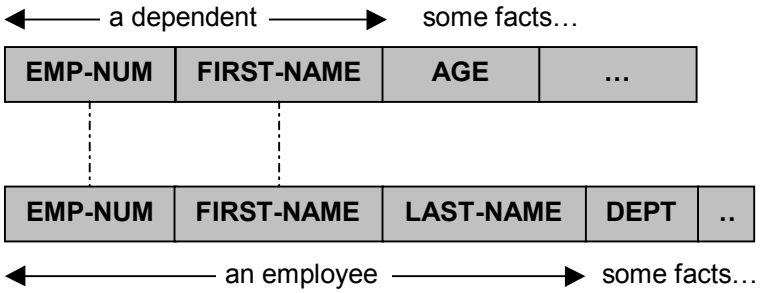
And the degree gets confused. The relationship between a dependent and a counselor is nominally a binary relationship, but it has three fields, two of which identify employees. In a certain sense, one of them is "really" there as part of the intended information, while the other is a "phantom" introduced by virtue of the naming convention. This phantom employee would disappear if dependents were to be identified by social security number instead of a qualified name. (Does the degree of the relation depend on the naming conventions used?)

Domains, Implied Relationships

In the relational model, a potential linkage (relationship) is implied when two fields take their values from the same domain. A join operation is not permitted unless the compared columns come from the same domain; this is supposed to insure that the "same kinds" of things are being compared. However, domains can only be specified for single fields (columns); there is no mechanism for indicating that multiple columns represent "entities" from a single domain, as is the case with composite keys. Thus, in the present example, dependents are identified by two fields, one from the domain of "first names" and one from the domain of "employee numbers" (or "employees"). Nothing in any record description would mention a domain of "dependents". If joining is permitted on multiple fields simultaneously, then these records could be joined with any records also containing two fields whose domains are "employees" and "first names" — no matter whose first names they might be: the employee's own, his dependent's, his manager's, or anyone else who might share a relation with him. There is no way to constrain records to be joinable only with

other records that refer to dependents; hence, any number of spurious relationships might be implied.

The following is a valid join. What does it signify?



8.8.4 The Reducibility Ambiguity

The theory of irreducible records (which, in the logical development of this book, isn't explained until section 10.3), encounters a severe ambiguity, which can be blamed precisely on the use of composite names (including qualified names). This highlights the confusion over which entities are involved in a relationship, and which facts are actually being represented in a record; and it illustrates how such an analysis is unduly affected by the choice of names for the entities involved.

Consider a person's birthday. On the face of it, this is an elementary fact — a simple binary relationship between a person and a certain day in the past. And, if we happen to represent dates in Julian notation (one field), then birthday actually has the structure of an elementary fact. But if we choose to change the *naming* of the date to the more conventional notation involving three fields, then we have a record containing four fields. This record can now be reduced to three binary records:

- Person and year,
- Person and month,
- Person and day of month.

The original birthday record can always be recovered by joining these three.

The same analysis, and ambiguity, applies whenever a composite naming convention is selected for an entity. City of birth, for example, is an irreducible fact if globally unique city codes are used; it is reducible if the city is identified by the composite of, e.g., city, state, and country names.

The analysis of the structure of information will always be confused and ambiguous if carried out in terms of record based concepts such as fields and data items, rather than in terms of the underlying entities. Composite names are in general *not* precisely equivalent in function to simple unique identifiers for the same entities. Composite names almost always convey additional information; when used in lieu of simple names they necessarily change the underlying structure of the information. A simple name simply designates an entity; a composite name does that, but it simultaneously informs us about other related entities. A city code simply designates a city; the conventional notation may additionally tell us the state and country in which it is located. A Julian date simply designates a certain day (if we don't bother to do certain computations); the conventional notation additionally tells us the year and month in which it occurred, as well as the day of the month.

This dual role of composite names precisely parallels the ambiguity of reducibility. In the role of designating a single entity, it could be part of an irreducible fact; in the role of providing auxiliary information about related entities, it leads to reducibility.

This kind of duality leads [Chen] to classify relations into two types, regular and weak, depending on whether the entities involved are identified by simple or qualified names. This is a curious situation: the nature of the *relation* is considered to be different according to the method of naming the related entities.

A precise model of information should distinguish carefully between the structure of entities being modeled and the various structures of names that might be associated with them. This implies a distinction in the model between entities and traditional data items.

8.8.5 Another Ambiguity

"Intersection data" — data about relationships — also leads to irreducible records with three fields, with two of the fields serving as a composite key. For example, the nature of the kinship between a dependent and an employee would be represented as:

Unfortunately, this configuration is indistinguishable from the form in which information is recorded about entities that happen to have qualified names. If, as earlier, dependents are identified via a qualified name including the related employee, then the age of the dependent would be recorded as follows:

This fact is really about the dependent alone, while the previous one was about the relationship between the dependent and an employee. But the structure of the two in "irreducible" form is indistinguishable. Thus, if naming conventions are not separated out from entity representation, "irreducible" records still do not model the structure of information unambiguously.

To see the significance, compare what happens to the preceding two structures if dependents were named simply, e.g., by social security numbers.

In unreduced records, a composite key is likely to be serving both roles simultaneously. It would not be unusual to see the two records shown above combined into one (since they have the same key), containing both age (a fact about the dependent) and

kinship (a fact about the relationship). It is thus ambiguous as to which entity is really represented by this record.

8.9 Records Are Useful

The record concept does serve a useful function in defining certain groupings of information.

In actually doing data processing, a record constitutes a package of information to be inserted or deleted, according to some pre-established conventions (via record type descriptions).

Such packages of information are also useful in managing the different views which different applications have of the underlying data, and in controlling the subsets of such data which different applications are authorized to access.

This is still another function of the record type name: it names a particular grouping of information, for such purposes as view or authorization management. This is much more appropriate than some of the other significances attached to the name, which is sometimes taken to signify a subject (entity type) and sometimes a relationship.

8.10 Implicit Constraints

It's also worth noting that, because a record gets created or destroyed as a unit, it imposes an implicit constraint on the various pieces of information collected in the record. In particular, it imposes 1:1 correspondences between various sets of entities. Maintaining an employee number, department number, and salary in the same record guarantees that the set of employees which have salaries is exactly the same as the set of employees which are assigned to departments (in the absence of null values, of course). This is a hidden constraint that one might argue should be asserted explicitly in an information model. (There's nothing wrong with *implementing* that constraint by storing the data in record structures.)

9 The Other Three Popular Models

In this chapter we comment briefly on the relational, hierarchical, and network models. There are two excuses for the brevity of this chapter. First of all, I don't really explain the models, but just make some comments about them. Secondly, most of my concerns have been factored out to the previous chapter: much of what I've said about records carries over into these three models.

To get more information about these data models, I would suggest the following as starting points: [Date 77], [Martin], [Chamberlin 76a], [Taylor], [Tsichritzis 76,77], and [Senko 77a].

"Ambiguity" is the principal theme of this chapter. Most of the comments on the three models focus on the diverse views from which each of them may be seen. When reading the literature on these models, try hard to get an understanding of the particular definition the author has in mind for the model in question. Try to determine which features he assumes are included, and which are not. It can be quite an experience to compare several papers nominally dealing with the same model.

9.1 The Relational Model

There is a mathematical definition of a relation, which is essentially the idea of a tabular structure. A relation of degree three, such as the one among parts, suppliers, and warehouses, may be represented as the set of rows in a table — i.e., as a set of triplets, with each triplet naming one part, one supplier, and one warehouse.

The pure form of the mathematical relation allows a single box in the table to contain a set of things (a person may be related to the set of his children, with all their names listed in one square of the table). The relational discipline of "first normal form" excludes this from the relational model.

Such formally defined relations (tables) do not always correspond exactly with the intuitive concept of relationships.

The correspondence is good in one direction: every relationship of degree n can be modeled as a table with n columns. But not every table with n columns corresponds to an intuitively satisfying relationship of degree n. Many such tables really model entities (e.g., employees), together with an assortment of single-valued relationships and attributes for that entity (department, spouse, salary). The "real" relationships here are the separate ones between employee and department, employee and spouse, and employee and salary. It is only in a very formal, artificial sense that a relationship of degree four exists here.

The most highly visible feature of the relational model is its tabular data structure. People who take this as its principal characteristic claim that anything which supports a homogeneous linear file (e.g., just about any old fashioned record processing technology) supports relational data. In this view, the relational model is no different from the record model discussed previously.

A more significant feature is that all relationships (paths) among "records" are based on symbolic associations, i.e., matching field values. In the model perceived by users, there are no manipulative operations that depend on pointers, adjacency, or other hidden forms of linkage. (See sections 4.6, 10.5.2.)

Note carefully that that last claim was not made for all relationships, but only for relationships between records. One should realize that there are two mechanisms for expressing relationships in the relational model: symbol matching, and coexistence in a row. A very large number of relationships, such as the one between an employee and a department, are in fact represented by whatever internal glue it is that holds the fields of a record together.

Another distinguishing feature is the set-oriented nature of the operations, which deal simultaneously with sets of records instead of processing them one at a time.

Still another view, reported by [Robinson], identifies the relational approach with the use of a language based on the predicate calculus, such as "Alpha" [Codd 71a].

Functional dependences, a normalization concept, candidate keys, and similar phenomena are involved in the definition

(creation) of relations. They seem to have an ambivalent role on the fringe of the relational theory. On the one hand, they do comprise a central part of the theory with respect to the definition of "proper" relations (cf. [Bernstein]). On the other hand, prototypes and implementations of relational systems simply presume that relations are normalized. Although some mathematical results concerning functional dependences have been developed ([Delobel], [Bernstein], [Armstrong], [Schmid 75], [Fadous]), there is to my knowledge no existing system which processes functional dependences. And functional dependences never seem to be mentioned when the relational model is compared with others. Furthermore, they seem to be overlooked by proponents of the relational model themselves, when they claim that a strong feature of the relational model is the symmetry between descriptive and manipulative facilities.

Constraints (enforced rules on the valid contents of relations) sometimes seem to be part of the relational model ([Eswaran], [Hammer]), and sometimes not.

The "domain" concept itself has an uncertain status. In most definitions of the relational model, values in a column are constrained to come from a named domain (generally distinct from the column name). Furthermore, joins are only permitted by matching columns that have common domains (perhaps to insure that a common entity is serving as the "pivot" in the implied relationship). But some papers, and most implementations, only have a column name — no domain construct. Joins can be formed on any numbers or letters which can be compared, e.g., one can relate two people if the height of one is written down in a form that looks like the age of the other. Anyone 72 years old is related to me, since that's my height in inches.

It is sometimes argued that the domain construct limits joins to "sensible" relationships. Others argue that anything you can match implies a relationship; sensibility is in the mind of the relator.

Even when there is a domain construct present, it often does not attempt to define common entities. All too often, domains are defined in terms of various classes of character strings (e.g.,

integers, complex numbers, alphabetics, etc.). These are almost as bad as having no domains at all.

More progress: define separate domains for things with common units of measure, but which aren't really comparable entities. [Eswaran] illustrates this by distinguishing the domains for an employee's height and his distance from work.

But even if there is some attempt to relate to entities, it is invariably sets of symbols that get defined, not sets of entities. I have never seen a relational domain for employees, or for dates. The domains are always of the form: employee number, or social security number, or person name, or Julian date, or standard date — each being a distinct domain. Thus one can't match columns on the basis of common entities, but only on the basis of common names (representations). The matches are thus always subject to the failures mentioned in section 3.9.2. You can't match on the basis of the same people occurring in two columns, if they are represented by employee numbers in one and social security numbers in the other.

The construct in the relational model which would come closest to an effective domain concept would be a unary relation (e.g., a one column table of city names), linked to other relations by integrity constraints which require a city to exist in this list before it can be mentioned anywhere else. The practice of defining the "domain" of cities in syntactic terms (by data type) is much weaker: the only constraint really in effect is that a city is anything with an alphabetic name (section 2.4.3). This domain is in no sense equal to the set of city names.

9.2 Hierarchies (IMS)

In general use, the term "hierarchy" refers to a system of stratifying or ranking things one above another. Upper, middle, and lower class represent the hierarchical structure of certain societies. The various titles within the British nobility comprise another hierarchy. [Smith 77a] speaks of a "hierarchy of relations" in this sense.

In a more restricted usage, the term can mean a tree-like system of relationships that tend to branch out in one direction.

An organization chart is an example: a person may have several subordinates, each of whom may in turn have several subordinates, etc. Or a parts breakdown: an item is made up of several sub-components, each of which in turn is broken down into its own sub-components, and so on until the elementary parts are identified. These trees represent "one-to-many" relationships. Traversing the tree in one direction, a person may have many subordinates; but going in the other direction, a person can have only one superior.

Trees might represent relationships among different kinds of things, as in the grammatical analysis of a sentence. Subordinate parts of such a structure might represent various kinds of clauses, phrases, grammatical classes (nouns, verbs, etc.) and, finally, individual words. A certain kind of thing, such as a noun phrase, might occur at various levels. It might be a direct constituent of the sentence, or it might be a subordinate part of a clause, and so on. Individual words might occur at any depth in the tree, depending on the complexity of the grammatical structures in which they occur.

A further restriction on the hierarchical structure fixes the levels on which various kinds of things may occur. Each kind of thing occupies one specified level in the structure; things of the same kind cannot occur at different levels of the hierarchy. A corporate structure might be represented in this way, with the corporation at the top, its divisions at the second level, the departments on the third level, and the employees on the fourth level. While a given kind of thing cannot occur on more than one level, several kinds of things might occur on the same level. The corporate tree might also have on the third level the products manufactured by each division, and on the fourth level it might also have the capital equipment owned by each department. Hierarchies in which the same kind of thing might occur at many levels, such as the personnel organization chart, the parts breakdown, and the sentence analysis, do not fit this restricted structure.

The common jargon for describing such structures has absorbed some very mixed metaphors. The structure is described as a "tree", but it is almost always perceived upside down, with

"root" being a reference to the top of the tree. In addition, the terms "parent" and "child" are frequently used to denote relative positions in the tree. In the corporate structure, a department is the "child" of a division, and the department is also the "parent" of an employee.

The "hierarchical data structure" supported by such systems as IMS corresponds to the most restricted form we have described. It is a tree-like set of one-to-many relationships, in which each kind of thing occurs at a single specified level of the hierarchy. One database (or file) in IMS consists of a family of trees, each having the same "pattern". Thus, one file may contain a number of corporate trees of the form described earlier. At each node (junction) of the tree corresponding to some object, there is a "data segment" whose fields contain information about that object. For example, a data segment at the division level of the corporate tree might contain the division name, the name of its president, and its headquarters location.

IMS uses the term "record" in a different way than I have in this book. An IMS "record" (sometimes called a database record), consists of one entire tree, from one "root" segment through its lowest related segments. In our example, it would include all the data about one corporation and its substructure. My use of the term "record" — a contiguous sequence of fields transmitted as a unit — corresponds more closely to an IMS data segment.

The one-to-many relationship is central to the IMS concept: a segment has exactly one parent segment (except a root, which has no parent). However, IMS does allow the definition of structures that are not so constrained. Using a construct called "logical relationships" one can, for example, define a department segment as being subordinate to a "geographical region" segment as well as to a division segment. Thus a department might in fact have two parents: a division and a region. Logical relationships can also be used to (indirectly) represent many-to-many relationships, such as the one between students and classes. While such structures can be defined, programs cannot process them (as a single file). For processing purposes, it is necessary to define subsets of the data ("logical files") which

only encompass tree-like structures. Such a file might see departments as subordinate to divisions or to regions, but not both. Thus the IMS concept of tree-like structures is imposed on the files which programs can process, but not necessarily on the underlying data maintained by IMS.

On the other hand, a program can process several such files at the same time. It can in fact perceive that a given department is subordinate to a certain division and also to a certain region, by looking into the two logical files. (Technically, one can claim that it is not the "same" department segment that has these two parents, since a separate segment is perceived for the department within each file.) Thus, while the IMS data model is generally held to be a tree-like hierarchy, it can in fact sometimes appear to be a more complex structure.

