# SOFTWARE AND MIND

Andrei Sorin

EXTRACT

Chapter 7: *Software Engineering*
Section *The Relational Database Model*

**This extract includes the book's front matter and part of chapter 7.**



This section analyzes the relational database model and its mechanistic fallacies, and shows that it is a pseudoscience.

The entire book, each chapter separately, and also selected sections, can be viewed and downloaded free at the book's website.

**www.softwareandmind.com**

# SOFTWARE AND MIND

## The Mechanistic Myth and Its Consequences

Andrei Sorin

ANDSOR BOOKS

Published by Andsor Books, Toronto, Canada (www.andsorbooks.com)
First edition 2013. Revised 2019.



For disclaimers see pp. vii, xvi.

Designed and typeset by the author with text management software developed by the author and with Adobe FrameMaker 6.0. Printed and bound in the United States of America.

## Acknowledgements

Excerpts from the works of Karl Popper: reprinted by permission of the University of Klagenfurt/Karl Popper Library.

Excerpts from *The Origins of Totalitarian Democracy* by J. L. Talmon: published by Secker & Warburg, reprinted by permission of The Random House Group Ltd.

Excerpts from *Nineteen Eighty-Four* by George Orwell: Copyright ©1949 George Orwell, reprinted by permission of Bill Hamilton as the Literary Executor of the Estate of the Late Sonia Brownell Orwell and Secker & Warburg Ltd.; Copyright ©1949 Harcourt, Inc. and renewed 1977 by Sonia Brownell Orwell, reprinted by permission of Houghton Mifflin Harcourt Publishing Company.

Excerpts from *The Collected Essays, Journalism and Letters of George Orwell*: Copyright ©1968 Sonia Brownell Orwell, reprinted by permission of Bill Hamilton as the Literary Executor of the Estate of the Late Sonia Brownell Orwell and Secker & Warburg Ltd.; Copyright ©1968 Sonia Brownell Orwell and renewed 1996 by Mark Hamilton, reprinted by permission of Houghton Mifflin Harcourt Publishing Company.

Excerpts from *Doublespeak* by William Lutz: Copyright ©1989 William Lutz, reprinted by permission of the author in care of the Jean V. Naggar Literary Agency.

Excerpts from *Four Essays on Liberty* by Isaiah Berlin: Copyright ©1969 Isaiah Berlin, reprinted by permission of Curtis Brown Group Ltd., London, on behalf of the Estate of Isaiah Berlin.

## Library and Archives Canada Cataloguing in Publication

Sorin, Andrei
Software and mind : the mechanistic myth and its consequences / Andrei Sorin.

Includes index.
ISBN 978-0-9869389-0-0

1. Computers and civilization. 2. Computer software – Social aspects.
3. Computer software – Philosophy. I. Title.

QA76.9.C66S67 2013 303.48'34 C2012-906666-4

Don't you see that the whole aim of Newspeak is to narrow the range of thought?… Has it ever occurred to you … that by the year 2050, at the very latest, not a single human being will be alive who could understand such a conversation as we are having now?

George Orwell, *Nineteen Eighty-Four*

# Disclaimer

This book attacks the mechanistic myth, not persons. Myths, however, manifest themselves through the acts of persons, so it is impossible to discuss the mechanistic myth without also referring to the persons affected by it. Thus, all references to individuals, groups of individuals, corporations, institutions, or other organizations are intended solely as examples of mechanistic beliefs, ideas, claims, or practices. To repeat, they do not constitute an attack on those individuals or organizations, but on the mechanistic myth.

Except where supported with citations, the discussions in this book reflect the author's personal views, and the author does not claim or suggest that anyone else holds these views.

The arguments advanced in this book are founded, ultimately, on the principles of demarcation between science and pseudoscience developed by philosopher Karl Popper (as explained in "Popper's Principles of Demarcation" in chapter 3). In particular, the author maintains that theories which attempt to explain non-mechanistic phenomena mechanistically are pseudoscientific. Consequently, terms like "ignorance," "incompetence," "dishonesty," "fraud," "corruption," "charlatanism," and "irresponsibility," in reference to individuals, groups of individuals, corporations, institutions, or other organizations, are used in a precise, technical sense; namely, to indicate beliefs, ideas, claims, or practices that are mechanistic though applied to non-mechanistic phenomena, and hence pseudoscientific according to Popper's principles of demarcation. In other words, these derogatory terms are used solely in order to contrast our world to a hypothetical, ideal world, where the mechanistic myth and the pseudoscientific notions it engenders would not exist. The meaning of these terms, therefore, must not be confused with their informal meaning in general discourse, nor with their formal meaning in various moral, professional, or legal definitions. Moreover, the use of these terms expresses strictly the personal opinion of the author – an opinion based, as already stated, on the principles of demarcation.

This book aims to expose the corruptive effect of the mechanistic myth. This myth, especially as manifested through our software-related pursuits, is the greatest danger we are facing today. Thus, no criticism can be too strong. However, since we are all affected by it, a criticism of the myth may cast a negative light on many individuals and organizations who are practising it unwittingly. To them, the author wishes to apologize in advance.

# Contents

# Preface

This revised version (currently available only in digital format) incorporates many small changes made in the six years since the book was published. It is also an opportunity to expand on an issue that was mentioned only briefly in the original preface.

*Software and Mind* is, in effect, several books in one, and its size reflects this. Most chapters could form the basis of individual volumes. Their topics, however, are closely related and cannot be properly explained if separated. They support each other and contribute together to the book's main argument.

For example, the use of simple and complex structures to model mechanistic and non-mechanistic phenomena is explained in chapter 1; Popper's principles of demarcation between science and pseudoscience are explained in chapter 3; and these notions are used together throughout the book to show how the attempts to represent non-mechanistic phenomena mechanistically end up as worthless, pseudoscientific theories. Similarly, the non-mechanistic capabilities of the mind are explained in chapter 2; the non-mechanistic nature of software is explained in chapter 4; and these notions are used in chapter 7 to show that software engineering is a futile attempt to replace human programming expertise with mechanistic theories.

A second reason for the book's size is the detailed analysis of the various topics. This is necessary because most topics are new: they involve either

entirely new concepts, or the interpretation of concepts in ways that contradict the accepted views. Thorough and rigorous arguments are essential if the reader is to appreciate the significance of these concepts. Moreover, the book addresses a broad audience, people with different backgrounds and interests; so a safe assumption is that each reader needs detailed explanations in at least some areas.

There is some deliberate repetitiveness in the book, which adds only a little to its size but may be objectionable to some readers. For each important concept introduced somewhere in the book, there are summaries later, in various discussions where that concept is applied. This helps to make the individual chapters, and even the individual sections, reasonably independent: while the book is intended to be read from the beginning, a reader can select almost any portion and still follow the discussion. In addition, the summaries are tailored for each occasion, and this further explains that concept, by presenting it from different perspectives.

The book's subtitle, *The Mechanistic Myth and Its Consequences*, captures its essence. This phrase is deliberately ambiguous: if read in conjunction with the title, it can be interpreted in two ways. In one interpretation, the mechanistic myth is the universal mechanistic belief of the last three centuries, and the consequences are today's software fallacies. In the second interpretation, the mechanistic myth is specifically today's mechanistic *software* myth, and the consequences are the fallacies *it* engenders. Thus, the first interpretation says that the past delusions have caused the current software delusions; and the second one says that the current software delusions are causing further delusions. Taken together, the two interpretations say that the mechanistic myth, with its current manifestation in the software myth, is fostering a process of continuous intellectual degradation – despite the great advances it made possible.

The book's epigraph, about Newspeak, will become clear when we discuss the similarity of language and software (see, for example, pp. 409–411).

Throughout the book, the software-related arguments are also supported with ideas from other disciplines – from the philosophies of science, of mind, and of language, in particular. These discussions are important, because they show that our software-related problems are similar, ultimately, to problems that have been studied for a long time in other domains. And the fact that the software theorists are ignoring this accumulated knowledge demonstrates their incompetence.

Chapter 7, on software engineering, is not just for programmers. Many parts

(the first three sections, and some of the subsections in each theory) discuss the software fallacies in general, and should be read by everyone. But even the more detailed discussions require no previous programming knowledge. The whole chapter, in fact, is not so much about programming as about the delusions that pervade our programming practices, and their long history. So this chapter can be seen as a special introduction to software and programming; namely, comparing their true nature with the pseudoscientific notions promoted by the software elite. This study can help both programmers and laymen to understand why the incompetence that characterizes this profession is an inevitable consequence of the mechanistic software ideology.

The book is divided into chapters, the chapters into sections, and some sections into subsections. These parts have titles, so I will refer to them here as *titled* parts. Since not all sections have subsections, the lowest-level titled part in a given place may be either a section or a subsection. This part is, usually, further divided into *numbered* parts. The table of contents shows the titled parts. The running heads show the current titled parts: on the right page the lowest-level part, on the left page the higher-level one (or the same as the right page if there is no higher level). Since there are more than two hundred numbered parts, it was impractical to include them in the table of contents. Also, contriving a short title for each one would have been more misleading than informative. Instead, the first sentence or two in a numbered part serve also as a hint of its subject, and hence as title.

Figures are numbered within chapters, but footnotes are numbered within the lowest-level titled parts. The reference in a footnote is shown in full only the first time it is mentioned within such a part. If mentioned more than once, in the subsequent footnotes it is abbreviated. For these abbreviations, then, the full reference can be found by searching the previous footnotes no further back than the beginning of the current titled part.

The statement "italics added" in a footnote indicates that the emphasis is only in the quotation. Nothing is stated in the footnote when the italics are present in the original text.

In an Internet reference, only the site's main page is shown, even when the quoted text is from a secondary page. When undated, the quotations reflect the content of these pages in 2010 or later.

When referring to certain individuals (software theorists, for instance), the term "expert" is often used mockingly. This term, though, is also used in its normal sense, to denote the possession of true expertise. The context makes it clear which sense is meant.

The term "elite" is used to describe a body of companies, organizations, and individuals (for example, the software elite). The plural, "elites," is used when referring to several entities within such a body.

The issues discussed in this book concern all humanity. Thus, terms like "we" and "our society" (used when discussing such topics as programming incompetence, corruption of the elites, and drift toward totalitarianism) do not refer to a particular nation, but to the whole world.

Some discussions in this book may be interpreted as professional advice on programming and software use. While the ideas advanced in these discussions derive from many years of practice and from extensive research, and represent in the author's view the best way to program and use computers, readers must remember that they assume all responsibility if deciding to follow these ideas. In particular, to apply these ideas they may need the kind of knowledge that, in our mechanistic culture, few programmers and software users possess. Therefore, the author and the publisher disclaim any liability for risks or losses, personal, financial, or other, incurred directly or indirectly in connection with, or as a consequence of, applying the ideas discussed in this book.

The pronouns "he," "his," "him," and "himself," when referring to a gender-neutral word, are used in this book in their universal, gender-neutral sense. (Example: "If an individual restricts himself to mechanistic knowledge, his performance cannot advance past the level of a novice.") This usage, then, aims solely to simplify the language. Since their antecedent is gender-neutral ("everyone," "person," "programmer," "scientist," "manager," etc.), the neutral sense of the pronouns is established grammatically, and there is no need for awkward phrases like "he or she." Such phrases are used in this book only when the neutrality or the universality needs to be emphasized.

It is impossible, in a book discussing many new and perhaps difficult concepts, to anticipate all the problems that readers may face when studying these concepts. So the issues that require further discussion will be addressed online, at *www.softwareandmind.com*. In addition, I plan to publish there material that could not be included in the book, as well as new ideas that may emerge in the future. Finally, in order to complement the arguments about traditional programming found in the book, I have published, in source form, some of the software I developed over the years. The website, then, must be seen as an extension to the book: any idea, claim, or explanation that must be clarified or enhanced will be discussed there.

# The Relational Database Model

The relational database model is the theoretical concept upon which the relational database systems are founded. Database systems are environments for data management, and among them the relational kind are the most popular. In this section we will try to determine how much of this popularity is due to their data management capabilities, and how much to our mechanistic delusions. What we will find is that, like the other mechanistic software theories, the relational database model is a pseudoscience; that it is worthless as a programming concept; and that the relational systems became practical only after *annulling* their relational features, and after *reinstating* – in a more complicated form, and under new names – the *traditional* data management principles.

The relational model belongs to the class of theories that promise us higher levels of abstraction than those offered by the traditional programming

languages. Based on these theories, elaborate development systems are created and promoted. But instead of being abandoned when the idea of higher levels proves to be a fantasy, these systems undergo a series of "improvements." And the improvements, it turns out, consist in the addition of *low-level* capabilities; that is, precisely those features we had in the traditional languages, and which these systems were meant to supersede. So, in the end, all we accomplish is to replace efficient and straightforward languages with slow, complicated, and expensive development systems. (See "The Delusion of High Levels" in chapter 6; see also "The Quest for Higher Levels" in the previous section.)

## The Promise

### 1

The idea of a database management system emerged in the late 1960s, when it was noticed that programmers had difficulty designing correct file relationships. Individually, the file operations are quite simple: reading a particular data record, writing a new record or modifying an existing one, and the like. The difficulty, rather, lies in creating correct *combinations* of operations. Applications need many files, and a file may have many records. Moreover, through the data present in their fields, the records form intricate relationships, and the file operations must exactly match these relationships if the application is to run correctly. It is the programmer's task to specify the iterations and conditions through which the application will create and use the various records at run time; and even a small error can have such consequences as reading or deleting the wrong record, corrupting the data stored in a record, or slowing down the application by performing unnecessary file operations.

The challenges that programmers face with file operations, thus, are similar to those they face with any other aspect of the application. So, as is the case with the other challenges, what they need is expertise: the knowledge and skills one develops over the years by programming increasingly complex applications. Programmers have difficulty designing correct and efficient file operations because they lack this expertise, because our programming culture prevents them from advancing past the level of novices.

The theories of structured programming and object-oriented programming, we saw, were invented by the software elite in an effort to obviate the need for programming expertise. Since programmers had difficulty creating correct flow-control constructs, restricting them to a few, standard constructs was seen as the answer; and since they had difficulty creating useful and modifiable applications, restricting them to ready-made modules was seen as the answer.

The solution to programming incompetence, in other words, was always thought to lie, not in encouraging programmers to *improve* their skills, but in discovering methods that would eliminate the *need* for skills; specifically, methods that would permit inexperienced programmers to accomplish tasks demanding expertise.

And it was the same ideology that prompted the invention of the relational database model. If programmers have difficulty designing combinations of file operations, let us provide these combinations in the form of built-in, high-level operations. For example, simply by specifying a few parameters, programmers would be able to read a set of logically related records. In contrast, to read the same records with the basic file operations, programmers must scan the file one record at a time, and control this process using iterative and conditional constructs.

❖

Historically, the first database systems (which were based on the so-called hierarchical and network database models) were seen largely as management tools: means to take away from programmers the responsibility of designing and maintaining the application's database. The software experts hoped that, by keeping the most important database functions outside the application, database systems would eliminate our dependence on the skills of programmers: the database would become the responsibility of managers or analysts, and the programmers would simply be told, for each requirement, what database operations to invoke.

Now, the general trend was already to break down the application into smaller and smaller parts, in order to match the capabilities of inexperienced programmers. The trend, in other words, was to prevent programmers from *designing* software, and to reduce their work to little more than translating into a programming language the instructions received from a superior. So what the first database systems were promising was to reduce database programming to the same type of work. For all practical purposes, programmers would no longer need to know anything about the files used by the application, or about the relations between files. All they would have to do is translate some simple instructions into the equivalent database operations.

As was the case with the relational model later, attempting to simplify programming by raising the level of abstraction only made it more complicated. New software concepts, design methodologies, and languages (known as data definition and data manipulation languages) had to be introduced to support the hierarchical and network models; and in the end, the high-level operations turned out to be more difficult than the traditional ones. Database

systems, thus, were a fraud from the start: they complicated programming instead of simplifying it, and did not provide any functions that could not be implemented through the traditional file operations. (In fact, certain file relationships, easily programmed using file operations, cannot be implemented at all with the hierarchical and network models.)

When judged from within our corrupt software culture, however, the appeal of the original promise is understandable. Once we accept the idea that the highest programming skills we can expect are those attained by an average person after a few months of practice, replacing programmers with software devices seems logical. The complexity created by database systems is a small price to pay, the software experts tell us, for what we gain: successful database management regardless of the skills of the available personnel. Surely, we cannot trust a *programmer* with the task of designing the complex relationships that make up a database. Besides, with so many programmers working on the same application, it is impractical to allow each one to modify the file definitions. A specialist should design the database, and the best way to separate this task from the programming tasks is with a database management system.

In reality, the programmer is the best person to design the application's database, just as he is the best person to deal with every other aspect of the application. However, this is true only of experienced, professional programmers. Everyone could see that the existing programmers were novices, not professionals. But instead of giving them the time and opportunity to develop their skills, the preferred solution was to employ hordes of these novices, and to create several levels of management – bureaucrats with titles like systems analyst and project manager – to supervise them.[1]

As I have already remarked, this ideology – the belief that programming skills can be replaced with management skills – was already accepted when the first database systems were being introduced. So the idea of transferring the responsibility for the application's *database* from programmers to managers, although just as absurd as the attempt to replace the other programming skills,

[1] Most business applications can be developed and supported by *one* person. Thus, working alone and only part-time, I designed, programmed, and maintained several business systems over the years – the kind of systems for which companies normally employ teams of programmers and analysts. Few people are aware of the immense inefficiency created when a number of inexperienced individuals work together on one software project. The resulting application can become, quite literally, hundreds of times larger than necessary, and also far more involved. The combination of large teams, incompetence, and the dependence on development environments and ready-made pieces of software gives rise to an inefficiency that feeds on itself. In the end, these bureaucrats spend most of their time solving specious problems, which they themselves keep creating, instead of genuine software and business problems.

appeared quite logical. (As it turned out, though, the complexity of the database systems exceeded the capacity of existing management, and a new type of software bureaucrat had to be invented – the database administrator.)