Curiously enough, in some respects the IMS structure looks less complex than a hierarchy. For data retrieval, the file looks essentially like a flat, linear sequence of segments. There is a standard, IMS-defined ordering of these segments (top to bottom, left to right), and data retrieval operations are defined in terms of this sequence. The principal operations supported by IMS consist either of retrieving the segments sequentially or of skipping forward to the next segment of a specified type containing a specified value. The semantics of retrieval operations are always explained in terms of this linear ordering. Certain operations which might be thought characteristic of tree structures are not provided, such as moving upward (from a segment to its parent), or detaching a sub-tree and re-attaching it at another point. (Strictly speaking, it is sometimes possible to move "up" as well as "down" over certain data, if appropriate logical relationships are defined and a separate file is used for each direction of travel.)

The situation with IMS and hierarchies is the same as with most implementations of data models. There usually are important differences between the implementation and the naive, abstract view of the data model. Several implementations of the same data model may behave quite differently. In general, criticisms and comparisons should begin by clarifying whether the subject in question is a data model or an implementation.

They aren't the same: networks and the DBTG model. I'll come back to that soon.

To me, the central semantic innovation of DBTG is the named relationship — that's what their "sets" really are. It is the only major model which provides this as an integral part of its semantics, but its significance seems to be largely unnoticed (although Bachman has remarked "I consider data structure sets as representing natural relationships which exist in the real world" [Bachman 75]).

Unfortunately, DBTG only grants such status to relationships that are one-to-many. Many-to-many relationships still have to be recorded in "intersection" records; in these cases it is the records, and not the sets, which represent the relationship. It is sometimes argued that DBTG supports many-to-many relationships by providing such a mechanism, but in fact the system does not know anything about this. The system doesn't know the difference between an intersection record and a record that happens to be a "member" in several sets (see section 8.4.3). DBTG doesn't "support" many-to-many relationships; it doesn't even know that they are there.

Some people equate DBTG with a ring-structured implementation of it, and they deplore its "spaghetti" of pointer chains.

Still others focus their critical comparisons on the manipulative language specified in the DBTG proposal. Such critics fail to see the possibility of developing a better language (exploiting the semantics of named relationships), much as a language like SQL shields users from relational joins and projections. Such critics also overlook the disclaimer on page 7 of [CODASYL 71].

In particular, some associate DBTG with a procedural language, contrasting it with the set-theoretic language of the relational model. The procedurality is sometimes labeled "navigation".

There are criticisms that focus on parts of the DBTG proposal which have nothing to do with the data structure (things like database keys, currency indicators, realms — cf. [Engles 71]). Objections to such features should not be misconstrued; they are not objections to the basic model.

The developers of DBTG judiciously chose the word "network", rather than "graph". "Graph" is a mathematical term, having some well-defined and much studied properties. "Network" informally conveys the same idea as "graph", without any real obligation to retain the same properties. Lest anyone confuse the two, let me indicate how a DBTG network differs from the general form of a directed graph with labeled edges:

- Nodes in the network are not elementary items. They are records. Thus much of the data content is already aggregated out of the graph structure.
- Nodes are partitioned into classes called "record types".
- Many edges in the graph bear the same label. A DBTG "set" consists of all the records connected by edges bearing the same label. That label is the name of the set (and also the name of the relationship that associates these records).

"Owners" are the records at the tails (sources) of the directed edges. "Members" are the records at the heads (targets) of the edges. A DBTG "set occurrence" consists of one owner record and all the members to which it is connected by edges labeled with the set name.

- All edges bearing the same label must emanate from nodes of one type (a set has only one owner record type).
- A node may not be the target of two edges bearing the same label (a set is a 1:n relationship).
- An edge may not connect two nodes of the same type (a set's member record types may not include the owner record type).

Hence a homogeneous graph (all nodes of the same type), such as an organization chart, cannot be directly represented in the network. It can be represented indirectly by introducing an "intersection record" as a second record type, and using two labels: one for the "from" sense of the association (manages), and one for the "to" sense (is managed by).

The same device is required to represent m:n relationships. It is sometimes claimed that such constructs "represent" a generalized graph in the network; all you have to do is visualize the intersection records as part of the edge structure, and not think of them as nodes.

- As a corollary, an edge cannot connect a node to itself.

Now a caveat: not everything I've said about DBTG is true. Since I first wrote this material, some changes to the Codasyl specifications have been adopted, and others are being considered. Which only goes to show that we are dealing with a moving target. You have to be very careful when discussing "the network model" to establish whether you have in mind some fixed, abstract structure, or whether you are referring to the Codasyl specifications, and, if the latter, which version.

10 The Modeling of Relationships

Having examined some of the major data models, we are now in a better position to explore some other issues and problems in the modeling of relationships.

10.1 Record Based Models

As suggested in section 8.3, the record oriented approach fosters an asymmetry in the treatment of relationships. One-to-many relationships are given fundamentally different treatment from many-to-many relationships. [Martin] refers to many-to-many relationships as "complex plex structures", and says "The reason for making the distinction between simple and complex plex structures is that the latter need more elaborate methods for representing them *physically*."

This treatment arises from the traditional aversion to repeating fields in a record, and from the convenience of lumping two things into the same record when there happens to be a singular relationship between them.

These options are all in addition to representations of relationships by means of file structure. These might include such techniques as record sequencing within a file, or placement within a hierarchy or CODASYL network. Even here the choices are constrained by the semantics of the relationship. Structural representations typically can only be used for certain kinds of relationships, and not for others. For example, the structural representation in hierarchies (of the IMS variety) may not be used for relationships between records of the same type.

The difficulty with respect to information modeling is what to do with this plethora of options ([Codd 74], [Nijssen 75]). Why is it necessary to make such choices? What are the criteria? Do the criteria have anything to do with the semantics of the information, as distinguished from the economics of storing or processing the data? Do all users have to know which options have been chosen, and to adapt their processing accordingly? Is

there any implied unavailability of other options, e.g., could one still retrieve a department record containing a list of employees?

The techniques an application employs need only be concerned with the complexity of a relationship in the direction the application is traversing it. The only difference it really makes to the application is whether it should expect, and allocate space for, one or many items in response.

Unfortunately, the choice of representation too often does show through to impact the way users use the information, and the way application code has to be written. If someone wanted information about all employees in a department, there are likely to be different kinds of paths for current and for past assignments. For current assignments, we can go directly to the employee records; for past assignments we must navigate indirectly via the employment history intersection records. Yet in both cases we are dealing with a relationship between employees and departments. In this respect, listing the employees in a department need be no different from listing the parts in a warehouse. It doesn't matter to these applications that the parts might also be in other warehouses, whereas the employees cannot also be in other departments (i.e., when traveling in the complex direction, it needn't matter whether the relationship is 1:n or m:n).

In current technology, such applications have to use two conventions for these cases, following direct linkages for the employees in a department, but detouring via intersection records for the parts in a warehouse. And, if corporate policy changes to permit employees to have several departments, then the first application has to change the way it asks the question, even though the question and the answer are unchanged.

The choice of representation also affects the way applications do updates. For example, moving an employee from one department to another is done by a simple record update, if the department number is kept in the employee record. But if the relationship is kept in an intersection record, then changing the department number is changing the key. Changing keys creates problems in many systems, and often is not permitted at all. In

such cases, this "update" must be done by deletion and insertion of records.

For descriptive purposes, it might be desirable to seek a method for declaring such relationships and their semantic characteristics directly, without having to choose among such a variety of representational alternatives.

10.2 Binary Versus N-ary Relationships

An "n-ary" relationship has degree n. "Binary" and "ternary" relationships have degree two and three, respectively.

Relationships of degree greater than two can be described as combinations of binary relationships. This follows from the observation that an instance of a relationship is itself a thing that can be related to other things. There are two ways of reducing n-ary relationships to structures of binaries. We will describe one here, and introduce the other in section 10.4.

Consider a ternary relationship among parts, warehouses, and suppliers, wherein a given part may be ordered from certain suppliers for one warehouse and from different suppliers for another warehouse. We can start with the binary relationship between parts and warehouses, i.e., which part is stored in which warehouse. Let's call this relationship "allocations", and call each instance of a part assigned to a warehouse an "allocation".

Notice the influence of language on our thinking. By having the single noun "allocation" for it, we can comfortably think of the instance of the relationship as being a thing in itself. Calling it "a part stored in a warehouse" doesn't give it the feeling of a "thing".

We next define the *binary* relationship between suppliers and allocations. This binary relationship literally identifies which supplier services which allocation. But, through our understanding of allocations, we know that it really means which supplier sends which part to which warehouse. We have thus defined the ternary relationship as a pair of binary relationships, which we can denote symbolically as PW and S(PW).

We use the following notation: "PW" stands for the binary relationship between parts and warehouses. "SPW" stands for

the ternary relationship between suppliers, parts, and warehouses. “S(PW)” stands for a particular decomposition of this relationship; it is the binary relationship between suppliers and the binary relationship between parts and warehouses — i.e., it is the binary relationship between suppliers and allocations. For simplicity, we are assuming that one domain is the same as one category. Also, the ordering of domains is not relevant to this discussion, and we can therefore ignore permutations of letters. PW is the same as WP; SPW, PSW, PWS, etc., are also all equivalent to each other.

Of course, S(PW) is not the only way we could have decomposed the relationship. An equivalent verbal description of the ternary relationship is that when we order the part from one supplier we may have it sent to certain warehouses, while we may have other suppliers send the same part to other warehouses. Now we are thinking first of a binary relationship between parts and suppliers, which we can call “zorgs” (because I can’t think of a good descriptive noun). Then we construct the binary relationship between zorgs and warehouses. This time we have decomposed the ternary relationship into the two binary relationships PS and W(PS).

And, in similar fashion, we can rationalize a third decomposition, into the pair of relationships WS and P(WS).

There are always multiple ways to decompose relationships of higher degree into binary relationships. We have seen the three possibilities for decomposing ternary relationships. For degree four, there are 15 different ways: PSWT can be decomposed into the forms P(SWT), S(PWT), W(PST), and T(PSW), with the ternary relation in each of these being decomposable three ways, plus the forms (PS)(WT), (PW)(ST), and (PT)(SW).

An example of the form (PW)(ST) is interesting to look at, because it illustrates how instances of relationships are themselves things which can be related to each other. We can take P and W to be parts and warehouses as before, with the relationship PW still called “allocations”. S is still suppliers, and T is now truckers: PSWT is the relationship that certain suppliers use certain truckers to deliver certain parts to certain warehouses.

There is a relationship ST between suppliers and truckers — which suppliers use which truckers — and we can arbitrarily call these "sources". A "source" is a combination of one supplier and one trucker. The relationship PSWT can now be expressed as the binary relationship (PW)(ST) between allocations and sources: which source services which allocation. This new relationship is relating instances of two other relationships.

10.2.1 Simplicity

The reason we dwell so long on these decompositions into binary relationships is that some important data models are founded on such decompositions. From some points of view binary decomposition may be the best and simplest way to describe all relationships.

The main advantage of such decompositions is that they permit any relationship to be built out of a single kind of building block (the binary relationship), giving the description of all relationships a very regular format [Titman]. Hence, it is "simpler".

Whether or not you believe that depends on what you mean by simplicity. Simplicity is very much a subjective notion, and we can give it at least two interpretations. On the one hand, simplicity in a descriptive language (for data or programs or anything else) means minimizing the number of different terms that can be used in the description, thus minimizing the number of terms that have to be learned by users and implemented in the system. The regularity of binary relationships provides this kind of simplicity. According to this criterion, Roman numbers are simpler than Arabic, since they use fewer different characters [C&A 70]. For that matter, binary notation must be the simplest of all. Also, by this criterion, the simplest computers of all are Turing machines. Would you like to program your application on one?

On the other hand, simplicity is achieved when descriptions can be written concisely, and can be carried on paper and in the mind as single broad concepts rather than complex bundles of tiny concepts. This kind of simplicity is provided, for example,

in the relational model, which can directly accept a definition of a PSWT relationship, rather than requiring a sequence of definitions such as WT, S(WT), P(S(WT)). This kind of simplicity is achieved at the cost of requiring the system to understand a larger variety of constructs (i.e., relationships of all possible degrees). It is often argued that this wealth of additional constructs is redundant and adds no new function, since everything can be expressed in terms of the "primitive" constructs, e.g., binary relationships.

We can further illustrate the concepts, and problems, in a system for specifying arithmetic expressions. Consider one language that has terms for expressing addition, division, and counting, and another language that has all these plus a term for "average". Both languages have the same functional capability for describing things. The first is simpler because it has fewer terms to be defined, learned, and implemented. But the second language is obviously easier to use if you want to compute an average. To average a LIST of values, one writes

X = SUM(LIST) DIVIDED-BY COUNT(LIST)

in the first language, and

X = AVERAGE(LIST)

in the second.

We might observe that each kind of simplicity is appropriate to a different level in the system. The first language is appropriate to a low, internal, implementation level of the system, where implementation cost is of primary concern. The second language is appropriate to a higher external interface that is seen by users. Some kind of transformation process is required between the levels, such as a language translator which takes the second statement above as written by a user and transforms it into the first statement above to be executed by the system. This corresponds, for example, to the implementation of a relational

database on top of an interface dealing only in binary relationships.

This split-level approach has some disadvantages. Since the direct intent of the user is not transmitted to the underlying system, the system may not be able to optimize and perform the function in the best possible way. In the averaging example above, the system executing the first statement is likely to take two passes through the list, once to accumulate the sum and again to count the elements. If the system understood "average" directly, it would do both in one pass.

In a nutshell, simplicity can mean either a small vocabulary or concise descriptions. Both have their value.

Incidentally, let me mention still a third kind of simplicity, which may be even more important than the other two in the area of data description. This is "familiarity". The easiest system to learn and use correctly may well be the one that is closest to something already known, regardless of how objectively complex that may be. It is precisely this phenomenon, for example, which makes the metric system of measurement much *less* simple for me (and many of my readers) to use, although it is obviously simpler by any objective criterion. The trouble with this approach, of course, is that it is subjective and depends very much on who the users are. How do you measure it? And does it require supporting a number of systems, each "familiar" to a different group of users?

And there is this hazard: the apparent familiarity can also lead users astray, in those cases where the system does not behave the same as the thing they are familiar with.

10.2.2 Unnecessary Choices

Another concern arises from the fact that in describing an n-ary relationship as a composition of binary relationships, one has to select one of the many possible compositions. That is, one has to decide whether to specify PWS as P(WS), W(PS), or S(PW). There is often nothing in the way that users think of this relationship to prefer one form over the other two. Hence this arbitrary choice should not have to be made in the conceptual

model of the information system. Also, there is a danger that this form of specification may become entangled in implementation and performance concerns. The specification P(WS) suggests an implementation in which people just interested in the relationship between warehouses and suppliers, i.e., WS, will get better service than those interested only in parts and warehouses, i.e., PW. Any such implication in the conceptual model should be avoided. An administrator should be free to change the implementation structures and performance optimizations without having, for example, to respecify P(WS) as S(PW).

10.3 Irreducible Relationships

"Irreducible relations" ([Rissanen 73], [Hall 76], [Falkenberg 76a]) is an area of relational theory which attempts to reconcile record structures with the requirements of accurate information modeling. The general idea is to model information in terms of elementary facts, generally (but not always) leading to binary relationships analogous to records having just two fields. For example, an employee record that included his spouse and birthplace would be reduced to two records, one containing the employee and spouse and the other containing the employee and birthplace. The original record can always be recovered from these by combining the known information about the employee (in relational terms, by a "join" operation).

This recoverability aspect is the essential test of reducibility. A record is reducible if shorter records can be defined which can be so combined to recover the original record. Such a reducible record has not been decomposed into its elementary facts. In contrast, an irreducible record is considered to represent one elementary fact, since it cannot be reconstructed from smaller units of information. Note that the danger is not that information will be lost, but that the reconstruction process will generate spurious (and false) data — as the next example illustrates.

An example of an irreducible record with more than two fields would be one representing the ternary relationship between suppliers, parts, and warehouses ("which supplier ships which part to which warehouse"). If we try to reduce this to shorter

records, e.g., one with suppliers and parts and one with suppliers and warehouses, we cannot accurately recover the original information. These short records might tell us that a certain supplier supplies a certain part, and that this same supplier services a certain warehouse — but we can't be sure that he ships that part to that warehouse. A reconstruction from the short records will generate that combination, whether or not it is a true fact. A similar analysis holds if we try to reduce to other pairs, e.g., a record with parts and warehouses plus one with parts and suppliers. Thus in this case the records with three fields are irreducible — they represent elementary facts.

This modeling approach has the advantage of modeling the actual structure of the information. Elementary facts are clearly identified, and the structure is more describable: a record type name (relation name, in the relational model) can correspond to the relationship or attribute expressed in the fact, and field names can be used to name the roles played by the entities associated by the fact. The approach still suffers from some of the record structure problems, e.g., those having to do with synonyms and with the representation of relationships having multiple entity types per domain (as mentioned in section 8.8.4).

Reducing records (or relations) to such elementary facts has a side effect that is considered by some to be a disadvantage. Keeping things bundled into one record type implicitly enforced a co-extensiveness constraint. In effect, the set of employees who earned salaries always had to be exactly the same as the set of employees who were assigned to departments. By separating these into independent elementary facts, we introduce the possibility of adding or deleting one fact separately from the other. The constraint that had been implicit in the structure now has to be made explicit, to the effect that employees may earn salaries if and only if they are assigned to departments.

10.4 Good and Bad Binaries and N-aries

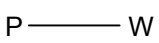
There are some differences of opinion concerning the relative merits of binary and n-ary relationships, and also concerning the merits of the relational model.

Where do I stand? There happen to be just enough ambiguities in the definitions of the various models that I cannot answer that question. I face these dilemmas:

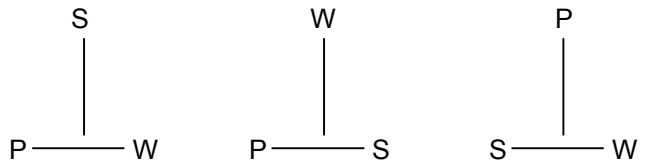
- There are two ways of employing n-ary relations, one of which I consider good and the other bad. So I can't take a stand on n-aries.
- There are similarly two ways of applying a binary model, again one good and one bad (and one might even say that the good one isn't really modeling binary relationships). No simple opinion here, either.
- Most amazingly, the good n-ary and the good binary look the same to me! So, when I tell you what I like, I still can't say if it's binary or n-ary.
- And, for good measure, I can't be sure which of these models is considered to be relational. (But I'm sure many of my readers are. I'd love to poll them.)

10.4.1 The Binaries

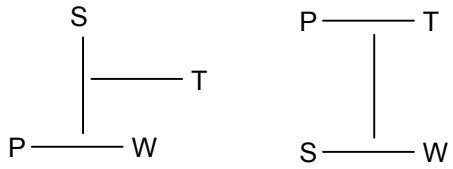
Many things are simpler when we can deal with things pairwise, two at a time. Binary relations fit a simple linguistic model, two objects connected by a relating verb: people own cars, employees are assigned to departments, parts are stored in warehouses, etc. They also lend themselves to a natural picture: a line connecting two points (or nodes).



The pure binary approach forces everything into this pattern, even when more than two things are involved. This exploits the fact that relationships are themselves entities, which can then in turn be related to other things. So, if three things are involved, we first link two of them to generate a new entity, which then gets linked to the third thing. There happens to be three distinct ways of doing this (section 10.2). For suppliers (S), parts (P), and warehouses (W), these are:



I don't happen to think this is a good approach. One shouldn't be faced with making an arbitrary choice among such alternatives. And the structures get more complex than they need to be, especially if even more than three things are involved. When there are four (such as suppliers, parts, warehouses, and truckers), the number of different pictures to choose from is fifteen! You can work them out; here's two samples to get you started:

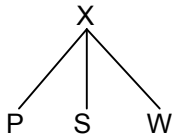


I haven't had the courage, or the patience, to compute the number of ways of combining five things.

At any rate, whether you agree with my opinion or not, those are what I call "bad binaries".

There is another way to perceive the relationships as entities. We can imagine that there does in fact exist a single relationship among the three things simultaneously, and treat that as an entity in its own right. In [Bracchi], a class of such new entities is called an "internal set of concepts" (ISC).

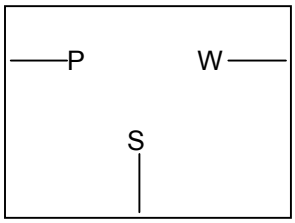
In representing the relationship between a part, a supplier, and a warehouse, each of these is linked to the object representing the relationship among them:



If we want to call this new entity X, we have in effect used the three binary relationships PX, SX, and WX. There is only one such configuration, not three to choose from, and we are still basically using a binary model. Similarly, if Y was a simultaneous relationship among four things, the configuration would be represented as the four binary relationships PY, SY, WY, and TY. Again, there is only one configuration, not fifteen options.

Let's refer to that approach as "pseudo binary". I'll explain why shortly.

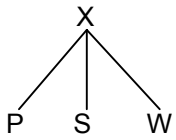
Some people have modified the binary model in a different way. The limiting factor in the familiar graph picture is an elementary bit of geometry: a line segment has two endpoints. Why should we be constrained by an irrelevant geometric truism? Why can't we imagine a "hyper" line segment with many endpoints, as some kind of generalized connector? We can, and the resulting structures are called "hypergraphs" [Furtado]. That is, we can imagine them, but not draw them — at least not with simple lines. But there is a picturing convention for hypergraphs, in which a connector is drawn as a box around the things it connects. Since many of these boxes will overlap (because a thing might participate in many relationships), there can be some confusion as to which things belong to which box. To avoid that, lines are drawn to link things with the boxes they belong to. (Another reason for the lines is so that they can be labeled with the role being played by each object in the relationship.) Thus, a relationship among P, S, and W would be pictured as the box:



This appears to be an entirely different approach, but with a flick of the wrist we can be right back to a familiar picture. This hypergraph approach is in fact isomorphic with the "pseudo binary" approach just described. All we have to do is:

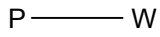
1. Flip P, S, and W outside of the box, taking their connecting lines with them.
2. Shrink the box to a point.
3. Label the new point "X".

And presto, we have a familiar picture of the relationship between P, S, and W:

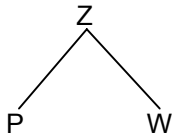


There is one difference: with hypergraphs, one does not have the ability to treat the connectors (boxes) as being points themselves, hence they cannot in turn be connected to other things. Pseudo binaries do have this added capability.

To be consistent, treating relationships as entities requires a new object to be introduced even if only two things are being related. That is, a relationship between P and W ought to be modeled, not as



but rather as

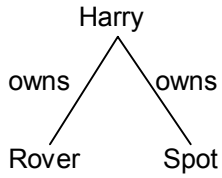


This practice is in fact commonly followed when the relationship happens to be many-to-many, giving rise to so-called intersection records (section 8.3). But the practice is rarely employed for one-to-many relationships. Even [Bracchi] fails to achieve this level of consistency in the treatment of relationships.

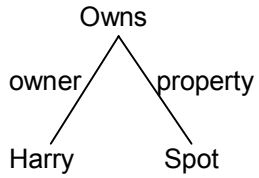
Why have I been referring to these as "pseudo" binaries?

There is a certain hoax being perpetrated throughout all of this. Such models are still labeled as "binary relations" — but it is n-ary relations that are being modeled. They have just been shifted from the lines to the nodes. The pictures look the same for the pure and pseudo binaries — they both involve lines connecting pairs of nodes — but the semantic interpretations of the two diagrams are totally different.

In the pure binaries, the lines are serving to represent the relationships that are being modeled. Just consider what labels would be written along the lines: they are the names of the relationships.



In contrast, in the pseudo binary model the relationships of real concern are themselves nodes. The lines are serving an entirely different function, which can actually be seen a little more clearly by their counterparts in the hypergraph picture. These lines are merely serving as a kind of internal glue, connecting a relationship node with each of the things it is relating. If there are any labels along these lines, they would be role names rather than relationship names.



Thus the similarity between pure and pseudo binaries is very superficial. While there is a trivial resemblance in the formats of their pictures, there really is a deep semantic difference between the two: pseudo binaries are in fact supporting n-ary relationships, while pure binaries require decompositions into pairwise relationships. The pure binary approach denies the existence of relationships involving more than two things at a time. In that view, the shipping of parts to a warehouse by a supplier is not a single indivisible fact. It must be viewed instead as a composition of smaller facts, e.g., fact 1: parts are shipped to warehouses; and fact 2: suppliers perform fact 1. In contrast, the pseudo binary view acknowledges the existence of a single complex fact, and simply draws a picture connecting the fact with each of its participants.

Despite the pejorative connotations of the term, I hope it's clear that I prefer the "pseudo" binary model.

10.4.2 The N-aries

An n-ary relation can be pictured as a table with n columns, each column having a heading. Consider two different relations, having the headings

EMPLOYEE	DEPT	SALARY

and

PART	WAREHOUSE	SUPPLIER

In developing an abstract model of information, one might have an intuitive desire to indicate which are the "elementary facts". One can have an intuitive feeling that the first relation above is a conglomerate of two elementary facts, since we can speak of the department of an employee independently of his salary. In contrast, one might feel that parts, warehouses, and suppliers comprise a single interdependent fact, since a given part for one warehouse comes from one supplier, while the same part for another warehouse comes from another supplier.

Some people make a counter-claim: you can just as easily perceive that a department having an employee making a certain salary is in itself a single fact inter-relating three things. Thus we have some difficulty in objectively sorting out "elementary facts" from hodge-podges of multiple facts.

There is one objective criterion that has emerged, in the notion of "irreducible" relations, which we covered in section 10.3. Those are my "good n-aries".

In comparing irreducible n-aries with pure binaries, one might find an analogy with linguistic terms: the singular n-ary form might be likened to the "deep structure" of a sentence, while the multiple ways of decomposing it into pure binaries corresponds to the multiple "surface structures" (sentences) that have the same meaning.

10.4.3 A Vanishing Distinction

To sum up on the issue of binary vs. n-ary: I do not see the desirability of being limited to purely binary relationships, hence I prefer n-ary relationships. That does not mean that I defend n-ary relations — I only like the kind that are irreducible.

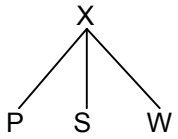
But I'm not opposed to binary relations either. That is, I do like the pseudo binary model — which most people still refer to as a binary model, although it models n-ary relationships.

Have I made that perfectly clear?

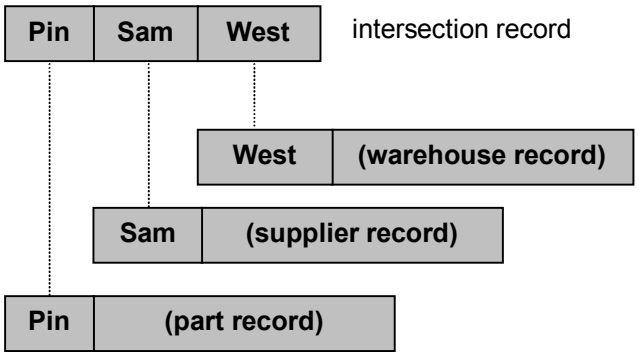
It's really simple: out of all this assortment, I prefer one model — which is simultaneously the pseudo binary and the irreducible n-ary. Let me try to explain why I see negligible

difference between them, at least in the essential structure of the model of a ternary relation.

In the pseudo binary model, we link four objects to model a relationship among a part, a warehouse, and a supplier:



In the irreducible n-ary relational model, we also have four objects. There is one record each for the part, the warehouse, and the supplier, containing information about those entities. And there is the intersection record with three fields, containing the keys (identifiers) for that part, that warehouse, and that supplier:



We can make the geometries of these two representations look alike. We are only dealing with differences in the portrayal of the linkages.

In the pseudo binary model, we draw explicit lines, suggesting some kind of internal pointer mechanism in the implementation. In the relational model, on the other hand, linkages are discovered by matching symbols (in this case, the identifiers in the intersection record match the keys of the other records). If we draw lines between the matching symbols, we see

a topology quite identical to that of the pseudo binary model. Thus, if we ignore the specific technique for achieving linkages, the two models look quite alike.

10.4.4 Case Models

There is still another path of development converging on this general model form. [Furtado] observes that linguistics based models portray a case structure for sentences that looks very much like our pseudo binary diagrams. The sentence is the n-ary relationship object, its constituents are the objects it is relating, and the lines represent the cases (roles) assumed by those objects in the sentence.

10.5 Which Relationships Are "In the System"?

In section 2.3 we observed that some information is explicit in the system, some is implicit (derivable), and we can't always sharply distinguish the two cases. The same obtains with relationships.

10.5.1 Explicitly Defined Relationships

Explicit declaration of relationships permits their properties (such as those described in sections 4.2 and 4.3) to be specified directly to the system, to be enforced independently of the representation of the relationship.

Unless names of relationships are known as part of the data content of the system, it is difficult to answer such queries as:

- "What relationships exist between x and y?"
- "In what relationships is x involved?"

When the manipulative interface is expressed directly in terms of named relationships, then there is considerable latitude in the manner of representing the relationships, with the alternatives being hidden from the user. Representation options include:

- Symbolic linkages via matching field values.
- Internal linkages such as pointers.
- Inclusion of such linkages with other attributes of the entities involved (e.g., pointing to or naming each other).
- Inclusion of such linkages in a separate structure (e.g., intersection records) that represents the relationship.
- Computational procedures, such as composition of other relationships, or comparison of attribute values. (Such procedures could only be used for inquiry, not for modification of relationships.)

With named relationships, the syntax and semantics of queries can be made simple and uniform, independent of the method of representing relationships. (Also, the form of the query is likely to be closer to the natural language form.) For example, an inquiry regarding an "is employed in" relationship could conceivably be handled by a procedure that searches employment history records, looking for the department to which the employee was most recently transferred. An assertion that someone "is employed in" a certain department could be handled by making a new entry in the employment history to the effect that the employee was transferred into that department today (or else the assertion might also provide an "as of" date). Some users would appreciate not having to know that this was the implementation of the relationship.

As another example, consider the inquiry "find all employees located in Stockton". In a record oriented model without adequate capability for naming relationships, the user is obliged to discover that locations are specified in department records. This user has to formulate a query which selects the records of departments located in Stockton, and then finds the corresponding employees. In SQL ([Astrahan 75], [Chamberlin 74]), for example, the query would take the form

```
SELECT NAME
FROM EMPS
WHERE DEPTNUM IN
        SELECT DEPTNUM
        FROM DEPTS
        WHERE LOC = 'STOCKTON'
```

In contrast, if the user were provided with a defined relationship named "located in", then he need not know whether location information is contained in employee records, department records, or division records. This user is simply interested in who works where; he need not be responsible for knowing current corporate practices regarding the centralization of divisions or departments.

This approach works better for inquiries than for updates. To change an employee's location, one does have to know whether employees can move about independently, or are constrained by the location of their departments or divisions. This knowledge can be gleaned from the organization of the relations — i.e., the functional dependences — in a relational database. It can also be specified in consistency rules in a model that describes relationships directly.

Named relationships need not be implemented by internal structures such as pointer chains. They could be provided by means of "macro" facilities in the interface languages. For example, a macro facility could conceivably be added to the SQL language, whereby a macro named "located-in" could be defined to expand into the SQL text illustrated above. An end user might then formulate his query as "find employees located in 'Stockton'", without knowing or caring about macro expansions, functional dependences, pointer chains — or even network vs. relational models.

Of course, that same "located-in" relationship could also be presented to the user in the tabular form of the relational model. There could be an interface at which the (apparent) existence of such a relation is maintained for the user, independent of the manner in which it has to be materialized from the real underlying data (cf. [Boyce]).

Having named relationships as an integral part of the model is much the same idea as perceiving the model as a set of functions [Folinus]. The external user understands the information system by knowing the names and descriptions of the functions, the required arguments, and the expectable return values. The functions correspond directly to the user's semantic understanding of the information. The implementation of the functions is hidden from the user.