And so it is how, from a totally unnecessary tool, database systems became one of the main preoccupations, and a major contribution to the astronomic cost of data processing, in most computer installations.

## 2

Although all database models suffer from the same fallacies, it is the relational model that concerns us, because it was beginning with this model that the software experts presented the concept of database systems as a *scientific* theory. If the earlier systems were promoted as management tools, the relational systems were also seen as a step in the formalization of application development. Because the relational model is based on certain mathematical concepts, the experts were now convinced that the benefits of database systems had been proved. Accordingly, a manager who refused to adopt this model was guilty of more than just resisting software progress; he was guilty of rejecting science.

The relational model is much more ambitious than the earlier ones. What we are promised is that, if we keep the data in a particular format, and if we restrict ourselves to a particular type of operations, we will never again have to deal with *low-level* entities (indexes, individual records and fields, and the related iterative and conditional constructs). In addition, the relational model will eliminate all data inconsistencies. The files are treated now simply as tables with rows and columns, and all we have to do is select and combine logical portions of these tables. Any database requirement, we are told, can be implemented in this fashion – in the same way that any mathematical expression is, ultimately, a combination of some basic operations.

As they did for the other mechanistic ideas, the software experts failed to understand why the mathematical background of a theory does not guarantee its usefulness for application development. Its exact nature only means that a mechanistic model has been found for *one* aspect of the application. And this is a trivial accomplishment: We know that complex phenomena can be represented as systems of simple hierarchical structures; and we also know that it is possible to extract any one of these structures and to represent it with a mechanistic model. Software applications comprise many structures, so it is not difficult to find exact models if all we want is to represent these structures *individually*.

Thus, the theory of structured programming asked us to extract that

aspect of the application that is its static flow diagram. Since it is possible to reduce this one aspect to a perfect hierarchy, and hence to represent it mathematically, the experts believed that the application as a whole can be represented mathematically. The theory of object-oriented programming asked us to identify the various aspects of our affairs, and to depict each one with a hierarchical classification of software entities. The experts believed that if each aspect is represented with a perfect hierarchy, whole applications can be developed simply by combining these hierarchies. Finally, the relational database theory asks us to extract that aspect of the application that is its database, and to reduce to perfect hierarchies the file relationships. This will permit us to represent mathematically the database structures, and hence the database operations.

What a mechanistic software theory does, in the final analysis, is model structures of a particular type, after separating them from the other structures that make up the application. So, no matter how successful these theories are in representing the individual structures, they are worthless as *programming* theories, because we cannot develop the application by dealing with each structure separately. The elements of these structures are the software entities that make up the application. Thus, since they share their elements, the structures interact, and we must deal with all of them at the same time. No theory that represents individual structures can model the whole application closely enough to be useful as a programming theory.

Like all mechanistic delusions, these theories are very naive. Since we already know that simple hierarchical structures can be represented mathematically, what these experts perceive as an important discovery is in reality a predictable achievement. All they are doing is breaking down software phenomena into smaller and smaller aspects, until they reach aspects that are simple enough to model with a hierarchical structure; and at that point they discover a mathematical theory for one of those aspects. But this is not surprising. The mathematical nature of the theory is a quality possessed by every hierarchical structure. We knew all along that they would find mathematical theories for the individual aspects. The very reason they separated the original, complex phenomenon into simpler phenomena is that simple structures can be represented with mathematical models while complex ones cannot.

Practitioners, though, must deal with whole applications, not isolated software phenomena. So, for these theories to have any value, we also need an exact theory for *combining* the various aspects – those neat hierarchical structures – into actual applications. And no such theory exists. The structures that represent the various aspects of an application are not related mechanistically, as one within another. They form complex structures.

The only way to create an application, therefore, is by relying on the non-mechanistic capabilities of our mind. But then, if we must have the expertise to deal with complex structures, why do we need theories that break down applications into simple structures in the first place? The software theorists are naive, thus, because they underestimate the difficulty of combining the isolated software structures into actual applications: they believe that we can combine them mechanistically, when the very reason for separating them was the impossibility of representing their totality mechanistically.

## 3

Like the other software theories, the delusion of the relational database model stems from the mechanistic fallacies of reification and abstraction: the belief that we can extract one aspect (the database structures, in this case) from the complex phenomenon that is a software application, and the belief that we can accomplish the same tasks by starting from high levels of abstraction as we can when starting from the lower levels.

So, as was the case with the ideas of structured programming and object-oriented programming, two great benefits are believed to emerge from the idea of a relational database. First, by reducing the application to strict hierarchical structures we will be able to represent software mathematically. Whether it is the flow of execution, the representation of a business process, or the database structures, we will deal with that aspect of the application formally, and thereby attain perfect, error-free software. Second, when we have strict hierarchical structures we can treat applications as systems of things within things. As we do in manufacturing, then, we will be able to use prefabricated software subassemblies, rather than depend on basic components. Application development will be easier and faster, since we will start our software projects from parts of a higher level of complexity – parts that already include other parts within them.

In the case of the relational model, this is accomplished by moving the low-level definitions and operations into the database system. All we need to do in the application is specify a relational operation, and the system will generate a database structure for us. In other words, not only do we have now a method that guarantees the correctness of the database, but this method consists of just a few simple, high-level operations. Instead of having to work with indexes and individual records, and with iterative and conditional constructs, we only need to understand now the concept of tables, of rows and columns: by extracting and combining portions of tables, we can generate all the database structures that we are likely to need in our applications.

Like the other software theories, the relational database model was seen as a critical step in the automation of programming. We only need to follow certain methods, and to use certain software systems; and because these methods and systems are based on mathematical concepts, we will end up with *provably* correct applications. It is already possible, we are told, to turn programming into a formal, routine activity. The concept of software engineering, if rigorously applied, already offers us the means to create perfect applications without depending on the skills or experience of individual programmers. And soon our systems will be powerful enough to eliminate the need for programmers altogether. Application development will then be completely automated: by means of sophisticated, interactive environments, managers and analysts will generate the application directly from the requirements.

❖

We discussed the fallacy of high-level starting elements in chapter 6. We saw, in particular, that even for a simple requirement like file maintenance we must link the file operations to the other types of operations – display and calculations, for instance. And consequently, if we want to be able to implement *any* file maintenance functions, we must start with the basic file operations and with the statements of a traditional programming language (see pp. 435–438).

Similarly, we can perhaps replace with a high-level operation the file operations and the logic needed to read a set of related records, but only for common requirements: comparing or totaling the values present in certain fields, displaying or printing these values, specifying some criteria for record selection or grouping, and the like. High-level operations are useless if what we need is a combination of file operations and logic peculiar to a particular requirement: displaying one field when a certain condition occurs and another field otherwise, performing one calculation for some records and another for other records, and so forth. Clearly, there is no limit to the number of situations that may require a particular combination of file operations and business requirements, so the idea of replacing the basic operations with high-level ones is absurd.

High-level database operations are useful, thus, if provided *in addition* to the basic file operations; that is, if we retain the means to create our own combinations of file operations, and use the high-level operations only when they are indeed more effective. But this is not how the relational database systems are presented. The concept of a database environment is promoted as a *replacement* of the basic file operations. We are told that these operations are no longer necessary, and that we must depend exclusively on the high-level relational ones.

It is easy to tell when starting from higher levels is indeed a practical alternative: the resulting operations are simple and beneficial. A good example is the idea of a mathematical function library: a collection of subroutines that evaluate for us, through built-in algorithms, various mathematical functions. Thus, simply by calling one of these subroutines, we can determine in the application such values as the logarithm or the sine of a variable. Because we seldom need to link the operations that make up these algorithms with the operations performed in the application, the calculations can be extracted from the rest of the application and replaced with independent modules – modules that interact with the application only through their input and output. And we notice the success of this idea in that the concept of a mathematical function library has remained practically unchanged over the years. One of the oldest programming concepts, the mathematical library is as simple and effective today as it was when first implemented. We didn't have to continually "improve" and "enhance" this concept, as we do our database systems.

But the best example of a practical move to higher levels is provided by the file operations themselves. The basic file operations are usually described as "low-level," but they are low-level only relative to the operations promised by database systems. The basic operations are executed by a file management system, and, relative to the operations performed internally by that system, their level is quite high. (The terms "basic file operations" and "file management system" will be discussed in greater detail in the next subsection; see pp. 672–673.) Thus, a simple statement that reads, writes, or deletes a record in the application becomes, when executed by the file management system, a complex set of operations involving indexes, buffers, search algorithms, and disk accesses. But because there are no links between these operations and the various operations performed by the application, they can be separated from the application and invoked by means of simple statements.

So, relative to the operations performed internally by the file management system, the basic file operations constitute in effect a library of file management functions, as do the mathematical functions relative to *their* internal operations. And, like the mathematical functions, the success of this concept is seen in the fact that it has remained practically unchanged since the 1960s, when it was introduced. But, just because we moved successfully from the low level of buffers and direct disk access to a level where all we need to do is read, write, and delete data records, it doesn't follow that we can move to an even higher level.

It is precisely because no higher levels are possible that the relational database systems evolved into such complicated environments. Were high-level database operations a practical idea, their use would be as straightforward as is the use of mathematical functions or basic file operations. The reason

these systems became increasingly large and complicated is that, in order to make them practical, their designers had to add more and more "features." These features are perceived as *enhancements* of the relational concept, but their real function is to counteract the *falsifications* of this concept. They may have new and fancy names, but these are features we always had – in our programming languages.

Thus, in order to save the relational theory from refutation, the software pseudoscientists had to incorporate into database systems *programming* features: means to define integrity and security checks, to access individual records, to deal with run-time errors, and so forth. These, obviously, are the low-level, application-related processes which they had originally hoped to eliminate through high-level database operations.

So the relational database systems became in the end a fraud: instead of admitting that the idea of high-level database operations had failed, the theorists reinstated the low-level capabilities of the traditional programming languages while making them look like features of a database system. Entire new languages had to be invented, in order to let programmers perform *within the database system* those operations they had been performing all along, through traditional programming languages, *within the application*.

The reason for the complexity of the relational systems, thus, is that they ended up incorporating concepts which belong in the application. Programming problems that are quite easy to solve as part of the application become awkward and complicated when separated from the application and moved into a database environment. Besides, the new languages are not as versatile as the traditional, general-purpose ones: they provide only *some* of the low-level elements we need, and only as artificial extensions to the high-level features. So our work is complicated also by having to solve low-level programming problems in a high-level environment. Like other development environments, the relational systems have reversed a basic programming principle: instead of freely creating high-level elements by combining low-level ones, we are forced to start with high-level elements, and to treat the low-level ones as extensions.

The idea of a database system emerged, we recall, not because of any requirements that could not be implemented with the basic file operations, but as an answer to the lack of programmers who could use these operations correctly. So, if what programmers must do now is even more difficult than is the use of file operations, how can the database systems help them? High-level database operations cannot replace programming expertise any more than could the idea of structured programming, or the idea of object-oriented programming.

# The Basic File Operations

1

To appreciate the inanity of the relational model, we must start by examining the basic file operations; that is, those operations which the relational systems are attempting to supplant. What I want to show is that these operations are *both necessary and sufficient* for implementing database management requirements, particularly in business applications. Thus, once we recognize the importance of the basic file operations, we will be in a better position to understand why the relational systems are fraudulent. For, as we will see, the only way to make them useful was by enhancing them with precisely those capabilities provided by the basic file operations; in other words, by restoring the very features that the database experts had claimed to be unnecessary.

Also, it is important to remember that the basic file operations have been available to programmers from the start, ever since mass storage devices with random access became popular. (They are sometimes called ISAM, Indexed Sequential Access Method.) For example, they have been available through COBOL (a language specifically designed for business applications) since about 1970. So these operations have always been well known: COBOL was always a public language, was implemented on all major computers, and was adopted by most companies. Thus, in addition to being an introduction to the basic file operations, this discussion serves to support my claim that the only motivation for database systems in general, and for the relational systems in particular, was to find a substitute for the knowledge required of programmers to use these operations correctly.

❖

Before examining the basic file operations, we must take a moment to clarify this term and the related terms "file operations" and "database operations." The basic file operations are a basic set of file management functions. They formed in the past an integral part of every major operating system, and were accessible through programming languages. These operations deal with *indexed data files* – the most versatile form of data storage; and, in conjunction with the features provided by the languages themselves, they allow us to use and to relate these files in any way we like.

"File operations" is a more general term. It refers to the basic file operations, but also to the various ways in which we combine them, using the flow-control constructs of a programming language, in order to implement file management requirements. "Database operations" is an even more general

term. It refers to the file operations, but in the context of the whole application, so it usually means *combinations* of file operations; in particular, combinations involving several files. The terms "traditional file operations" and "low-level file operations" refer to any one of the operations defined above.

The term "database" refers to a set of related files; typically, the files used by a particular application. Hence, the term "database system" ought to mean any software system that helps us to manage a database.[1] Through their propaganda, though, the software elites have created in our minds a strong association between terms like "database," "database system," and "database management system" (or DBMS) and *high-level* database operations. And as a result, most people believe that the only way to manage a database is through high-level operations; that the current database systems provide indispensable features; and that it is impossible to implement a serious application without depending on such a system.

But we must not allow the software charlatans to control our language and our minds. Since we can implement any database functions through the basic file operations and a programming language, systems that provide high-level operations are not at all essential for database management. So we can continue to use the terms "database" and "database operations" even while rejecting the notion of a system that restricts us to high-level operations.

Strictly speaking, since the basic file operations permit us to manage a database, they too form a database system. But it would be confusing to use this term for the basic operations, now that it is associated with the high-level operations. Thus, I call the systems that provide basic file operations "*file* management systems," or "*file* systems" for short. This term is quite appropriate, in fact, seeing that these systems are limited to operations involving single files; it is *we* who implement the actual database management, by combining the operations provided by the file system with those provided by a programming language.

So I use the term "database," and terms like "database operations" and "database management," to refer to *any* set of related files – regardless of whether the files and relations are managed through the high-level operations of a *database* system, or through the basic operations of a *file* system.

The term "database structures" refers to the various hierarchical structures created by the files that make up the database: related files can be seen as the levels of a structure, and their records as the elements that make up these levels (see p. 688). In most applications, the totality of database structures is a complex structure.

[1] The term "database system" is used by everyone as an abbreviation of "database management system." It is somewhat misleading, though, since it sounds as if it refers to the database itself.

# 2

Two types of files make up the database structures of an application: *data* files and *index* files. The data files contain the actual data, organized as *records*; the index files (or indexes, for short) contain the pointers that permit us to access these records.

The record is the unit that the application typically reads from the file, or writes to the file. But within each record the data is broken down into *fields*, and it is the values present in the individual fields that we normally use in the application. For example, if each record in the file has 100 bytes, the first field may take the first 6 bytes, the second one the next 24 bytes, and so on. This is how the fields reside on disk, and in memory when the record is read from disk, but in most cases their relative order within the record is immaterial. For, in the application we assign names to these fields, and we refer to them simply by their names. Thus, once a record is read into memory, we treat database fields, for all practical purposes, as we do memory variables.

The records and fields of a data file reflect the structure and type of the information stored in the file. In an employee file, for example, there is a record for each employee, and each record contains such fields as employee number, name, salary, and year-to-date earnings and deductions; in a sales history file there is a record for each line in a sales order, with such fields as the customer and order numbers, date, price, and quantity sold. While in simple cases the required fields are self-evident, generally it takes some experience to design the most effective database for a given set of requirements. We must decide what information should be processed by the application, how to represent this information, how to distribute it among files, how to index the files, and how to relate them. Needless to say, it is impossible to predict all future requirements, so we must be prepared to alter the application's database structure later: we may need to add or delete fields, move fields from one file to another, and create new files or indexes.

We don't normally access data records directly, but through an index. Indexes, thus, are service files, means to access the data files. Indexes fulfil two essential functions: they allow us to identify a specific record, and to scan a series of records in a specific sequence. It is through *keys* that indexes perform these tasks. The key is one of the fields that make up the record, or a set of several fields. Clearly, if the combination of values present in these fields is different for each record in the file, each record can be uniquely identified. In addition, key uniqueness allows us to scan the records in a particular sequence – the sequence that reflects the current key values – regardless of their actual,

physical sequence on disk. When the key is one field, the value present in the field is the value of the key. When the key consists of several fields, the value of the key is the combination of the field values, in the order in which they make up the key. The records are scanned, in effect, in a sorted sequence. For example, if the key is defined as the set of three fields, *A*, *B*, and *C*, the sorting sequence can be expressed as either "by *A* by *B* by *C*" or "by *C* within *B* within *A*."

Note that if we permit *duplicate* keys – if, that is, some combinations of values in the key fields are not unique – we will be unable to identify the individual records within a set of duplicates. Such an index is still useful, however, if all we need is to *scan* those records. The scanning sequence within a set of duplicate records is usually the order in which they were added to the file. Thus, for scanning too, if we want better control we must ensure key uniqueness.

An especially useful feature is the capability to create several indexes for the same data file. This permits us to access the same records in different ways – scan the file in one sequence or another, or read a record through one key or another. For example, we may scan a sales history file either by order number or by product number; or, we may search for a particular sales record through a key consisting of the customer number and order number, or through a key consisting of the product number and order date.

Another useful indexing feature is the option of *descending* keys. The normal scanning sequence is *ascending*, from low to high key values; but some file systems also allow indexes that scan records from high to low key values. Any one field, or all the fields in the key, can then be either ascending or descending. Simply by scanning the data file through such an index we can list, for instance, orders in ascending sequence by customer number, but within each customer those orders with a higher amount first; or we can list the sales history by ascending product number, but within each product by descending date (so those sold most recently come first), and within each date by ascending customer number. A related indexing feature, useful in its own right but also as an alternative to descending keys, is the capability to scan records backward.