The "links" of [Tsichritzis 75a] and the "selection structures" of [Earnest] also relate to the concept of specifying named relationships.

10.5.2 Implicit Relationships

There is a disadvantage to systems that deal only in named relationships. They limit the user to following paths that have been previously declared by a data administrator, and make it difficult to follow paths implicit in other data stored in the system.

As mentioned in section 4.6, if two entities are related to a third in any way, then that in itself constitutes a relationship among the first two. One employee might work in the same department as another. The secretary of a department probably serves as secretary for each employee in that department.

Attributes can provide such links in the same way as relationships. If an employee works at a certain location, this implies that his department has someone working at that location. If we have a mechanism for establishing that two attributes are "in the same domain", then we can infer a relationship between two entities having the same value of such attributes (cf. section 8.4.2). E.g., we could infer that a supplier and a warehouse are in the same city.

Both the domain and the role of the attributes must be considered, to avoid misunderstanding the significance of the implied relationship. If an employee was hired on the date his manager graduated college, we mustn't infer that they were hired on the same date, or born on the same date.

Other kinds of erroneous inferences might also be carelessly drawn. If a part is available from a certain supplier, and a warehouse is serviced by that supplier, we can't infer that the part is stocked in that warehouse. (And even if it was, it might be a different supplier who supplied that part to that warehouse.) This is the *connection trap* [Codd 70], whereby an erroneous inference may be drawn from the "join" of two relationships on a common domain. This is a user error, not the fault of the data model, in ascribing the wrong meaning to the results. The user error arises out of mistakenly taking such relational operators as "project" and "join" to be inverse operations, expecting that performing the two in succession returns the original information. A projection can decompose one relation into two; joining these two does not necessarily re-create the original relation.

One of the strengths of the relational model is that all such "implicitly defined" relationships are readily available, simply by joining relations on a common domain. It does require, however, that users correctly interpret the meaning of the joined relations.

There are some risks involved in the use of symbol matching to detect implicit relationships. Implicit (computed) relationships based on symbol matching are subject to the failures mentioned in section 3.9.2: synonyms prevent detection of relationships, ambiguities induce spurious connections.

Furthermore, if qualified names (multi-field keys) are used, there is potentially another kind of spurious connection. A match may be made with any other relation containing those two columns, even when those two columns are not serving as the qualified names of single entities. This was illustrated in section 8.8.3.

10.5.3 Orderings

An ordering has the appearance of a relationship (X is less than Y), but it would be cumbersome to model it as a binary relationship (pairing each item with every larger item).

Some orderings are obtained by sorting on data maintained about the entity (e.g., order employees alphabetically by name, or by employee number, or by salary, etc.). Other orderings, however, are not based on such data; they simply reflect a sequence based on some criterion that is not provided to the system in any "sorting" procedure. Examples of these include the lines in a text file, the statements in a program, the order of succession to an office (e.g., the presidency), a chronological sequence of measurements, the starting positions in a race, batting orders, and so on.

There are two ways to model such sequences, and it is not clear which is appropriate for a conceptual model. The first way is to specify a field or attribute whose value represents the ordinal position of an entity in this sequence or ranking. (Since the ranking may intermix several entity types, this attribute would have to be defined as common to all of them.) The management of this attribute can be presented to external users in several ways. The least desirable is to make the user fully responsible for its maintenance: on inserting, deleting, or moving an entity relative to the ranking, the user must update the sequence fields in all the subsequent entities in the ordered set. Alternatively, the system could understand the semantics of a sequence field: when the user places or moves one entity behind another, the system recomputes the sequence fields as needed. This facility is commonly provided by text processing systems.

The alternative is to represent order as an explicit relationship, e.g., "precedes". In the viewer's mind, the entities could be perceived as being physically adjacent, or connected by chains of pointers. However, the user could still be presented with facilities to either "insert X after Y" or "insert X as fifth" (perhaps implemented by a procedure that counts its way down a chain). An external representation could still contain a sequence field, perhaps generated by a counting process when the record is materialized.

The two approaches are functionally equivalent. Either can be made "primitive" in the conceptual model, with the other being derived or computed. Either can be presented to external users in the two forms "insert X after Y" and "insert X as fifth".

Either can be implemented internally by physical adjacency, sequence fields, pointer chains, indexes, or other techniques. Each implementation has its own performance tradeoffs, and perhaps different locking implications. If two users are each working with half of an ordered set, can one user insert or delete an entity without having to wait for sequence field updates to be propagated into the other half? Sometimes that interference is undesirable, in cases where only the relative order and not the actual sequence number is significant.

10.6 Existence Lists

This topic really concerns the modeling of entities, particularly with respect to establishing their existence. But we mention it here to contrast it with the most common practice: the existence of entities is typically not modeled independently, but is implied by their participation in various relationships.

For example, we often speak of employee records as though there was just one set of them, with exactly one record for each employee. But there is nothing in any of the record based models to preclude defining several sets of employee records, each containing different kinds of information. One might contain payroll information, another might have health information, and so on. Irreducible relations move us in that direction.

In a sense, we also have other kinds of records about employees, namely the intersection records that a normalized system forces us to maintain for many-to-many relationships (e.g., employment history). True, this is not a simple employee list, since there may generally be several such records per employee (e.g., one for each department in which he has worked). A given employee is likely to be involved in several such record types (one for each type of multi-valued fact about him, e.g., departments, children, skills, etc.), as well as several instances within each type.

Ironically, it is conceivable that those are the *only* kinds of records we have about him. If it happened that every single fact about an employee could be multi-valued (e.g., several names, several departments, several salaries, etc.), then there would be

no such thing as “an” employee record. All we could show for the employee is a collection of various types of intersection records.

Of all the kinds of records in which employees might occur, which type is to be considered the definitive list of employees? What is going to serve as the defining list for an existence test (section 2.4)?

Conceptually, at least, it would help to always have a notion of an existence list, whose purpose is to exhibit the currently known set of members of that type. Put another way, one ought to be able to assert the existence of something separately from providing assorted facts about it.

The relational model appears (depending on which papers you read) to have several partial approximations to existence lists. Each column of a relation may have, in addition to a column name, the name of some associated domain — but that domain is not a manipulatable structure in which one can add or delete members as with other relations. It is possible to specify constraints of the form “keys occurring in this relation must also occur in that relation” (e.g., [Smith 77a]) — but there is no discipline saying that you must nominate the same “that” relation in two constraints involving employees (hence there may still be no single relation defining the domain of employees). And, finally, defenders of the relational model point out that one could introduce unary relations (relations with only one column, e.g., employee number) to serve as domain sets, and consistently refer to such relations in constraints — one could, but the relational model doesn’t require it, and I don’t think anyone has ever done it.

[Furtado] notes: “An immediate consequence of adopting a graph-theoretical model is that, being assimilated to nodes, the domain elements exist by themselves. This is at variance with the original relational model.... where the existence of a domain element is conditional to its presence in some relation tuple”.

A final concern: domains are too often defined in terms of symbols (character strings) rather than entities. [McLeod], for example, considers a relational database to consist of a collection of normalized relations and a collection of domains, but defines

a domain to be "a set of atomic data values (objects). In particular, a domain is a subset of one of the two 'natural' domains: real number and character string." This is not at all an existence list. It is a syntax test for the acceptability of symbols; it is in no way a list of entities that permits individuals to be added or removed.

11 Elementary Concepts: Another Model?

Thus far we have been largely critical, and negative. We have identified problems without really suggesting solutions.

Can we identify an appropriate set of elementary concepts that will on the one hand serve as a general base for modeling information (in our limited use of that term), and on the other hand be an appropriate base for computerized implementations? Let us try.

What follows here is a sketch of work in progress, some basic ideas about the "right" set of constructs for such a model. Much work remains to be done — including an attempt to define more precisely the criteria by which the model is "right" in the first place.

I will begin (shortly) with some partially worked out ideas for a specific model, so that we know at the outset what conclusions I wish to justify. Then some motivations and comments will follow.

The model is not intended for modeling reality as such. It is rather an idealized system for processing information, which hopefully has some very useful characteristics for modeling reality. It is highly abstract, and can be implemented (realized) in real systems in many ways — just as the abstract concept of "ten" can be represented many ways in machines. Also, in its pure form, the model has certain properties that prevent it from ever being implemented perfectly — just as the infinite set of real numbers can never all be represented in a finite computer. For example, some things in the model are infinite, and some things exist without ever being created. Such things can only be approximated in real systems.

11.1 System Organization

As introduced in chapter 0, the model has to be understood as functioning in the context of a system organization, consisting of a repository, an interface, and a processor. Mostly, we will talk about the (apparent) contents of the repository. Very little will be said about the interface.

The processor is a large piece of unfinished business. If done properly, it can be the basis for canonical definitions of computer operations on data. It is the dynamic component of information definition, doing for data manipulation what the repository model does for static data structures.

It seems obvious that natural operations of the processor include creation, destruction, connection, and disconnection of objects. Executable objects (to be introduced below) will also have to be executed by the processor. Some aspects of the processor can be tailored — their precise definition is not intrinsic to the model. (But it will govern the kinds of things that may occur in the executables.) E.g.:

- Kinds and complexities of queries and expressions.
- Conventions followed for name resolution.

11.2 Primary Model Elements

11.2.1 Objects

The descriptions of most models begin by making distinctions, between such constructs as entities, relationships, attributes, names, types, collections, etc. These are implicitly taken to be mutually exclusive concepts, more or less.

We start instead from a unifying premise: all of these constructs are in fact entities. Each of these phenomena, and each of their instances, is a distinct integral concept, capable of being represented as a unit item in a model.

Everything in the repository is an “object”. The term is used interchangeably with “surrogate”, “representative”, and sometimes “thing”.

There are four kinds of objects: simple ones, and the three kinds described in subsequent sections — symbols, relationships, and executables. Simple objects don’t do anything else except represent entities; they occur very frequently.

There are very few general properties I can think of which apply to all objects. The main ones that come to mind now are these:

1. Objects can be related to each other.
2. Their existence can be detected by the processor.

The four kinds of objects are primitive to the processor in the sense that the processor can determine the kind of object directly, without chasing relationships to find out the “type”.

The four kinds are mutually exclusive, e.g., a relationship is not a symbol. But let me re-emphasize, because it’s important: these four kinds are all objects. Any kind of object can be related to any other kind of object.

11.2.2 Symbols

Some objects in the repository are “symbols”, for which I sometimes also use the term “string” (in the sense of a character string). Symbols are the only objects that can pass across the interface. The communication that takes place between you and the processor across the interface consists entirely of a stream of symbols. Some of those symbols come to rest in the repository; others are interpreted or generated by the processor. The symbols in the repository are related to other objects, serving as names, descriptions, or representations for those objects (more precisely, for the entities represented by those objects).

Remember, this is an abstract model. A real implementation doesn’t have to have this complex of two objects and a naming relationship for each and every entity; it should just behave as though it did.

A symbol object also needs to have a name itself, which can be passed through the interface. One often needs to reference it, as in the request to relate a certain entity to a certain name. Across the interface, a quotation mark convention can indicate that a symbol is naming itself. “Relate Harry to X” connects a thing named Harry to a thing named X. “Relate Harry to ‘X’“ connects a thing named Harry to the symbol X.

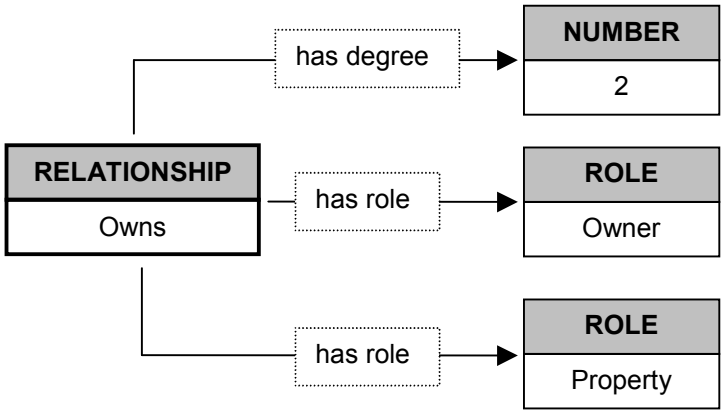
11.2.3 Relationships

Two aspects of relationships need to be modeled: the relation type, and the occurrences of the relationship.

I don’t have the type object well designed yet, but I hope it can be fabricated from ordinary objects. However, it might require the introduction of some additional primitive objects. The following appear to be the necessary characteristics of a relation type object:

- It is connected by naming relationships to the name(s) of the relationship.
- It is connected (by relationships?) to affiliated role objects, which represent the roles defined within the relationship. (Such “roles” are sometimes referred to as “selectors”.) An n-ary relationship has n affiliated role objects.
- The degree of the relationship is recorded in an ordinary way, e.g., a “has degree” relationship between the relationship type object and an ordinary quantity object.
- The role objects and/or the relationship type object are connected (by relationships?) to executable objects, which represent the domain constraints and other validation rules defined for the relationship.

A relationship type object may be drawn as:



This diagram uses conventions that will be defined in section 11.3.3: the type and name relationships are abbreviated by writing types and names inside the boxes. Also, constraints have been omitted from this diagram.

The unfinished state of these objects doesn't hurt the model. Their purpose and function is well understood. If they all have to be made primitive, nothing is really lost. It's just that there's a challenge in minimizing the number of primitives, and in defining the model elegantly.

Relationship occurrences, on the other hand, are well defined in this model. For brevity, I will sometimes use the term *link* for a relationship occurrence.

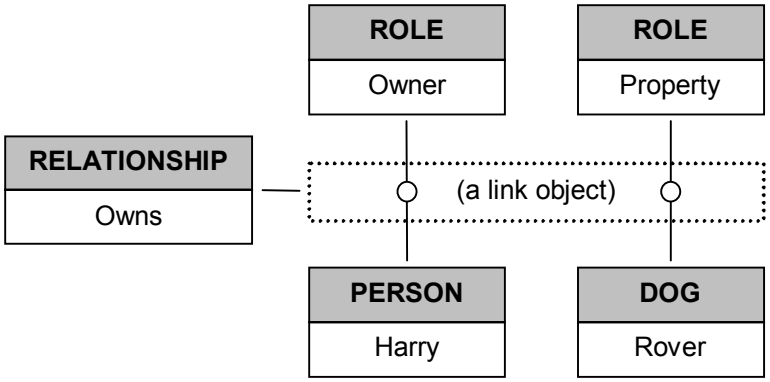
These link objects glue together all the objects in the repository, providing the basis of virtually all the information in the repository. The structure of a link is somewhat elaborate, serving the following functions:

- It has a connection to the relationship type of which this link is an occurrence.
- It has connections to the n objects being related, where n is the degree of the relationship.

- It connects each of these objects to a role object, to establish in which role each related object is occurring. (One could think of the roles and objects being ordered, with the correspondence being by position. But we don't make position numbers an explicit part of the model—so long as some correspondence mechanism exists.)

These connections are provided primitively by the link object. The connections are not themselves modeled as relationships, thus preventing an epidemic of infinite recursion.

A link object may be drawn as:



A link requires the continued existence of the objects it connects. It should be specifiable (in the creation of a relation type) what should happen on an attempt to delete an object connected to one of its links. Either the deletion is blocked, or the link goes too.

11.2.4 Executable Objects

This is another piece of largely unfinished business.

These are objects in the repository that can direct operations of the processor. They are introduced primarily to represent constraints to be enforced by the processor. They can also

represent implications or derivations — the generation of auxiliary objects or relationships that are consequences of other information. And they probably have other uses. Maybe they can be the embodiment of certain kinds of existence tests. And equality tests.

I don't really know how they will occur in the repository. To be pure, the object should be distinct from the "symbol" (long character string) which is its description in some language (its source text). In theory, at least, a variety of text strings could specify the same action.

There also has to be some mechanism for its connection into the web of information. It sometimes makes sense to link it to relationship type objects, since an executable is often triggered by the assertion of a relationship of a specified type. But sometimes it is triggered by a specified relationship to a particular object (a constraint on which things may be related to X by R). And there certainly has to be some mechanism that causes the processor to encounter the executable object at the right time, recognize it as such, and execute it.