In addition to indexed data files, most file management systems support two other types of files, *relative* and *sequential*. These files provide simpler record access, and are useful for data that does not require an elaborate indexing scheme. In relative data files, we access a record by specifying its relative position in the file (first, second, third, etc.). These files are useful, therefore, in situations where the individual records cannot, or need not, be identified by the values present in their fields (to store the entries of a large table, for instance). Sequential data files are organized as a series of consecutive

records, which can only be accessed sequentially, starting from the beginning. These files are useful in situations where we don't need to access individual records directly, and where we normally read the whole file anyway (to store data that has no specific structure, for instance). Text data, too, is usually stored in sequential files. I will not discuss further the relative and sequential files. It is the indexed data files that interest us, because it is only *their* operations that the relational database systems are attempting to replace with high-level operations.

❖

File systems provide at least two types of fields, *alphanumeric* (or *alpha*, for short) and *numeric*. And, since these types are the same as the memory variables supported by most high-level languages (COBOL, in particular), database fields and memory variables can be used together, and in the same manner, in the application. In alphanumeric fields, data is stored as character symbols, so these fields are useful for names, addresses, descriptions, notes, identifiers, and the like. When these fields are part of an indexing key, the scanning sequence is alphabetical. In numeric fields, the data is stored as numeric values, so these fields can be used directly in calculations. Numeric fields are useful for any data that can be expressed as a numeric value: quantities, dollar amounts, codes, and the like. When these fields are part of an indexing key, the scanning sequence is determined by the numeric value.

Some file systems provide additional field types. *Date* fields, for instance, are useful for storing dates. In the absence of date fields, we must store dates in numeric fields, as six- or eight-digit values representing the combination of the month, day, and year; alternatively, we can store dates as values representing the number of days elapsed since some arbitrary, distant date in the past. (The latter method is preferable, as it simplifies date calculations, comparisons, and indexing.) Another field type is the *binary* field, used to store such data as text, graphics, and sound; that is, data which can be in any format whatsoever (hence "binary," or raw), and which may require many thousands of bytes. (Because of its large size, this data is stored in separate files, and only pointers to it are kept in the field itself.)

## 3

Now that we have examined the structure of indexed data files, let us review the basic file operations. Six operations, combined with the iterative and conditional constructs of high-level languages, are all we need in order to use

indexed data files. I will first describe these operations, and then show how they are combined with language features to implement various requirements. The names I use for the basic operations are taken from COBOL. (There may be some small variations in the way these operations are implemented in a particular file system, or in a particular version of COBOL; for example, in the way multiple indexes or duplicate keys are supported.)

The following terms are used in the description of the file operations: The *current index* is the index file specified in the operation. *File* is a data file; although the file actually specified in the operation is an index file, the record read or written belongs to the data file (we always access a data file through one of its indexes). *Record area* is a storage area – the portion of memory where the fields that make up the record are specified; each file has its own record area, and this area is accessed by both the file system and the application (the application treats the fields as ordinary memory variables). *Key* is the field or set of fields, within the record area, that was defined as the key of a particular index; the *current key* is the key that was defined for the current index. The record *pointer* is an indicator maintained by the file system to identify the next record in the scanning sequence established by a particular index; each index has its own pointer, and the *current pointer* is the pointer corresponding to the current index.

WRITE: A new record is added to the file. Typically, the data in this record consists of the values previously placed by the application into the fields that make up the file's record area. The values present in the fields that make up the current key will become the new record's key in the current index. If the file has additional indexes, the values in their respective key fields will become the keys in those indexes. All indexes are updated together: following this operation, the new record can be accessed either through the current index or through another index. If one of the file's indexes does not permit duplicate keys and the new record would cause such a condition, the operation is aborted and the system returns an error code (so that the application can take appropriate action).

REWRITE: The data in the record area replaces the data in the record currently in the file. Typically, the application read previously the record into the record area through the current index, and modified some of the fields. The record is identified by the current key, so the fields that make up this key should not be modified. If there are other indexes, the fields that make up their keys may be modified, and REWRITE will update those indexes to reflect the change. REWRITE, however, can also be used without first reading the existing record: the application must place some values in all the fields, and REWRITE functions then like WRITE, except that it replaces an existing record. In either case, if no record is found with the current key, or if one of the file's indexes

does not permit duplicate keys and the modified record would cause such a condition, the operation is aborted and the system returns an error code.

DELETE: The record identified by the current key is removed from the file. Only the values present in the current key fields are important for the operation; the rest of the record area is ignored. The application, therefore, can delete a record either by reading it first into the record area (through any one of its indexes) or just by placing the appropriate values into the current key fields. If no record is found with the current key, the system returns an error code.

READ: The record identified by the current key is read into the record area. The current index can be any one of the file's indexes, and only the values present in the current key fields are important for the operation. Following this operation, the fields in the record area contain the values present in that record in the file. If no record is found with the current key, the system returns an error code.

START: The current pointer is positioned at the record identified by the current key. The current index can be any one of the file's indexes, and only the values present in the current key fields are important for the operation. The specification for the operation includes a relation like *equal*, *greater*, or *greater or equal*, so the application need not indicate a valid key; the record identified is simply the first one, in the scanning sequence of the current index, whose key satisfies the condition specified (for example, the first one whose key is *greater* than the values present in the current key fields). If no record in the file satisfies that condition, the system returns an error code.

READ NEXT: The record identified by the current pointer is read into the record area. This operation, in conjunction with START, makes the file scanning feature available to the application. The application must first perform a START for the current index, in order to set the current pointer at the first record in the series of records to be scanned. (To indicate the first record in the file, null values are typically placed in the key fields, and the condition *greater* is specified.) READ NEXT will then read that record and advance the pointer to the next record in the scanning sequence of the current index. The subsequent READ NEXT will read the record indicated by the pointer's new position and advance the pointer to the next record, and so on. Through this process, then, the application can read a series of consecutive records without having to know their keys.[2] Typically, READ NEXT is part of a loop, and the application knows when the last record in the series is reached by checking a certain condition (for example, whether the key exceeds a particular value). If the pointer was already positioned past the last record in the file (the *end-of-file* condition), the

[2] Since no search is involved, it is not only simpler but also faster to read a record in this fashion, than by specifying its key. Thus, even when the keys are known, it is more efficient to read consecutive records with READ NEXT than with READ.

system returns an error code. (Simply checking for this code after each READ NEXT is how applications typically handle the situation where the last record in the series is also the last one in the file.)

❖

These six operations form the minimal practical set of file operations: the set of operations that are both necessary and sufficient for using indexed data files in serious applications.[3] I will demonstrate now, with a few examples, how the basic file operations are used in conjunction with other types of operations to implement typical requirements. Again, I am describing COBOL constructs and statements, but the implementation would be very similar in other high-level languages.

A common requirement involves the *display* of data from a particular record: the user identifies the record by entering the value of its key (customer number, part number, invoice number, and the like), and the application responds by retrieving that record and displaying some of its fields. When the key consists of several fields, the user must enter several values. To implement this operation in the application, all we need is a READ: we place the values entered by the user into the current key fields, perform the READ, and then display for the user various fields from the record area. If, however, the system returns an error code, we display a message such as "record not found."

If the user wants to *modify* some of the fields in a particular record, we start by performing a READ and displaying the current values, as before; but then we allow the user to enter the new values, place them in the appropriate fields in the record area, and perform a REWRITE. And if what the user wants is to *delete* a particular record, we usually start with a READ, display some of the fields to allow the user to confirm it is the right record, and then perform a DELETE.

Lastly, to *add* a record, we display blank fields and allow the user to enter their actual values. (In a new record, some fields may have null values, or some default values; so these fields may be left out, or just displayed, or displayed with the option to modify them.) The user must also enter the value of the key fields, to identify the new record. We then perform a WRITE, and the system will add this record to the file. If, however, it returns an error code, we display a message such as "duplicate key" to tell the user why the record could not be added.

[3] I will not discuss here the various *support* operations – opening and closing files, locking and unlocking records in multiuser applications, and the like. Since there is little difference between these operations in file systems and in database systems, they have no bearing on my argument. Many of these operations can be performed automatically, in fact, in both types of systems.

Examples of this type of record access are found in the *file maintenance* operations – those operations that permit the user to add, delete, and modify records in the database. And, clearly, any maintenance requirement can be implemented through the basic file operations: any file, record, and field in the database can be read, displayed, or modified. If we must restrict this freedom (permit only a range of values for a certain field, permit the addition or deletion of a record only under certain conditions, etc.), all we have to do is add appropriate checks; then, if the checks fail, we bypass the file operation and display a message.

So far I have discussed the *interactive* access of individual records, but the basic file operations are used in the same way when the user is not directly involved. Thus, if we need to know at some point in the application the quantity on hand for a certain part, we place the part number in the key field, perform a READ, and then get the value from the quantity field; if we want to add a new transaction to the sales history file, we place the appropriate values in the key fields (customer number, invoice number, etc.) and in the non-key fields (date, price, quantity, etc.), and perform a WRITE; if we want to update a customer's balance, we place the customer number in the key field, perform a READ, calculate the new value, place it in the balance field, and then perform a REWRITE. Again, any conceivable requirement can be implemented through the basic file operations.

❖

Accessing *individual* records, as described above, is one way of using indexed data files. The other way is by *scanning* records, an operation accomplished with an iterative construct based on START and READ NEXT. This construct, which may be called the basic file scanning loop, is used every time we read a series of records sequentially through an index. The best way to illustrate this loop is with a simple example (see figure 7-13). The loop here is designed to read the PART file in ascending part number sequence. The indexing key, P-KEY, consists of one field, P-NUM (part number). START positions the record pointer so that the first record read has a part number no less than P1, and the

```
  MOVE P1 TO P-NUM START PART KEY>=P-KEY INVALID GO TO L4.
L3. READ PART NEXT END GO TO L4. IF P-NUM>P2 GO TO L4.
  IF P-QTY<Q1 GO TO L3.
  [various operations]
  GO TO L3.
L4.
```

Figure 7-13

condition >P2 terminates the loop at the first record with a part number greater than P2. The loop will read, therefore, only the *range* of records, P1 through P2, inclusive.[4] In addition, within this range, the loop selects only those records where the quantity field, P-QTY, is no less than a certain value, Q1. The operations following the selection conditions will be performed for every record that satisfies these conditions. The labels L3 and L4 delimit the loop.[5]

We rarely perform the same operations with all the records in a file, so the selection of records is a common requirement in file scanning. The previous example illustrates the two selection methods – based on key fields, and on non-key fields. The method based on key fields is preferable when what we select is a range of records, as the records left out don't even have to be read. This can greatly reduce the processing time, especially if the file is large and the range selected is relatively small. In contrast, when the selection is based on non-key fields, each record in the file must be read. This is true because the value of non-key fields is unrelated to the record's position in the scanning sequence, so the only way to know what the values are is by reading the record. The two methods are often combined in the same loop, as illustrated in the example.

It should be obvious that these two selection methods are completely general, and can satisfy any requirement. For example, if the range must include all the records in the file, we specify null values for the key fields in START and omit the test for the end of the range. The loop also deals correctly with the case where no records should be selected (because there are none in the specified range, or because the selection based on non-key fields excludes all those in the range). It must be noted that the selection conditions can be as complex as we need: they can involve several fields, or fields from other files (by reading in the loop records from those files), or a combination of fields, memory variables, and constants. A complex condition can be formulated either as one complex IF statement or as several consecutive IF statements. And,

[4] Note the END clause in READ NEXT, specifying the action to take if the end of the file is reached before P2. (INVALID and END are the abbreviated forms of the COBOL keywords INVALID KEY and AT END. Similarly, GO TO can be abbreviated in COBOL as GO.)

[5] It is evident from this example that the most effective way to implement the basic file scanning loop in COBOL is with GO TO jumps. This demonstrates again the absurdity of the claim that GO TO is harmful and must be avoided (the delusion we discussed under structured programming). Modifying this loop to avoid the GO TOs renders the simple operations of file scanning and record selection complicated and abstruse; yet this is exactly what the experts have been advocating since 1970. It is quite likely that the complexity engendered by the delusions of structured programming contributed to the difficulty programmers had in using file operations, and was a factor in the evolution of database systems: because they tried to avoid the complications created by one pseudoscience, programmers must now deal with the greater complications created by another.

in addition to the conditions that affect all the operations in the loop, we can have conditions *within* the loop; any portion of the loop, therefore, can be restricted to certain records.

Let us see now how the basic file scanning loop is used to implement various file operations. In a typical file listing, or query, or report, the scanning sequence and the record selection criteria specified by the user become the index and the selection conditions for the scanning loop. And within the loop, for each record selected, we show certain fields and perhaps accumulate their values. Typically, one line is printed or displayed for each record, and the totals are shown at the end. When the indexing key consists of several fields, their value will change hierarchically, one within another, in the sorting sequence of the index; thus, we can have various levels of subtotals by noting within the loop when the value of these fields changes. In an orders file, for instance, if the key consists of order number within customer number, and if we need the quantity and amount subtotals for the orders belonging to each customer, we must show and then clear these subtotals every time the customer number changes.

Another use of the scanning loop is for *modifying* records. The reading and selection are performed as before, but here we modify the value stored in certain fields; then we perform a REWRITE (at the end of the loop, typically). This is useful when we must modify a series of records according to some common logic. Not all the selected records need to be modified, of course; we can perform some calculations and display the results for all the records in a given range, for instance, but modify only those where the fields satisfy a certain condition. Rather than modify records, we can use the scanning loop to *delete* certain records; in this case we perform a DELETE at the end of the loop.

An interesting use of indexed data files is for sorting. If, for instance, we need a listing of certain values in a particular scanning sequence (values derived from files or from calculations), we create a temporary data file where the indexing key is the combination of fields for that scanning sequence, while the non-key fields are the other values to be listed. All we have to do then is perform a WRITE to add a record to the temporary file for each entry required in the listing. The system will build for us the appropriate index, and, once complete, we can scan the temporary file in the usual manner. Similarly, if we need to scan a portion of a data file in a certain sequence, but only occasionally, then instead of having a permanent index for that sequence we create a temporary data file that is a subset of the main data file: we read the main data file in a loop through one of its indexes, and for each selected record we copy the required fields to the record of the temporary file and perform a WRITE.

If we want to analyze certain fields in a data file according to the value present in some other fields (total the quantity by territory, total various

amounts by the combination of territory and category, etc.), we must create a temporary data file where the indexing key is the field or combination of fields by which we want to group the records (the *analysis* fields in the main data file), while the non-key fields are the values to be totaled (the *analyzed* fields). We read the main file in a loop, and, for each record, we copy the analysis values and the analyzed values to the respective fields in the record of the temporary file. We then perform a WRITE for this file and check the return code. If the system indicates that the record already exists, it means this is not the first time that combination of key values was encountered; the response then is to perform a READ, *add* the analyzed values to the respective fields, and perform a REWRITE. In other words, we create a new record in the temporary file only the first time a particular combination of analysis values is encountered, and *update* that record on subsequent occasions. At the end, the temporary file will contain one record for each unique combination of analysis values. This concept is illustrated in figure 7-14.

```
    MOVE C1 TO C-NUM START CUSTOMER KEY>=C-KEY INVALID GO TO L4.
L3. READ CUSTOMER NEXT END GO TO L4. IF C-NUM>C2 GO TO L4.
    MOVE C-TER TO SR-TER MOVE C-QTY TO SR-QTY.
    WRITE SR-RECORD INVALID READ SORTFL
      ADD C-QTY TO SR-QTY REWRITE SR-RECORD.
    GO TO L3.
L4.
```

Figure 7-14

In this example, a certain quantity in the CUSTOMER file is analyzed by territory for the customers in the range C1 through C2. SORTFL is the temporary file, and SR-RECORD is its record area. The simplicity of this operation is due to the fact that much of the logic is implicit in the READ, WRITE, and REWRITE.

## 4

One of the most important uses of the file scanning loop is to *relate* files. If we nest the scanning loop of one file within that of another, a logical relationship is created between the two files. From a programming standpoint, the nesting of file scanning loops is no different from the nesting of any iterative constructs: the whole series of iterations through the inner loop is repeated for every iteration through the outer loop. In the inner loop we can use fields from both files; any operation, therefore, including the record selection conditions, can depend on the record currently read in the outer loop.

Figure 7-15 illustrates this concept. The outer loop scans the CUSTOMER file and selects the range of customer numbers C1 through C2. The indexing key, C-KEY, consists of one field, C-NUM (customer number). Within this loop, in addition to any other operations performed for each customer record, we include a loop that scans the ORDERS file. The indexing key here, O-KEY, consists of two fields, O-CUS (customer number) and O-ORD (order number), in this sorting sequence. Thus, to restrict the inner loop to the orders belonging to one customer, we select only the range of records where the customer number equals the one currently read in the outer loop, while allowing the order number to be any value. (Note that the terminating condition, "IF O-CUS NOT=C-NUM," could be replaced with "IF O-CUS>C-NUM," since the first O-CUS read that is not equal to C-NUM is necessarily greater than it.) The inner loop here selects *all* the orders for the customer read in the outer loop; but we could have additional selection conditions, based on non-key fields, as in figure 7-13 (for example, to select only orders in a certain date range, or over a certain amount).

```
  MOVE C1 TO C-NUM START CUSTOMER KEY>=C-KEY INVALID GO TO L4.
L3. READ CUSTOMER NEXT END GO TO L4. IF C-NUM>C2 GO TO L4.
  [various operations]
  MOVE C-NUM TO O-CUS MOVE 0 TO O-ORD.
  START ORDERS KEY>O-KEY INVALID GO TO L34.
L33. READ ORDERS NEXT END GO TO L34. IF O-CUS NOT=C-NUM GO TO L34.
  [various operations]
  GO TO L33.
L34.
  [various operations]
  GO TO L3.
L4.
```

Figure 7-15

Although most file relations involve only two files, the idea of loop nesting can be used to relate hierarchically any number of files, simply by increasing the number of nesting levels. Thus, by nesting a third loop within the second one and using the same logic, the third file will be related to the second in the same way that the second is related to the first. With two files, we saw, the second file's key consists of two fields, and the range selected includes the records where the first field equals the first file's key. With three files, the third file's key must have three fields, and the range will include the records where the first two fields equal the second file's key. (The keys may have additional fields; two and three are the minimum needed to implement this logic.)