On the other hand, executables might be used for triggered actions not related to specific relationship assertions, such as scheduled data changes based on time.

Also, something has to be designed concerning actions to take when constraints are violated. Maybe the general form of an executable is: "when (event) if (condition) then (action) else (action)"... or perhaps it should include a case statement.

And another part of the unknown: they will undoubtedly involve references — how linked? — to other objects involved in the constraints, e.g., type objects, or limit values.

11.3 Secondary Elements: A Vernacular

Those primary objects will be sufficient, I contend, for our modeling purposes (after we finish defining the objects, of course). But we can't escape the fact that such things as type, attribute, and set are also useful and common notions. I need them myself — I couldn't avoid using them throughout this book. And at the same time they remain ambiguous and

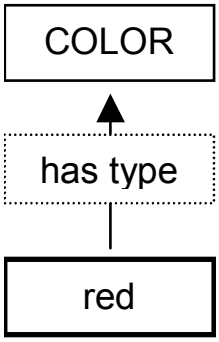
troublesome, being difficult to define precisely, or to distinguish from each other and from the primary concepts.

We cope with this by allowing two levels of thought, the rigorous and the vernacular. At the rigorous level we use only the primary concepts of the model. This is adequate for all purposes except comprehensibility: the sentences get awfully convoluted, and the diagrams are frightening.

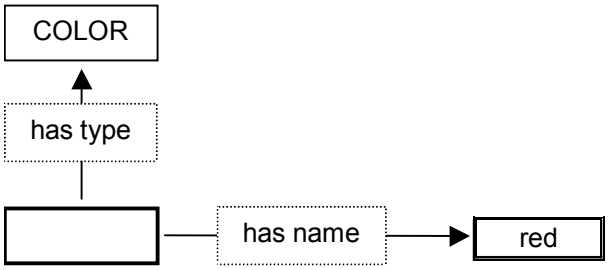
In the vernacular mode, we use secondary concepts that can be defined in terms of the primaries (they may not have been so defined, but they can be). Thus we may informally speak of "objects of type X", understanding that we mean objects related by a "has type" relation to an object named X, where X in turn is an object whose type is "type" (i.e., X is a type). And even that's not the full refinement: "an object named X" is vernacular for "object related by a 'has name' relation to symbol 'X'" .... and furthermore, the phrase "related by a 'has name' relation" refines to mean an instance of a relation that is itself related by the 'has name' relation to the symbol 'has name'.... and so on. The recursion can be stopped fairly soon — but it takes a frightening diagram to demonstrate it.

To illustrate, consider the phrase "red is a color", for which we have used the notation

We might say at a vernacular level that "red has-type color":

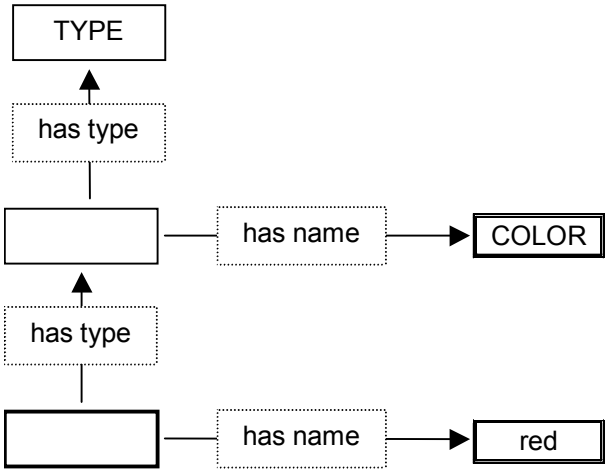


But this assumes concepts of types and names that aren't in the primary model. We might go through several stages of refinement to reduce this to purely primary concepts, starting with "an object whose name is 'red' and whose type is COLOR":

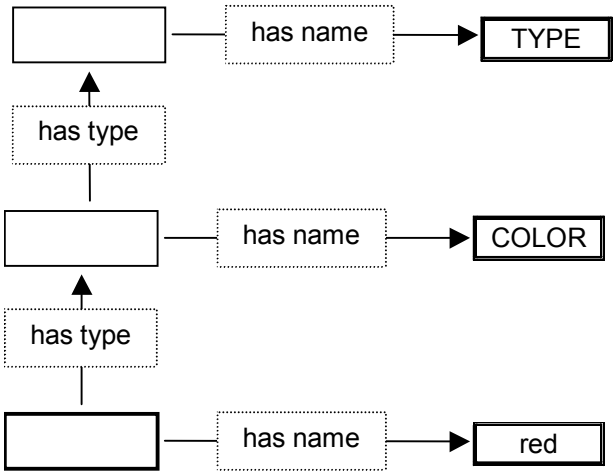


A box like red denotes a symbol.

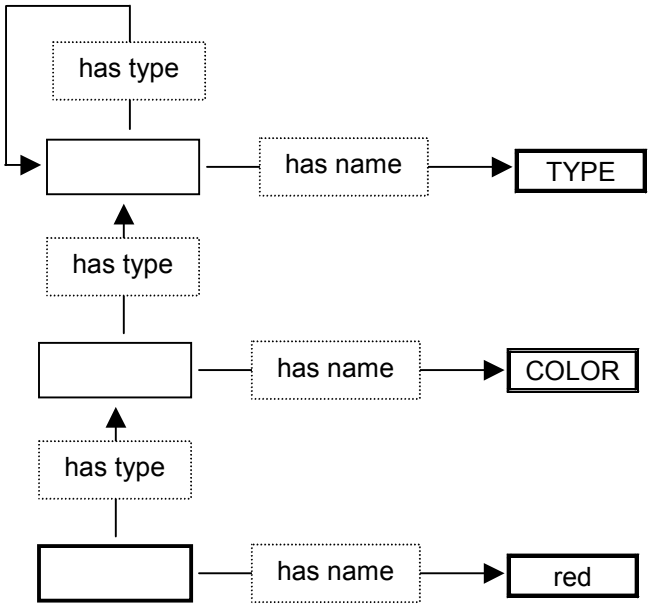
This in turn refines to "an object whose name is 'red' and whose type is an object whose name is 'COLOR' and whose type is TYPE":



And then to "an object whose name is 'red' and whose type is an object whose name is 'COLOR' and whose type is an object named 'TYPE'":



We stop here to escape infinite regress, since the type of TYPE is TYPE:



Several purposes are served by having both the rigorous and the vernacular levels. A broad range of important concepts is accounted for, without requiring their inclusion in the base model. The base model can be kept elegant, with the primitive concepts being few and well defined.

The base model provides a medium for very precise definition of the other concepts. Thus, when many of the dilemmas described in this book arise, they can often be resolved by referring to (or agreeing on) precise definitions in terms of primitives.

At the same time, by keeping them out of the base model, we don't need permanent or universal definitions, thus avoiding many tedious debates. We can agree on local definitions in the context of a particular conversation. And, in fact, different implementations of the model might employ different definitions of the secondary constructs.

Thus, we could view our product here as a "model generator", capable of producing various modeling systems

distinguished by their differing definitions of secondary constructs.

Also, we can avoid many of the difficulties concerning the distinction between concepts. They can appear distinct in the vernacular, while having similar underlying definitions. Thus, for example, type and attribute can appear to be distinct in the vernacular, but both be defined in terms of relationships. Consider the notion of being an employee, with the concept being represented by a single object. This object could be both a type and a status. An entity could have one relationship to this object, which could be interpreted by some as an attribute and by others as a type. (It may require a complex synonym for the relationship, with one person thinking it was named “has type” while the other thought it was named “has status”. On the other hand, we avoid that if the relation is simply named “has”.)

Thus two apparently distinct phenomena are given an interpretation as two views of the same phenomenon. And a mechanism is provided for treating them interchangeably: what is treated in one application as a type can be treated by another as an attribute.

I will not formally define any secondary elements.

11.3.1 Type

Here’s a way to introduce types, and to tailor them to suit your taste.

Let there be a distinguished object in the repository that your processor recognizes as representing the concept of “type”. (We could let this object be related to the symbol “TYPE” by a naming relationship.)

Let there be a relationship named “has-type”, one instance of which links TYPE with itself (i.e., TYPE is itself a TYPE; it is an object whose type is TYPE). Relate has-type to an executable, which will only permit X has-type Y if Y has-type TYPE (e.g., X can only have type EMPLOYEE if EMPLOYEE is a type).

The type Y can be introduced by asserting Y has-type TYPE. Things can now be of type Y by having a has-type relationship to Y.

The use of executables in this context needs to be worked out. For the constraints on objects of type Y, the executable would have to be triggered whenever a has-type relationship to Y is asserted. The executable would include constraints on naming conventions, and on allowable overlaps with other types.

If desired, a global constraint to the effect that no types may overlap could be attached to the TYPE object itself. ("For any X and Y and Z, X has-type Y and X has-type Z may only occur together if Y equals Z".) But I don't know how to effectively introduce the triggering of another global constraint that you might want: every object must have at least one type ("for each X there must be a Y such that X has-type Y"). Maybe it has to be triggered by the create operation of the processor.

11.3.2 Naming

Naming is really a very complex topic, with a large variety of structures and algorithms actually employed in real situations. The topic could include anything ranging from simple labels to complex (and perhaps interactive) stratagems for isolating a single object (name qualification, catalog cascades, intersecting or converging descriptions, and so on).

Our approach is to not prescribe any one simplified technique, but rather to provide an environment in which any desired structure and algorithm can be expressed. This is generally achieved by permitting any configuration of relationships among things and symbols, which may then be exploited by manipulative functions in any desired manner.

The simplest notion for naming is that there are two objects in the system, a nameless element that is the actual representative (surrogate) for something in the real world, and another object that is a symbol. A naming relationship connects these two, as in [Hall 76]. The surrogate may be so connected to several symbols, serving as synonyms or aliases, or even as descriptions. An implementation need not supply two such distinguishable objects; this device merely serves to describe the semantics of the model.

Such a naming convention, or any more complex ones desired, can be incorporated into the processor for a given system, making use of ordinary model objects (including executables).

The model does not require a thing to have any unique name. In fact, it does not require a thing to have any name at all. ("Create object owned by Harry, weighing 12 pounds." "What are the weights of things owned by Harry?" "12 pounds." "What does Harry own?" "One nameless thing weighing 12 pounds.") And there are babies. Many of them don't have names for a few hours, or even days, after birth. But data still gets recorded about them, and they certainly do get talked about.

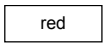
The requirement to have a name, or a unique name, can be imposed in various implementations, or when used in conjunction with particular data processing systems. But they are not intrinsic requirements of the model.

11.3.3 Vernacular Pictures

It is fairly natural to use abbreviated diagrams corresponding to these vernacular concepts. We have been using some already.

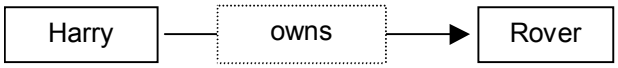
A picture of this form has a fairly natural interpretation. It depicts an object linked by a naming relationship to the symbol "red", and linked by a typing relationship to a type named "COLOR". That is, we have the type in the top of the box and the name in the bottom. Such a diagram is appropriate *if*:

- There is exactly one naming relationship that is understood to apply, and everyone knows which one (or nobody cares).
- The object has exactly one name.
- Ditto for types.

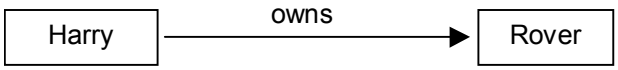


Sometimes even the type can be omitted, if we are willing to assume that everyone understands which one is implied.

Similar conventions hold for relationships. The link we showed in section 11.2.3 is often abbreviated as:



Or even as



Again, such conventions are only applicable when the full structure of the link is clearly inferable, e.g., the roles being played by each of the participants in the relationship.

11.3.4 Sets

Derived relationships (section 11.8.3) can be used to introduce a vernacular notion of sets. The derived relationships would be used for:

- General definitions for all sets re propagations of subset and membership relationships.
- For maintenance of membership relationship to a particular set, defined in terms of things having certain relationship to certain object.

11.4 The Name of the Model

Perhaps this model ought to have a name, for handy reference. Any model worth its salt ought to have a catchy acronym.

Sometimes I call it STAR, standing for "Strings (or Symbols), Things, And Relationships".

But it could also be ROSE: "Relationships, Objects, Symbols, and Executables".

Maybe I should have a contest. Winner gets to finish defining the model. (Losers get to use it?)

11.5 About Entities

11.5.1 Existence

We have to be careful about the sense in which we mean that an object (representative) "exists" in the repository. The existence tests have to be specified (using a vernacular based on type), and this conditions what we believe about existence.

The kind of existence test I understand best, and which I tend to assume most of the time, is a list-test with objects distinguished from symbols. That is, the "list" is a set of objects, each of which may have any number of symbols linked to it by any complex pattern of naming relationships. Existence is established when one of these symbol paths leads to an object in this set (and equality is established when two of these symbol paths leads to the same object).

Modeling something abstractly as having representatives does not necessarily imply that physical storage space will be occupied by such representatives. Nor does it imply that such representatives are explicitly created or destroyed.

The interaction of overlapping types must be considered. A given entity might be subject to multiple existence or equality tests. Are they consistent?

11.5.2 The Butler Did It

I haven't solved the problem of collapsing two entities into one (section 1.4.), and I don't know how. A brute force approach would be to discard one of the surrogates, and replace all references (linkages) to it by references to the other one. This could lead to enormous validation problems.

A mischievous idea: we could introduce a relationship meaning "is the same entity as". At the end of our mystery, we would simply assert such a relationship between the butler and the murderer. What in the world would that mean to our model?

Looking at the model "physically", we can plainly see that there are two entities connected by a relationship. But if we take cognizance of the semantics of the relationship, we must only perceive one entity (and no relationships?). Should a counting

function have to examine the semantics of such relationships? ("How many entities are in the room?" "How many relationships exist between the murderer and the butler?")

11.6 About Symbols

The initial concept is that of finite sequences of characters from some alphabet (including numbers and special characters). We don't really care which alphabet, so long as we can construct all the names, representations, descriptions, etc., that we need from it. Even with this looseness, we get into some funny considerations. Does the character size, case, font, color, etc. make a difference? How shall we decide when two symbols are the same or not? What about strings that run from right to left, or vertically? What about hieroglyphics, pictographs, ideograms, graphic images?

I give up. The essential idea is this. We postulate some interface in front of the model across which all communications with the model take place (typically involving some form of computer related input/output devices). We postulate the existence of some set of symbols that can pass across this interface, without really defining what we mean, except to say that:

- All communication across the interface is in terms of such symbols.
- There is some definite algorithm for determining when two symbols are the same or different (we don't care what the algorithm is).

11.7 The Symbol Stream and the Processor

It's becoming increasingly evident to me that we really need to think of the symbol stream that passes across the interface as being addressed to the processor. Symbols get into the repository indirectly, under control of the processor.

One could take a simple view to the effect that symbols go directly to the repository, where they serve to represent the thing

named by the symbol. One could say to the computer "Create John", causing the symbol "John" to enter the repository, to serve as the representative of the person named John. In subsequent communications, the symbol "John" will occur to indicate reference to that person. This won't do, in our model, because of our insistence on separating surrogates for entities from the symbols that name the entities.

But we can still imagine the computer having a straightforward interpretation of "Create John": it will first create the symbol "John" if it doesn't already exist, then create a new surrogate, and then link the new surrogate to the symbol "John" under a "name" relationship. This is still not too far removed from thinking that the symbol "John" goes directly into the repository, and that references to that person will contain the symbol "John" (or some other defined synonym).

But now consider this series of instructions: "Create a surrogate for something. It has a height of six feet. It is a person. Etc." In the second and third instructions, what is the symbol that refers to the subject of the information? It is the pronoun "it". To what extent is that a symbol, of the kind that one might find in the repository, connected to something by a naming relationship? Or is that pronoun really better described as an instruction to the processor, to be interpreted as another reference to the most recently referenced entity? (Which implies that the processor is maintaining a history of the dialog, to recall which was most recently referenced.)

More generally, one might refer to an entity by using a complex descriptive phrase causing the traversal of many entities and relationships, or the application of testing procedures, in order to arrive at the entity in question. Something like this occurs in many query systems, and is implicit in most name qualification conventions.

Thus, the concept of a symbol in a communication "representing" something becomes quite nebulous. It could be a simple symbol directly linked to the surrogate in question, or it could be an instruction guiding the processor to the surrogate. This spectrum of processing possibilities underlies our difficulty

in defining the difference between naming and description (chapter 3).

In any case, it is often held that there has to be some sort of explicit symbol in the communication that is associatable with the entity in question, even if the symbol is only a placeholder, like the pronouns “it” or “something”. But let me endow the processor with intelligence enough to play a familiar game (which assuredly can be programmed). I will say to this processor “Let’s play twenty questions.” That instruction is sufficient to cause the processor to create a new surrogate with absolutely no information attached, to which the processor will later expect to attach information. The information can be attached because the processor will understand the pronoun “it” to refer to that surrogate. Eventually the processor will try to match the surrogate and its attached information with some existing surrogate. But look at the initial instruction: “Let’s play twenty questions.” It caused a surrogate to be created. Where in that sentence is any sort of symbol that could be interpreted as a symbol (placeholder or otherwise) for that surrogate?

11.8 About Relationships

11.8.1 Entities

Once more, for emphasis: they are entities, and a relationship can link another relationship to something else.

Too many of the graphical models make entities and relationships mutually exclusive by forcing the entities to be points (nodes) and the relationships to be lines (edges). Then you’re not permitted to draw a line between two lines, or from a line to a point. What we have done, if you must picture it, is to give each line a bulge in its middle, so that it can itself function as a node.

11.8.2 Existence

Like entities, relationships can also have several modes of existence. Thus we have to be careful also about the sense in which we imagine links to exist. I really can't imagine an implementation in which every abstractly modeled link actually corresponds to, say, a pointer in storage. So, it may be necessary to introduce specifications of the existence modes of relationships.

Some of them may be functional (computed, procedural) — which could lead to the use of executables to model relationships. Orderings, trig functions, etc. seem to fit this mold. For such relationships, certain semantic characteristics ought to be made explicit, such as:

- Assertability, modifiability, deletability of occurrences (probably can't be done).
- Ability of occurrences to participate in other relationships (probably limited to other computed ones).
- Bidirectionality (inverse function might not be provided).
- Finiteness (listability, satisfiability of queries: list all occurrences of "less than").

Also, there is undoubtedly some interaction between the existence modes of a relationship and the objects it relates. It's hard to imagine an explicit link to an object whose existence is procedurally verified.

11.8.3 Derived (Implied) Relationships

The model needs a general mechanism for the specification and execution of derived relationships (cf. section 4.6). They are likely to take the form of executable objects, linked to:

- The relationship (type) whose derivation is being defined.
- The relationships and objects (types) from which it is derived.

11.8.4 Specification

The following things ought to be specifiable when a relationship type is created in the model, and a configuration is needed for representing these using model objects:

- Relationship name(s).
- Degree.
- Role names.
- Domain constraints.
- Other constraints.
- Existence mode, executable generators, etc.
- Cascading deletion rules.

11.8.5 Symmetric Relationships

Some variations might be appropriate to support symmetric relationships. The roles need not all be distinct, and hence a relationship type of degree n might have fewer than n affiliated role objects. The significance of n is that each link object must connect n objects; several of these may be connected to the same role object.

11.9 About Attributes

As promised in chapter 5, we don't distinguish these from relationships. If you want to introduce a definition, I suggest that the most promising route is as a special case of relationships, framed in terms of the existence tests for the types of entities involved.

11.10 Descriptions: Data About Data

Definitions, constraints, etc. are modeled right in the same repository, using model constructs.

Such definitions are often considered to be expressed in terms of “types”, e.g., X is the existence test for objects of type Y. We assume instead that such definitions are couched in a more general form, in terms of objects and relationships. E.g., X is the existence test for objects having relationship T to object Y.

If Y is interpreted as a type, then we can see that types and instances are necessarily modeled as being in the same repository, since they are connected by relationships (T, in this case).

11.11 Implementations

It is very important to keep in mind that the separation between things and symbols is a conceptual property of the model, without any direct implications for implementations. A naming relationship does not have to exist as a collection of distinct physical objects, each of which links a distinct thing object with another distinct symbol object. It can be implemented in any number of efficient and compact ways. The separation is only a device for explaining the semantic properties of the model.

Similarly, a type relationship does not have to be implemented with an enormous number of distinct relationship objects. Most implementations are likely to encode the type (or types) of a thing directly into its representative.

The same applies to other aspects of the model as well. It is not expected that a distinct physical object will exist for each and every entity. That would be very impractical for, e.g., numeric quantities. However, the model does provide the capability to express the corresponding semantic implications, e.g., whether or not entities of a given type need to be explicitly created before they are referenced.

To be perfect, use of the model should involve absolutely no assumptions about the existence of any kinds of entities, symbols, or relationships. They should all be asserted explicitly. But there is a large class of things we take for granted, and which are implicitly assumed to be provided in an expected realization in a computer. We take advantage of these being “built into” the

semantics of a computer; in any real use of the model, nobody wants to be bothered specifying these explicitly.

The things we get "for free" include:

- The existence of a set of symbols, and an alphabet from which they are constructed.
- The existence of the concept of numbers.
- Conventions establishing which symbols are naming which numbers under various conditions.
- The existence of ordering relationships, including collating sequences for non-numeric symbols.
- The existence of certain equality tests.
- The existence of certain synonym relationships: data type conversions, floating point normalizations, etc.
- The existence of certain acceptance tests: data types.

While we don't wish to be bothered specifying these, we do not always get what we had assumed.

If there is a limitation on the set of integers that can occur in the repository (limited by the capacity of the underlying computer), shouldn't that be expressed in the model?

It is possible to implement some of the primitive model objects simply as integers, i.e., internally generated globally unique identifiers. It works for some of the simple objects, depending on the nature of their existence tests. It doesn't work for links, since a simple integer doesn't capture the structure needed for connecting to the other objects involved. And it is likely to be clumsy for symbols, which in most cases should contain the actual string of characters comprising the symbol.

Even where internal integers are usable, they aren't necessary. Just about any large enough collection of discrete and linkable things will do. And even when such objects do have string-like internal representations (such as machine addresses or database keys), if they are not exposed across the interface they had might as well be arbitrary objects.

Executable objects are not necessarily implemented by discrete executable procedures. Structures in the underlying

system or machine may implicitly enforce certain constraints, in the way that hierarchical data structures enforce one-to-many relationships.

11.12 Comparison With Other Models

This is not really a new model, but a "better" packaging of existing ideas. Depending on which details you suppress, there are many models that are "just like" this one. The works to which I think the resemblance is especially strong include [Bracchi], [Hall 76], [Senko 76], [Falkenberg].

The following are some brief characterizations of the model. Which ones interest you depends on which other models you favor, or consider important to be compared with.

- The model is much like an irreducible n-ary model.
- The model is much like a binary relational model, in the sense of [Bracchi].
- It differs from any form of relational model in that a relationship occurrence is an aggregation of surrogates, not symbols.
- The model distinguishes between the objects (surrogates) that represent entities and the symbols that name entities.
- It supports many-to-many relationships directly.
- Its primary constructs are objects, relationships, symbols, and executable objects.
- It does not take type, attribute, set, or naming rules to be primary constructs.
- To the extent that it does have a type phenomenon, it allows types to overlap (an object can be of multiple types).
- The semantics to be specified for surrogates includes a description of the existence tests and the equality tests.
- Descriptions are not segregated from data. They reside in the same repository, and are interconnected.

- The model is described in the context of a system organization, consisting of a repository, an interface, and a processor.

12 Philosophy

12.1 Reality and Tools

I have tried to describe information as it "really is" (at least, as it appears to me), and have kept tripping over fuzzy and overlapping concepts. This is precisely why system designers and engineers and mechanics often lose patience with academic approaches. They recognize, often implicitly, that the complexity and amorphousness of reality is unmanageable. There is an important difference between truth and utility. We want things that are useful — at least in this business; otherwise we'd be philosophers and artists.

Perhaps it is inevitable that tools and theories never quite match. There are some opposite qualities inherent in them.

Theories tend to distinguish phenomena. A theory tends to be analytical, carefully identifying all the distinct elements and functions involved. Unifying explanations are abstracted, relationships and interactions are described, but the distinctness of the elements tends to be preserved.

Good tools, on the other hand, intermingle various phenomena. They get a job done (even better, they can do a variety of jobs). Their operation tends to intermix fragments of various theoretical phenomena; they embody a multitude of elementary functions simultaneously. That's what it usually takes to get a real job done. The end result is useful, and necessary, and profitable.

Theories tend toward completeness. A theory is defective if it does not account for all aspects of a phenomenon or function.

Tools tend to be incomplete in this respect. They incorporate those elements of a function that are useful and profitable; why bother with the rest? The justification for a tool is economic: the cost of its production and maintenance vs. the value of its problem solving functions. This has nothing to do with completeness. (In 1975, a government official asked to have his job abolished, because nobody actually needed the services of

his office. His job did have a well defined function, in theory. “Completeness” would have dictated that his job be retained.)

Useful tools have well defined parts, and predictable behavior. They lend themselves to solving problems we consider important, by any means we can contrive. We often solve a problem using a tool that wasn’t designed for it. Tools are available to be used, don’t cost too much, don’t work too slowly, don’t break too often, don’t need too much maintenance, don’t need too much training in their use, don’t become obsolete too fast or too often, are profitable to the toolmaker, and preferably come with some guarantee, from a reliable toolmaker. Tools don’t share many of the characteristics of theories. Completeness and generality only matter to the extent that a few tools can economically solve many of the problems we care about.

Thus the truth of things may be this: useful things get done by tools that are an amalgam of fragments of theories. Those are the kinds of tools whose production and maintenance expense can be justified. Theories are helpful to gain understanding, which may lead to the better design of better tools. This understanding is not essential; an un-analytic instinct for building good tools is just as useful, and often gets results faster.

It may be a mistake to require a tool to fit the mold of any theory. If this be so, then we’d better be aware of when we are discussing theory and when we are discussing tools.

Data models are tools. They do not contain in themselves the “true” structure of information. What really goes on when we present a data model, e.g., hierarchies, to a user? Does he say “Aha! Of course my information is hierarchically structured; I see how the model fits my data”? Of course not. He has to learn how to use it. We generally presume that this learning is required only because of the complexity of the tool. Difficulties are initially perceived as a failure to fully understand the theory; there is an expectation that perseverance will lead to a marvelous insight into how the theory fits the problem. In fact, much of his “learning” is really a struggle to contrive some way of fitting his problem to the tool: changing the way he thinks about his information, experimenting with different ways of representing it, and perhaps even abandoning some parts of his intended

application because the tool won't handle it. Much of this "learning" process is really a conditioning of his perceptions, so that he learns to accept as fact those assumptions needed to make the theory work, and to ignore or reject as trivial those cases where the theory fails.

Tools are generally orthogonal to the problems they solve, in that a given tool can be applied to a variety of problems, and a given problem can be solved in different ways with different tools. Versatility is in fact a very desirable property in a tool. It is useful then also to understand separately the characteristics of a tool and the nature of the problems to which it can be applied.

12.2 Points of View

A conceptual model, by its very nature, needs to be durable — at least in form, if not content. Its content should be adjusted to reflect changes in the enterprise and its information needs — only. The form of the conceptual model — the constructs and terms in which it is expressed — should be as impervious as possible to changes in the supporting computer technology. We can postulate that the man-machine interface will continue to evolve toward man; data processing technology will move toward handling information in ways that are natural to the people who use it. It follows then that a durable conceptual model should be based on constructs as close as possible to the human way of perceiving information.

There's a catch right there: the implicit assumption that there is just one "technology" by which all people perceive information, and hence which is most natural and easy for everybody to use. There probably isn't. Human brains undoubtedly function in a variety of ways. We know that some people do their thinking primarily in terms of visual images; others hear ideas being discussed in their heads; still others may have a different mode of intuiting concepts, neither visual nor aural. Analogously, some people may structure information in their heads in tabular form, others work best with analytic subdivisions leading to hierarchies, and others naturally follow paths in a network of relationships.

This may well be the root of the debates over which data model is best, most natural, easiest to learn and use, most machine independent, etc. The camps are probably divided up according to the way their brains function — each camp advocating the model that best approximates their own brain technology.

12.3 A View of Reality

"I do not know where we are going, but I do know this —that wherever it is, we shall lose our way." (Sagatsa)
"If you're confused, it just proves you've been paying attention." (G. Kent)

This book projects a philosophy that life and reality are at bottom amorphous, disordered, contradictory, inconsistent, non-rational, and non-objective. Science and much of western philosophy have in the past presented us with the illusion that things are otherwise. Rational views of the universe are idealized models that only approximate reality. The approximations are useful. The models are successful often enough in predicting the behavior of things that they provide a useful foundation for science and technology. But they are ultimately only approximations of reality, and non-unique at that.

This bothers many of us. We don't want to confront the unreality of reality. It frightens, like the shifting ground in an earthquake. We are abruptly left without reference points, without foundations, with nothing to stand on but our imaginations, our ethereal self-awareness.

So we shrug it off, shake it away as nonsense, philosophy, fantasy. What good is it? Maybe if we shut our eyes the notion will go away.

What do we know about physical entities, about ourselves?

Lewis Thomas tells us that a human being is not exactly a single discrete living thing, but more a symbiotic interaction of hordes of discrete living things inhabiting and motivating our cells. We are each an enormously divisible social structure [Thomas].

Sociobiologists are telling us that the human being is not the unit of evolution and survival. It is our genes that are motivated to survive and perpetuate themselves. Individual people are merely vehicles whose survival serves that higher purpose — sometimes! [Time Magazine, Aug. 1, 1977.]

Our precious self image is being challenged from another quarter, too. Some scientists aren't quite so sure any more that they can clearly distinguish between the categories of "man" and "animal". "People" might not be a well defined category! Recent experiments have demonstrated the capabilities of chimpanzees and gorillas to acquire language, concepts, symbols, abstractions—traits held by some to be the only significant hallmarks of the human species. A lawyer is prepared to argue that such animals are entitled to some of the protections accorded individuals under the law — such animals may be "legal persons". An article in the New York Times Magazine of June 12, 1977 observes: "If apes have access to language, can they not be expected to reason? And if they can reason, what distinction is there remaining between man and beast?" "Separately, and in some instances collectively, these animals have demonstrated the ability to converse with humans for as long as 30 minutes, to combine learned words in order to describe new situations or objects, to perceive difference and sameness, to understand 'if-then' concepts, to describe their moods, to lie, to select and use words in syntactic order, to express desire, to anticipate future events, to seek signed communication with others of their species and, in one extraordinary sequence .... to force the truth from a lying human." "... It's a heretical question, really. I was brought up a good Catholic. Man is man and beast is beast. I don't really think that now. You can't spend four or five years with a chimp, watch it grow up, and not realize that all the going on in her head is pretty much the same as that going on in mine ..."

Which brings to mind that our vision of ourselves as uniquely intelligent creatures is also threatened from quite another quarter — the one we've been dealing with all along here. What, in some people's view, is one of the objectives of artificial intelligence, if not to endow machines with an

intelligence competitive with humans? Is science fiction really mistaken in its visions of humanoids and robots functioning like, or better than, human beings? How often have those visionaries been wrong before?

In the monthly magazine published by the American Museum of Natural History, we read: "Some futurists ... view the current difference between human and artificial intelligence as one of degree, not of kind, and predict that the gap between humans and machines will be crossed about the year 2000" [Jastrow]. Data processing people are fond of saying that the category of employees is a subset of the category of people. How long before we have to expand that to include animals and robots? I wonder if that question will really sound as foolish to someone reading this, say, twenty or fifty years from now.

What does all this do to our sense of identity, to our egocentric view of people as entities? If we have to rebuild our world view so radically again (as, for example, Copernicus forced us to do once before), then how much faith can we have in the permanence of any world view?