To illustrate this concept, figure 7-16 adds to the previous example a loop to scan the LINES file (the individual item lines associated with each order).

If ORDERS has fields like customer number, order number, date, and total amount, which apply to the whole order, LINES has fields like item number, quantity, and price, which are different for each line. Its indexing key consists of customer number, order number, and line number, in this sorting sequence. And the third loop isolates the lines belonging to a particular order by selecting the range of records where the customer and order numbers equal those of the order currently read in the second loop, while the line number is any value. Another example of a third nesting level is a transaction file, where each record is an invoice, payment, or adjustment pertaining to an order, and the indexing key consists of customer number, order number, and transaction number.[6]

```
  MOVE C1 TO C-NUM START CUSTOMER KEY>=C-KEY INVALID GO TO L4.
L3. READ CUSTOMER NEXT END GO TO L4. IF C-NUM>C2 GO TO L4.
  [various operations]
  MOVE C-NUM TO O-CUS MOVE 0 TO O-ORD.
  START ORDERS KEY>O-KEY INVALID GO TO L34.
L33. READ ORDERS NEXT END GO TO L34. IF O-CUS NOT=C-NUM GO TO L34.
  [various operations]
  MOVE O-CUS TO L-CUS MOVE O-ORD TO L-ORD MOVE 0 TO L-LINE.
  START LINES KEY>L-KEY INVALID GO TO L334.
L333. READ LINES NEXT END GO TO L334.
  IF NOT(L-CUS=O-CUS AND L-ORD=O-ORD) GO TO L334.
  [various operations]
  GO TO L333.
L334.
  [various operations]
  GO TO L33.
L34.
  [various operations]
  GO TO L3.
L4.
```

Figure 7-16

Note that in the sections marked "various operations" we can access fields from all the currently read records: in the outer loop, fields from the current CUSTOMER record; in the second loop, fields from the current CUSTOMER and ORDERS records; and in the inner loop, fields from the current CUSTOMER, ORDERS, and LINES records.

Note also that the sections marked "various operations" may contain additional file scanning loops; in other words, we can have more than one

[6] Note, in figures 7-13 to 7-16, the numbering system used for labels in order to make the jumps self-explanatory (as discussed under the GOTO delusion, pp. 621–624).

scanning loop at a given nesting level. For instance, by creating two consecutive third-level loops, we can scan first the lines and then the transactions of the order read in the second-level loop.

The arrangement where the key used in the outer loop is part of the key used in the inner loop, as in these examples, is the most common and the most effective way to relate files, because it permits us to select records through their key fields (and to read therefore only a *range* of records). We can also relate files, though, by using non-key fields to select records (when it is practical to read the entire file in the inner loop).

Lastly, another way to relate files is by reading within the loop of one file just one record of another file, with no inner loop at all (or, as a special case, reading just one record in both files, with no outer loop either). Imagine that we are scanning an invoice file where the key is the invoice number and one of the key or non-key fields is the customer number, and that we need some data from the customer record – the name and address fields, for instance. (This kind of data is normally stored only in the customer record because, even though required in many operations, it is the same for all the transactions pertaining to a particular customer.) So, to get this data, we place the customer number from the currently read invoice record into the customer key field, and perform a READ. All the customer fields are then available within the loop, along with the current invoice fields.



The relationship just described, where several records from one file point to the same record in another file, is called *many-to-one* relationship. And the relationship we discussed previously, where one record from the first file points to several records in the second file (because several records are read in the inner loop for each record read in the outer loop) is called *one-to-many* relationship. These two types of file relationships are the most common, but the other two, *one-to-one* and *many-to-many*, are also important.

We have a one-to-one relationship when the same field is used as a key in two files. For example, if in addition to the customer file we create a second file where the indexing key is the customer number (in order to store some of the customer data separately), then each record in one file corresponds to one record in the other. And we have a many-to-many relationship when one record in the first file points to several records in the second one, and at the same time one record in the second file points to several records in the first one. (We will study the four types of file relationships in greater detail later; see pp. 738–741.)

To understand the many-to-many relationship, imagine a factory where a

number of different products are being built by assembling various parts from a common inventory. Thus, each product is made from a number of different parts, and at the same time a part may be used in different products. The product file has one record for each product, and the key is the product number. And the part file has one record for each part, and the key is the part number. We can use these files separately in the usual manner, but to implement the many-to-many relationship between products and parts we need an additional file – a service file for storing the cross-references. This file is a dummy data file that consists of key fields only. It has two indexes: in the first one the key is the product number and the part number, and in the second one it is the part number and the product number, in these sorting sequences. In the service file, therefore, there will be one record for each pair of product and part that are related in the manufacturing process (far more records, probably, than there are either products or parts). Now we can scan the product file in the outer loop, and the service file, through its first index, in the inner loop; or, we can scan the part file in the outer loop, and the service file, through its second index, in the inner loop. Then, by selecting in the inner loop a range of records in the usual manner, we will read in the first case the parts used by a particular product, and in the second case the products that use a particular part. What is left is to perform a `READ` in the inner loop using the part or product number, respectively, in order to read the actual records.

## The Lost Integration

The preceding discussion was not meant to be an exhaustive study of indexed data files. My main intent was to show that any conceivable database requirement can be implemented with file operations, and that this is a fairly easy programming challenge: every one of the examples we examined takes just a few statements in COBOL. We only need to understand the two ways of using indexes (reading individual records or scanning a range of records) and the two ways of selecting records (through key fields or non-key fields). Then, simply by combining the basic file operations with the other operations available in a programming language, we can access and relate the files in the database in any way we like.

So the difficulties encountered by programmers are not caused by the basic file operations, nor by the selection of records, nor by the file scanning loops. The difficulties emerge, rather, when we combine file operations, and when we combine them with the other types of operations required by the application. The difficulties, in other words, are due to the need to deal with

interacting software structures. Two kinds of structures, and hence two kinds of interactions, are generated: one through the file relationships we discussed earlier (one-to-many, many-to-many, etc.), the other through the links created between the application's elements by the file operations.

Regarding the first kind of structures, the file relationships are easy to understand individually, because we can view them as simple hierarchical structures. If we depict the nesting of files as a structure, each file can be seen as a different level of the structure, and its records as the various elements which make up that level. The relationship between files is then the relationship between the elements of one level and the next. But, even though each relationship is hierarchical, most files take part in several relationships, through different fields. In other words, a record in a certain file can be an element in several structures at the same time, so these structures interact. The totality of file relationships in the database is a complex structure.

As for the second kind of structures, we already know that the file operations give rise to processes based on shared data (see pp. 349–351). So they link the application's elements through many structures – one structure for each field, record, or file that is accessed by several elements. Thus, in addition to the interactions due to the file relationships, we must cope with the interactions between the structures generated by file operations. And we must also cope with the interactions between these structures and the structures formed by the other types of processes – practices, subroutines, memory variables, etc. To implement database requirements we must deal with complex software structures.

When replacing the basic file operations with higher-level operations, what are the database experts trying to accomplish? All that a database system can do is replace with a built-in process the two or three statements that constitute the use of a basic file operation. The experts misinterpret the difficulty that programmers have in implementing file operations as the problem of dealing with the relatively low levels. But, as we saw, the difficulty is not due to the individual file operations, nor to the individual relationships. The difficulty emerges when we deal with *interacting* operations and relationships, and with their interaction with the rest of the application. And these interactions cannot be eliminated; we must have them in a database system too, if the application is to do what we want it to do. Even with a database system, then, the difficult part of database programming remains. The database systems can perhaps replace the easy challenges – the individual operations; but they cannot eliminate the difficult part – the need to deal with interacting structures.

What is worse, database systems make the interactions even more complex, because some of the operations are now in the application while others are in the database system. The original idea was to have database functions akin to

the functions provided by a mathematical library; that is, entities of a high level of abstraction, which interact with the application only through their input and output. But this is impossible, because database operations must interact with the rest of the application at a lower level – at the level of fields, variables, and conditions. Thus, the level of abstraction that a database system can provide while remaining a practical system is not as high as the one provided by a mathematical library. We cannot extract, for example, a complete file scanning loop, with all the operations in the loop, and turn it into a high-level database function – not if we want to retain the freedom of implementing *any* scanning loops and operations.

❖

All we needed before was the six basic file operations. The database operations, and their interaction with the rest of the application, could then be implemented with the same programming languages, and with the same methods and principles, that we use for the other operations in the application. With a database system, on the other hand, we need new and complicated principles, languages, rules, and methods; we must deal with a new kind of operations in the database system, plus a new kind of operations in the application, the latter necessary in order to link the application to the database system. So, in the end, the difficulties faced by programmers in implementing database operations are even greater than before.