Our notions of reality are overwhelmingly dominated by the accidental configurations of our physical senses. We are very parochial in our sense of scale. Bacteria and viruses and sub-atomic particles are not very real to most of us, nor are galaxies. We don't really know how to comprehend them. Our concept of motion is bounded by the physiology of our eyes: the continental plates don't move, but motion pictures (sequences of still pictures!) do. Most of us think of continents and islands as permanent and discrete entities — rather than as accidents of the current water level in the oceans. Are islands and mountains such different things? Have you ever had the opportunity to observe a reservoir get filled, or emptied?

And our sense of reality is quite conditioned by the very narrow frequency range to which our eyes respond. Imagine if we couldn't see the "visible" spectrum, but could see ultra-violet, or infra-red, or x-rays — or maybe sound waves! We might not have any notion of opaque objects; everything might be translucent or transparent. Things might appear to have entirely different shapes or boundaries. We might not have such a

primary notion of things having sharp or fixed boundaries; the normal mode of things might be a state of flux, like the wind or clouds or currents in the ocean. Think of perceiving people in terms of the thermal gradients around their bodies, rather than gradients in the visible spectrum. We might have no concept of day or night. Those concepts are only so "real" and "fundamental" because we are so dependent on visible light. Clumps of heat might look like "things" to us, just as clouds do now. We might see sounds as physical things moving through the air, and we might see the wind.

Or suppose that senses other than sight dominated our world view. The universe of many animals — their sense of what things exist, and what they are — is based on smells. To them, the existence and nature of a thing is defined primarily by what it smells like. What it looks like is an occasional, trivial consideration (like the smell of things is to us). In a heavy fog, we suddenly live in a universe of things heard, rather than things seen.

The shark seems to have sense organs responding directly to electrical phenomena. What image of reality could it have, which we don't even know how to imagine? (And what view of reality do we have, which a blind person doesn't even know how to imagine? Can you even begin to imagine how it feels to have no comprehension at all of what the verb "see" means?)

To a greater or lesser extent, we all operate with somewhat different foundations for our perceptions of reality. Biologist Robert Trivers comments: "The conventional view that natural selection favors nervous systems that produce ever more accurate images of the world must be a very naive view of mental evolution." [Time Magazine, Aug. 1, 1977.] Among many of us, the differences are trivial. Between some of us they are enormous.

Compare your view of reality with that of a mathematical physicist, or an astronomer. (If you are one, how does it feel to be singled out as having a peculiar view?) The world view of such people includes as regular features such notions as Einsteinian time and space, particles of light, light being bent by gravity, everything accelerating away from everything else, black

holes, and seeing things (stars) that may have vanished thousands or millions of years ago. How often do these crop up in your world view?

Your brain may be obliged to confess such views are real, but your intuition isn't. What shall we make of it? The earth does look flat, after all, doesn't it? And, no matter how much schooling we've had, we can't seem to stop thinking of the sun as rising and setting. Incidentally, do your children share your world view of this phenomenon?

"Consider how the world appears to any man, however wise and experienced in human life, who has never heard one word of what science has discovered about the Cosmos. To him the earth is flat; the sun and moon are shining objects of small size that pop up daily above an eastern rim, move through the upper air, and sink below a western edge; obviously they spend the night somewhere underground. The sky is an inverted bowl made of some blue material. The stars, tiny and rather near objects, seem as if they might be alive, for they 'come out' from the sky at evening like rabbits or rattlesnakes from their burrows, and slip back again at dawn. 'Solar system' has no meaning to him, and the concept of a 'law of gravitation' is quite unintelligible — nay, even nonsensical. For him bodies do not fall because of a law of gravitation, but rather 'because there is nothing to hold them up' — i.e., because he cannot imagine their doing anything else. He cannot conceive space without an 'up' and 'down' or even without an 'east' and 'west' in it. For him the blood does not circulate; nor does the heart pump blood; he thinks it is a place where love, kindness, and thoughts are kept. Cooling is not a removal of heat but an addition of 'cold'; leaves are not green from the chemical substance chlorophyll in them, but from the 'greenness' in them. It will be impossible to reason him out of these beliefs. He will assert them as plain, hard-headed common sense; which means that they satisfy him because they are completely adequate as a system of communication between him and his fellow men. That is, they are adequate linguistically to his social needs, and will remain so until an additional group of needs is felt and is worked out in language" [Whorf].

So far I've dealt with variations in perceived reality that I can at least describe. They are close enough to my world view (and yours, I hope) that I can describe the differences in terms of familiar concepts. But I must acknowledge the existence of world views so alien to mine that I can't even grasp the central concepts. These are exemplified by some of the Eastern philosophies, various theologies, mystical cults. The Hopi Indians have a world view of time and causality that can hardly even be expressed in our vocabulary of concepts. "I find it gratuitous to assume that a Hopi who knows only the Hopi language and the cultural ideas of his own society has the same notions, often supposed to be intuitions, of time and space that we have, and that are generally assumed to be universal. In particular, he has no general notion or intuition of time as a smooth flowing continuum in which everything in the universe proceeds at an equal rate, out of a future, through a present, into a past." "The Hopi language and culture conceals a metaphysics, such as our so-called naive view of space and time does, or as the relativity theory does; yet it is a different metaphysics from either. In order to describe the structure of the universe according to the Hopi, it is necessary to attempt — insofar as it is possible — to make explicit this metaphysics, properly describable only in the Hopi language, by means of an approximation expressed in our own language, somewhat inadequately it is true ...." [Whorf].

Do you and I have the "real" notion of time? What shall we make of contemporary physics, which wants us to believe that time passes at different rates for objects traveling at different speeds? The astronaut who has been traveling a year close to the speed of light has been gone from us for ten years? Or is it vice versa?

Language has an enormous influence on our perception of reality. Not only does it affect how and what we think about, but also how we perceive things in the first place. Rather than serving merely as a passive vehicle for containing our thoughts, language has an active influence on the shape of our thoughts. "...language produces an organization of experience... language

first of all is a classification and arrangement of the stream of sensory experience that results in a certain world order..."

[Whorf].

Whorf quoting Edward Sapir: "Human beings do not live in the objective world alone, nor alone in the world of social activity as ordinarily understood, but are very much at the mercy of the particular language that has become the medium of expression for their society. It is quite an illusion to imagine that one adjusts to reality without the use of language and that language is merely an incidental means of solving specific problems of communication or reflection. The fact of the matter is that the 'real world' is to a large extent unconsciously built up on the language habits of the group.... We see and hear and otherwise experience very largely as we do because the language habits of our community predispose certain choices of interpretation."

"Hopi has one noun that covers every thing or being that flies, with the exception of birds, which class is denoted by another noun.... The Hopi actually call insect, airplane, and aviator all by the same word, and feel no difficulty about it.... This class seems to us too large and inclusive, but so would our class 'snow' to an Eskimo. We have the same word for falling snow, snow on the ground, snow packed hard like ice, slushy snow, wind-driven flying snow — whatever the situation may be. To an Eskimo, this all-inclusive word would be almost unthinkable; he would say that falling snow, slushy snow, and so on, are sensuously and operationally different, different things to contend with; he uses different words for them and for other kinds of snow. The Aztecs go even farther than we in the opposite direction, with 'cold', 'ice', and 'snow' all represented by the same basic word with different terminations; 'ice' is the noun form; 'cold', the adjectival form; and for 'snow', 'ice mist'."

We are more ready to perceive things as entities when our language happens to have nouns for them. For what reason does our language happen to have the noun "schedule" for the

connection between, say, a train and a time, but no such familiar noun for the connection between a person and his salary?

The way we bundle relationships is similarly affected. If we think of the relationships “has color” and “has weight”, we might be inclined to lump them into a single “has” relationship, with several kinds of entities in the second domain. But if we happen to employ the word “weighs”, then that makes it easier to think of the second relationship as being distinct in its own right. By what accident of linguistic evolution do we fail to have a similar verb for the color phenomenon? (“Appears” might be a close approximation.)

Other examples: “has salary” vs. “earns”, “has height” vs. what?

The accidents of vocabulary: we are most prepared to identify as entities or relationships those things for which our vocabulary happens to contain a word. The presence of such a word focuses our thinking onto what then appears as a singular phenomenon. The absence of such a word renders the thought diffuse, non-specific, non-singular.

This is all very unsatisfying. It is consistent with this philosophy of reality (perhaps even necessary, rather than just consistent), that I cannot see it applied consistently. I must accept paradoxes embedded right in the process of embracing such views. I am not, after all, such an alien creature. I see the world in much the same terms as you do. I have a name, and an employer, and a social security number, and a salary, and a birth date, etc. etc. There is a reasonably accurate description of me and my environment in several files. I have a wife, and children, and a car, all of which I believe to be very real. In short, I can share with you a very traditional view of reality; most of the useful activities of my daily life are predicated on such familiar foundations.

Well then, what’s going on? What are these contradictions all about?

I’m really not sure, but perhaps I can try to frame an answer in terms of purpose and scope. I am convinced, at bottom, that no two people have a perception of reality that is identical in every detail. In fact, a given person has different views at

different times — either trivially, because detailed facts change, or in a larger sense, such as the duality of my own views.

But there is considerable overlap in all of these views. Given the right set of people, the differences in their views may become negligible. Reducing the number of people involved greatly enhances this likelihood. This is what I mean by "scope": the number of people whose views have to be reconciled.

In addition, there is a question of purpose. Views can be reconciled with different degrees of success to serve different purposes. By reconciliation I mean a state in which the parties involved have negligible differences in that portion of their world views that is relevant to the purpose at hand. If an involved party holds multiple viewpoints, he may agree to use a particular one to serve the purpose at hand. Or he may be persuaded to modify his view, to serve that purpose.

If the purpose is to arrive at an absolute definition of truth and beauty, the chances of reconciliation are nil. But for the purposes of survival and the conduct of our daily lives (relatively narrow purposes), chances of reconciliation are necessarily high. I can buy food from the grocer, and ask a policeman to chase a burglar, without sharing these people's views of truth and beauty. It is an inevitable outcome of natural selection that those of us who have survived share, within a sufficiently localized community, a common view of certain basic staples of life. This is fundamental to any kind of social interaction.

If the purpose is to maintain the inventory records for a warehouse, the chances of reconciliation are again high. (How high? High enough to make the system workably acceptable to certain decision makers in management.) If the purpose is to consistently maintain the personnel, production, planning, sales, and customer data for a multi-national corporation, the chances of reconciliation are somewhat less: the purposes are broader, and there are more people's views involved.

So, at bottom, we come to this duality. In an absolute sense, there is no singular objective reality. But we can share a common enough view of it for most of our working purposes, so that reality does appear to be objective and stable.

But the chances of achieving such a shared view become poorer when we try to encompass broader purposes, and to involve more people. This is precisely why the question is becoming more relevant today: the thrust of technology is to foster interaction among greater numbers of people, and to integrate processes into monoliths serving wider and wider purposes. It is in this environment that discrepancies in fundamental assumptions will become increasingly exposed.

Bibliography

[Abrial] J.R. Abrial, "Data Semantics", in [Klimbie].

[ANSI 75] ANSI/X3/SPARC, Study Group on Database Management Systems, Interim Report, Feb. 1975.

[ANSI 77] *The ANSI/X3/SPARC DBMS Framework, Report of the Study Group on Database Management Systems,* (D. Tsichritzis and A. Klug, editors), AFIPS Press, 1977.

[Armstrong] W.W. Armstrong, "Dependency Structures of Database Relationships", in J.L. Rosenfeld (ed.), *Information Processing* 74, North Holland, 1974.

[Ash] W.L. Ash and E.H. Sibley, "TRAMP: An Interpretive Associative Processor With Deductive Capabilities", Proc. 1968 ACM Nat. Conf., 144-156.

[Astrahan 75] M.M. Astrahan and D.D. Chamberlin, "Implementation of a Structured English Query Language", Comm. ACM 18 (10), Oct. 1975.

[Astrahan 76] M.M. Astrahan et al., "System R: Relational Approach to Database Management", ACM Transactions on Database Systems 1 (2), June 1976, pp. 97-137.

[Bachman 75] C.W. Bachman, "Trends in Database Management - 1975", National Computer Conference, 1975.

[Bachman 77] C.W. Bachman and M. Daya, "The Role Concept in Data Models", in [VLDB 77].

[Bell] A. Bell and M.R. Quillian, "Capturing Concepts in a Semantic Net", Proc. Symp. on Associative Information Techniques, Sept. 30-Oct. 1, 1968, Warren, Mich.

[Berild] S. Berild and S. Nachmens, "Some Practical Applications of CS4 — A DBMS for Associative Databases", in [Nijssen 77].

[Bernstein 75] P.A. Bernstein, J.R. Swenson, and D.C. Tsichritzis, "A Unified Approach to Functional Dependencies and Relations", in [SIGMOD 75].

[Bernstein 76] P.A. Bernstein, "Synthesizing Third Normal Form Relations From Functional Dependencies", ACM Transactions on Database Systems 1 (4), Dec. 1976.

[Biller 76] H. Biller and E.J. Neuhold, "Semantics of Databases: The Semantics of Data Models", Technical Report 03/76, Institut fur Informatik, University of Stuttgart, Germany.

[Biller 77] H. Biller and E.J. Neuhold, "Concepts for the Conceptual Schema", in [Nijssen 77].

[Bobrow] D.G. Bobrow and A. Collins (ed.), *Representation and Understanding,* Academic Press, 1975.

[Boyce] R.F. Boyce and D.D. Chamberlin, "Using a Structured English Query Language as a Data Definition Facility", IBM Research Report RJ1318, Dec. 1973.

[Bracchi] G. Bracchi, P. Paolini and G. Pelagatti, "Binary Logical Associations in Data Modelling", in [Nijssen 76].

[C&A 70] "The Empty Column", Computers and Automation, Jan. 1970.

[Chamberlin 74] D.D. Chamberlin and R.F. Boyce, "SEQUEL: A Structured English Query Language", in [SIGMOD 74].

[Chamberlin 76a] D.D. Chamberlin, "Relational Database Management Systems", ACM Computing Surveys 8 (1), March 1976, pp. 43-66.

[Chamberlin 76b] D.D. Chamberlin et al., "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control", IBM Journal of Research and Development 20 (6), Nov. 1976, pp. 560-575.

[Chen] P.P.S. Chen, "The Entity-Relationship Model: Toward a Unified View of Data", ACM Transactions on Database Systems 1 (1), March 1976, pp. 9-36.

[Childs] D.L. Childs, "Extended Set Theory", in [VLDB 77].

[CODASYL 71] CODASYL Database Task Group Report, ACM, New York, April 1971.

[CODASYL 73] CODASYL DDL, Journal of Development, June 1973 (Supt. of Docs., U.S. Govt. Printing Office, Washington D.C., catalog no. C13.6/2:113).

[Codd 70] E.F. Codd, "A Relational Model of Data for Large Shared Data Banks", Comm. ACM 13 (6), June 1970.

[Codd 71a] E.F. Codd, "A Database Sublanguage Founded on the Relational Calculus", in [SIGFIDET 71].

[Codd 71b] E.F. Codd, "Normalized Database Structure: A Brief Tutorial", in [SIGFIDET 71].

[Codd 72] E.F. Codd, "Further Normalization of the Database Relational Model", in R. Rustin (ed.), *Database Systems (Courant Computer Science Symposia 6)*, Prentice-Hall, 1972.

[Codd 74] E.F. Codd and C.J. Date, "Interactive Support for Non-Programmers: The Relational and Network Approaches", in [SIGMOD 74-2].

[Date 74] C.J. Date and E.F. Codd, "The Relational and Network Approaches: Comparison of the Application Programming Interface", in [SIGMOD 74-2].

[Date 77] C.J. Date, *An Introduction to Database Systems (second edition)*, Addison-Wesley, 1977.

[Davies] C.T. Davies, "A Logical Concept for the Control and Management of Data", Report AR-0803-00, IBM, 1967.

[DBTG] Same as [CODASYL].

[Delobel] C. Delobel and R.G. Casey, "Decomposition of a Database and the Theory of Boolean Switching Functions", IBM Journal of Research and Development, 17 (5), Sept. 1973, pp. 374-386.

[Douque] B.C.M. Douque and G.M. Nijssen (eds.), *Database Description,* North Holland, 1975. (Proc. IFIP TC-2 Special Working Conf., Wepion, Belgium, Jan. 13-17, 1975.)

[Durchholz] R. Durchholz, "Types and Related Concepts", in [ICS 77].

[Earnest] C. Earnest, "Selection and Higher Level Structures in Networks", in [Douque].

[Engles 70] R.W. Engles, "A Tutorial on Database Organization", Annual Review in Automatic Programming, 7 (1), Pergamon Press, Oxford, 1972, pp. 1-64.

[Engles 71] R.W. Engles, "An Analysis of the April 1971 DBTG Report", in [SIGFIDET 71].

[Eswaran] K.P. Eswaran and D.D. Chamberlin, "Functional Specifications of a Subsystem for Database Integrity", in [VLDB 75], pp. 48-68.

[Fabun] Don Fabun, "Communications: The Transfer of Meaning", Glencoe Press, 1968.

[Fadous] R. Fadous and J. Forsyth, "Finding Candidate Keys for Relational Databases", in [SIGMOD 75].

[Fagin] R. Fagin, “Multivalued Dependencies and a New Normal Form for Relational Databases”, ACM Transactions on Database Systems 2 (3), Sept. 1977.

[Falkenberg 76a] E. Falkenberg, “Concepts for Modelling Information”, in [Nijssen 76].

[Falkenberg 76b] E. Falkenberg, “Significations: The Key To Unify Database Management”, Information Systems 2 (1), 1976, pp. 19-28.

[Falkenberg 77] E. Falkenberg, “Concepts for the Coexistence Approach to Database Management”, in [ICS 77].

[Folinus] J.J. Folinus, S.E. Madnick, and H.B. Schutzman, “Virtual Information in Database Systems”, FDT (SIGFIDET Bulletin) 6(2) 1974.

[Furtado] A.L. Furtado, “Formal Aspects of the Relational Model”, Monographs in Computer Science and Computer Applications, No. 6/76, Catholic University, Rio de Janeiro, Brazil, April 1976.

[Goguen] J.A. Goguen, “On Fuzzy Robot Planning”, in [Zadeh].

[Griffith 73] R.L. Griffith and V.G. Hargan, “Theory of Idea Structures”, IBM Technical Report TR02.559, April 1973.

[Griffith 75] R.L. Griffith, “Information Structures”, IBM Technical Report TR03.013, May 1976.

[GUIDE-SHARE] “Database Management System Requirements”, Joint Guide-Share Database Requirements Group, Nov. 1970.

[Hall 75] P.A.V. Hall, P. Hitchcock, and S.J.P. Todd, “An Algebra of Relations for Machine Computation”, Second ACM Symposium on Principles of Programming Languages, Palo Alto, California, Jan. 1975.

[Hall 76] P.A.V. Hall, J. Owlett and S.J.P. Todd, “Relations and Entities”, in [Nijssen 76].

[Hammer] M.M. Hammer and D.J. McLeod, “Semantic Integrity in a Relational Database System”, in [VLDB 75].

[Hayakawa] S.I. Hayakawa, *Language in Thought and Action,* third edition, Harcourt Brace Jovanovich, 1972.

[Heidorn] G.E. Heidorn, “Natural Language Inputs to a Simulation Programming System”, Report NPS-55HD72101A, Naval Postgraduate School, Monterey, 1972.

[ICS 77] *International Computing Symposium 1977,* North Holland, 1975, E. Morlet and D. Ribbens (eds.). (Proc. ICS77, Liege, Belgium, April 4-7, 1977.)

[IMS] IMS/VS General Information Manual, IBM Form No. GH20-1260.

[Jardine] D.A. Jardine, *The ANSI/SPARC DBMS Model,* North Holland, 1977. (Proc. SHARE Working Conference on DBMS, Montreal, Canada, Apr. 26-30, 1976.)

[Jastrow] Robert Jastrow, "Post-Human Intelligence", Natural History 86(6), June-July 1977, pp. 12-18.

[Kent 73] W. Kent, "A Primer of Normal Forms", Technical Report TR02.600, IBM, San Jose, California, Dec. 1973.

[Kent 76] W. Kent, "New Criteria for the Conceptual Model", in [Lockemann].

[Kent 77a] W. Kent, "Entities and Relationships in Information", in [Nijssen 77].

[Kent 77b] W. Kent, "Limitations of Record Oriented Information Models", IBM Technical Report TR03.028, May 1977.

[Kerschberg 76a] L. Kerschberg, A. Klug, and D. Tsichritzis, "A Taxonomy of Data Models", in [Lockemann].

[Kerschberg 76b] L. Kerschberg, E.A. Ozkarahan, and J.E.S. Pacheco, "A Synthetic English Query Language for a Relational Associative Processor", Proc. 2nd Intl. Conf. on Software Engineering, San Francisco, 1976.

[Klimbie] J.W. Klimbie and K.L. Koffeman (eds.), *Database Management,* North Holland, 1974. (Proc. IFIP Working Conf. on Database Management, Cargese, Corsica, France, April 1-5, 1974.)

[Levien] R.E. Levien and M.E. Maron, "A Computer System for Inference Execution and Data Retrieval", Comm. ACM 1967, 10, 715-721.

[Lockemann] P.C. Lockemann and E.J. Neuhold (eds.), *Systems for Large Databases,* North Holland, 1977. (Proc. Second International Conference on Very Large Databases, Sept. 8-10, 1976, Brussels.)

[Martin] J. Martin, *Computer Data-Base Organization,* Prentice-Hall, 1975.

[McLeod] D.J. McLeod, "High Level Domain Definition in a Relational Database System", Proceedings of Conference on Data: Abstraction, Definition, and Structure, (Salt Lake City, Utah, March 22-24, 1976), ACM 1976.

[Mealy] G.H. Mealy, "Another Look at Data", Proc. AFIPS 1967 Fall Joint Computer Conf., Vol. 31.

[Meltzer 75] H.S. Meltzer, "An Overview of the Administration of Databases", Second USA-Japan Computer Conference, Tokyo, Aug. 28, 1975, pp. 365-370.

[Metaxides] A. Metaxides, discussion on p. 181 of [Douque].

[Mumford] E. Mumford and H. Sackman (eds.), *Human Choice and Computers,* North Holland, 1975.

[Nijssen 75] G.M. Nijssen, "Two Major Flaws in the CODASYL DDL 1973 and Proposed Corrections", Information Systems, Vol. 1, 1975, pp. 115-132.

[Nijssen 76] G.M. Nijssen, *Modelling in Database Management Systems,* North Holland, 1976. (Proc. IFIP TC-2 Working Conf., Freudenstadt, W. Germany, Jan. 5-9, 1976.)

[Nijssen 77] G.M. Nijssen, *Architecture and Models in Database Management Systems,* North Holland, 1977. (Proc. IFIP TC-2 Working Conf., Nice, France, Jan. 3-7, 1977.)

[Pirotte] A. Pirotte, "The Entity-Association Model: An Information-Oriented Database Model", in [ICS 77].

[Rissanen 73] J. Rissanen and C. Delobel, "Decomposition of Files, a Basis For Data Storage and Retrieval", IBM Research Report RJ1220, May 1973.

[Rissanen 77] J. Rissanen, "Independent Components of Relations", ACM Transactions on Database Systems 2 (4), Dec. 1977.

[Robinson] K.A. Robinson, "Database — The Ideas Behind the Ideas", Computer Journal 18 (1), Feb. 1975, pp. 7-11.

[Roussopoulos] N. Roussopoulos and J. Mylopoulus, "Using Semantic Networks for Database Management", in [VLDB 75], pp. 144-172.

[Sapir] E. Sapir, "Conceptual Categories in Primitive Languages", Science (74), 1931, p. 578.

[Schank] R.C. Schank and K.M. Colby, *Computer Models of Thought and Language,* W.H. Freeman, 1973.

[Schmid 75] H.A. Schmid and J.R. Swenson, “On the Semantics of the Relational Model”, in [SIGMOD 75], pp. 211-223.
[Schmid 77] H.A. Schmid, “An Analysis of Some Constructs for Conceptual Models”, in [Nijssen 77].
[Senko 73] M.E. Senko, E.B. Altman, M.M. Astrahan, and P.L. Fehder, “Data Structures and Accessing in Database Systems”, IBM Systems J. 1973, 12, 30-93.
[Senko 75a] M.E. Senko, “The DDL in the Context of a Multilevel Structured Description: DIAM II with FORAL”, in [Douque], 239-257.
[Senko 75b] M.E. Senko, “Information Systems: Records, Relations, Sets, Entities, and Things”, Information Systems 1 (1), 1975, pp. 1-13.
[Senko 76] M.E. Senko, “DIAM as a Detailed Example of the ANSI SPARC Architecture”, in [Nijssen 76].
[Senko 77a] M.E. Senko, “Data structures and data accessing in database systems past, present, future”, IBM Systems Journal 16 (3), 1977, pp. 208-257.
[Senko 77b] M.E. Senko, “Conceptual schemas, abstract data structures, enterprise descriptions”, in [ICS 77].
[Shapiro] S.C. Shapiro, “The Mind System. A Data Structure for Semantic Information Processing”, Rand Corp., Santa Monica, California, Aug. 1971.
[Sharman 75] G.C.H. Sharman, “A New Model of Relational Database and High Level Languages”, Technical Report TR.12.136, IBM United Kingdom, Feb. 1975.
[Sharman 77] G.C.H. Sharman, “Update-by-Dialogue: An Interactive Approach to Database Modification”, in [SIGMOD 77].
[Sibley] E.H. Sibley and L. Kerschberg, “Data Architecture and Data Model Considerations”, National Computer Conference, 1977.
[SIGFIDET 71] ACM SIGFIDET Workshop on Data Description, Access, and Control, Nov. 11-12, 1971, San Diego, California, E.F. Codd & A.L. Dean (eds.).
[SIGMOD 74] ACM SIGMOD Workshop on Data Description, Access, and Control, May 1-3, 1974, Ann Arbor, Mich., R. Rustin (ed.).

[SIGMOD 74-2] Volume 2 of [SIGMOD 74]: "Data Models: Data Structure Set Versus Relational".

[SIGMOD 75] ACM SIGMOD International Conference on Management of Data, May 14-16, 1975, San Jose, California, W.F. King (ed.).

[SIGMOD 77] ACM SIGMOD International Conference on Management of Data, Aug. 3-5, 1977, Toronto, Canada, D.C.P. Smith (ed.).

[Smith 77a] J.M. Smith and D.C.P. Smith, "Database Abstractions: Aggregation", Comm. ACM 20 (6), June 1977.

[Smith 77b] J.M. Smith and D.C.P. Smith, "Database Abstractions: Aggregation and Generalization", ACM Transactions on Database Systems 2 (2), June 1977.

[Smith 77c] J.M. Smith and D.C.P. Smith, "Integrated Specifications for Abstract Systems", UUCS-77-112, University of Utah, Sept. 1977.

[Sowa 76] J.F. Sowa, "Conceptual Graphs for a Database Interface", IBM J. Res. & Dev. 20 (4), July 1976.

[Sowa] J.F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine,* Addison-Wesley, forthcoming.

[Stamper 73] R. Stamper, *Information in Business and Administrative Systems,* John Wiley, 1973.

[Stamper 75] R. Stamper, "Information Science for Systems Analysis", in [Mumford].

[Stamper 77] R.K. Stamper, "Physical Objects, Human Discourse, and Formal Systems", in [Nijssen 77].

[Sundgren 74] Bo Sundgren, "Conceptual Foundation of the Infological Approach to Databases", in [Klimbie].

[Sundgren 75] Bo Sundgren, *Theory of Databases,* Petrocelli, N.Y., 1975.

[Taylor] R.W. Taylor and R.L. Frank, "CODASYL Database Management Systems", ACM Computing Surveys 8 (1), March 1976, pp. 67-104.

[Thomas] Lewis Thomas, *The Lives of a Cell,* Viking Press, N.Y., 1974.

[Titman] P.J. Titman, "An Experimental Database System Using Binary Relations", in [Klimbie].

[Tsichritzis 75a] D. Tsichritzis, "A Network Framework for Relation Implementation", in [Douque].

[Tsichritzis 75b] D. Tsichritzis, "Features of a Conceptual Schema", CSRG Technical Report No. 56, University of Toronto, July 1975.

[Tsichritzis 76] D. Tsichritzis and F.H. Lochovsky, "Hierarchical Database Management Systems", ACM Computing Surveys 8 (1), March 1976, pp. 105-124.

[Tsichritzis 77] D.C. Tsichritzis and F.H Lochovsky, *Database Management Systems,* Academic Press, 1977.

[Tully] C.J. Tully, "The Unsolved Problem — A New Look At Computer Science", Computer Bulletin 2 (2), Dec. 1974.

[VLDB 75] Proceedings of the International Conference on Very Large Databases, Sept. 22-24, 1975, Framingham, Mass. (ACM, New York).

[VLDB 76] (Same as [Lockemann]).

[VLDB 77] Proceedings of the Third International Conference on Very Large Databases, Oct. 6-8, 1977, Tokyo, Japan. Database 9 (2), Fall 1977; SIGMOD Record 9 (4), Oct. 1977.

[Weber] H. Weber, "D-Graphs: A Conceptual Model for Databases", in [ICS 77].

[Whorf] Benjamin Lee Whorf, *Language, Thought, and Reality,* MIT, 1956.

[Zadeh] L.A. Zadeh, K. Fu, K. Tanaka, and M. Shimura (eds.), *Fuzzy Sets And Their Applications to Cognitive and Decision Processes,* Academic Press, 1975.

[Zemanek 72] H. Zemanek, "Some Philosophical Aspects of Information Processing", in *The Skyline of Information Processing,* North Holland, 1972 (H. Zemanek, ed.).

[Zemanek 75] H. Zemanek, "The Human Being and the Automaton", in [Mumford].

Detailed Contents

ABOUT BILL KENT

Bill Kent likes to write about information processing as well as a variety of non-technical topics. His published technical papers, both tutorial and advanced, cover the relational data model, data analysis and design, entity-relationship models, object technology, and other areas of information processing.

Bill's career in data processing spans thirty-seven years at IBM and Hewlett-Packard. At HP's Research Laboratory in Palo Alto, California, he helped develop a prototype object-oriented database system and a follow-on prototype supporting interoperability of heterogeneous database systems.

Bill was a founder and first chairman of the NCITS (National Committee for Information Technology Standards) Committee on Object Information Management (formerly ANSI X3H7). He served as acting chair of the ANSI/SPARC DBSSG Object-Oriented Database Task Group. Other activities include ANSI X3H2 (Database) and IFIP TC2.6 (Data Bases), as well as the Object Management Group (OMG), serving on their Technical Committee, Object Model Subcommittee, and Database Special Interest Group.

Bill enjoys writing, photography, canyon country, Tai Chi, drumming, skiing, rafting, and a few other things. You can discover more at his web site: http://www.bkent.net .

A brief sampler of Bill's publications:

"A Simple Guide to Five Normal Forms in Relational Database Theory", Communications of the ACM, Feb. 1983.

"The Many Forms of a Single Fact", Proc. IEEE COMPCON, Feb. 27-Mar. 3, 1989, San Francisco.

"The Leading Edge of Database Technology", in E.D. Falkenberg, P. Lindgreen (eds), *Information System Concepts: An In-depth Analysis,* North Holland, 1989. Also in F.H. Lochovsky (ed), *Entity-Relationship Approach to Database Design and Querying,* Elsevier Science Publishers (North Holland), 1990.

"The Breakdown of the Information Model in Multi-Database Systems", SIGMOD Record, Dec 1991.

"A Rigorous Model of Object Reference, Identity, and Existence", Journal of Object-Oriented Programming, June 1991,

"The Objects Are Coming!", Computer Standards and Interfaces, July 1993.

Richard Soley and William Kent, "The OMG Object Model", in *Modern Database Systems: The Object Model, Interoperability, and Beyond",* Won Kim (editor), ACM Press/Addison-Wesley, 1995.