It is easy to see why the basic file operations are both necessary and sufficient for implementing database operations: for most applications – business applications, in particular – they are just the right level of abstraction. The demands imposed by our applications rarely permit us to move to higher levels, and we rarely need lower ones. An example of lower-level file operations is the requirement for a kind of fields, indexes, or records that is different from the one provided by the standard data files. And, in the rare situations where such a requirement is important, we can implement it in a language like C. Similarly, in those situations where we can indeed benefit from higher-level operations, we can create them by means of subroutines in the same language as the application itself: we design the appropriate combination of basic file operations and flow-control constructs, store it as a separate module, and invoke it whenever we need that particular combination.

For the vast majority of applications, however, we need neither lower nor higher levels, since the level provided by the basic file operations is just right. This level is similar to the level provided, for general programming requirements, by our high-level languages. With the features found in a language like COBOL, for instance, we can implement any business application. Thus, it

is no coincidence that, in conjunction with the operations provided by a programming language, the basic file operations can be used quite naturally to implement practically all database operations, and also to link these operations to the other types of operations: iterative constructs are just right for scanning a data file sequentially through one of its indexes; nested iterations are just right for relating files hierarchically; conditional constructs are just right for selecting records; and assignment constructs are just right for moving data between fields, and between fields and memory variables. It is difficult to find a single database operation that cannot be easily and naturally implemented with the constructs found in the traditional languages.

This flexibility is due to the correct level of abstraction of both the basic file operations and the traditional languages. This level is sufficiently low to make all conceivable database operations possible, and at the same time sufficiently high to make them simple and convenient – for an experienced programmer, at least. We can so easily implement any database requirement using ordinary features, available in most languages, that it is silly to search for higher-level operations.

High-level database operations offer no benefits, therefore, for two reasons: first, because we can so easily implement database requirements using the basic file operations, and second, because it is impossible to have built-in operations for all conceivable situations. No matter how many high-level operations we are offered, and no matter how useful they are, we will always encounter requirements that cannot be implemented with high-level operations alone. We cannot give up the lower levels, thus, because we need them to implement details, and because the links between database operations, and also between database operations and the other types of operations, occur at the low level of these details.

So the idea of higher levels is fallacious for database operations in the same way it is fallacious for the other types of operations. This was also the idea behind the so-called fourth-generation languages (see pp. 452–453). And, like the 4GL systems, the relational systems became in the end a fraud.

The theorists start by promising us higher levels. Then, when it becomes clear that the restriction to high levels is impractical, they restore – in the guise of enhancements – the low levels. Thus, with 4GL systems we still use such concepts as conditions, iterations, and assigning values to variables; in other words, concepts of the same level of abstraction as those found in a traditional language. It is true that these systems provide *some* higher-level operations (in user interface, for instance), but they do not eliminate the lower levels. In any case, even in those situations where operations of a higher level are indeed useful, we don't need these systems; for, we can always provide the higher levels ourselves, in any language, through subroutines. Similarly, we will see in the

present section, the relational database systems became practical only after restoring the low levels; that is, the traditional file management concepts.

In conclusion, the software elites promote ideas like 4GL and relational databases, not on the basis of any real benefits, but in order to deprive us of the programming freedom conferred by the traditional languages. Their real motive is to force us to depend on expensive and complicated development systems, which they control.



I want to stress again that remarkable quality found in the basic file operations, the fact that they are at the same level of abstraction as the operations provided by the traditional programming languages. This is why we can so easily link these operations and implement database requirements. One of the most successful of all software concepts, this simple feature greatly simplifies both programming and the resulting applications.

There is a *seamless integration* of the database and the rest of the application, for both data and operations. The fields, the record area, and the record keys function as both database entities and memory variables at the same time. Database fields can be mixed freely with memory variables in assignments, calculations, or comparisons. Transferring data between disk and memory is a logical extension of the data transfers performed in memory. Most statements, constructs, and methods we use in programming have the same form and meaning for file operations as they have for the other types of operations; iterative and conditional constructs, for example, are used in the same way to scan and select records from a file as they are to scan and select items from an array or table stored in memory.

Just by learning to use the six basic file operations, then, a programmer gains the means to design and control databases of any size and complexity. The most difficult part of this work is handled by the file management system, and what is left to the programmer is not very different from the challenges he faces when dealing with any other aspect of the application.

The seamless integration of the database and the application is such an important feature that, had we not already had it in the traditional file operations, we could have rightly called its introduction today a breakthrough in programming techniques. The ignorance of the academics and the practitioners is betrayed, thus, by their lack of appreciation of a feature that has been widely available (through COBOL, for instance) since the 1960s. Instead of studying it and learning how to make the most of it, the software experts have been promoting the relational model, whose express purpose is to *eliminate* the integration. In their attempt to simplify programming, they restrict the links

between files, and between files and the rest of the application, to high levels of abstraction. But this is an absurd idea, as we saw, because serious applications require low-level links too.

Then, instead of admitting that the relational model had failed, the experts proceeded to *reestablish* the low-level links. For, in order to make the relational model practical, they had to restore the integration – the very quality that the relational model had tried to eliminate. But the only way to provide the low levels and the integration now, as part of a database system, is through a series of artificial enhancements. When examined, the new features turn out to be nothing but particular instances of the important quality of integration: they are means to link the database to the rest of the application in specific situations. What is the very nature of the traditional file operations, and in effect just one simple feature, is now being restored by annulling the relational principles and replacing them with a multitude of complicated features. Each new feature is, in reality, a substitute for a particular high-level software element (a particular database function) that can no longer be implemented naturally, by combining lower-level elements.

Like all development systems that promise a higher level of abstraction, the relational systems became increasingly large and complicated because they attempted to replace with built-in operations the infinity of alternatives that we need at high levels but can no longer create by starting from low levels. Recall the analogy of software with language: If we had to express ourselves through ready-made sentences, instead of creating our own starting with words, we would end up depending on systems that become increasingly large and complicated as they attempt to provide all necessary sentences. But even with thousands of sentences, we would be unable to express all possible ideas. So we would spend more and more time trying to communicate through these systems, even while being restricted to a fraction of the ideas that can be expressed by combining words.

Thus, the endless problems engendered by relational database systems, and the astronomic cost of using them, are due to the ongoing effort to overcome the restrictions imposed by the relational model. They are due, in the end, to the software experts, who not only failed to understand why this model is worthless, but continued to promote it while its claims were being falsified.

The relational model became a pseudoscience when the experts decided to "enhance" it, which they did by turning its falsifications into features (see "Popper's Principles of Demarcation" in chapter 3); specifically, by restoring the traditional data management concepts. It is impossible, however, to restore the seamless integration we had before. So all we have in the end is some complicated and inefficient database systems that are struggling to emulate the simple, straightforward file systems.

# The Theory

## 1

To understand the relational delusions, we must start with a brief review of *formal logic* – that branch of mathematics upon which the relational model is said to be founded.

Formal logic is treated as a branch of mathematics because its exact principles and its deductive methods are similar to those of traditional mathematics. For this reason, it is also called *mathematical logic*. But, whereas algebra and calculus deal with numerical values, and geometry with lines and planes, logic deals with *truth values*: assertions that can be either *True* or *False*. As in other branches of mathematics, the elements and formulas of logic are expressed as variables – abstract entities that stand for a large number of particular instances. Thus, we normally use symbols like *x* and *y*, rather than actual assertions. This is why formal logic is also known as *symbolic logic*.

The oldest system of formal logic is *syllogistics*. Created by Aristotle in the fourth century BC, and further developed over time, syllogistic logic is based on propositions of the form "all *S* are *P*," "no *S* is *P*," "some *S* are *P*," and "some *S* are not *P*." (Examples: all fishes are swimmers, some buildings are tall, some people are not nice.) These propositions consist of two terms (the subject *S* and the predicate *P*) and a quantifier (all, some, none). The propositions assert, therefore, that a certain thing, or a class of things, possess a certain attribute. Additional flexibility is attained by permitting negative terms: "all *S* are *not-P*," "some *not-S* are *P*," and so on. A syllogism consists of three such propositions: two are premises, and the third one is the conclusion. The premises are related through one of their terms, and the conclusion uses the other two terms. (Example: Some *A* are *B*, all *A* are *C*, therefore some *C* are *B*.)

Clearly, many combinations of propositions are possible, but not all constitute valid syllogisms. A syllogism is valid if the conclusion follows by logical necessity from the two premises, as in the classic inference "All men are mortal, Socrates is a man, therefore Socrates is mortal." An example of invalid syllogisms is "Some dogs are vicious, this animal is not a dog, therefore this animal is not vicious" (even if the two premises are true, the conclusion can be either true or false).[1]

The study of syllogisms involves the classification of the various combinations of propositions, their logical relationships and transformations, and the methods for determining their validity. It should be obvious that, if we can

[1] In syllogisms, a reference to an individual entity is interpreted as a class of things that comprises only one element, and therefore implies the quantifier *all*.

reduce an argument to a structure of propositions consisting of subjects and predicates, syllogistic logic allows us to determine *formally* whether a particular statement can or cannot be inferred from certain premises. In other words, if we know that the two premises are true, we can determine whether the conclusion is true or false strictly from the *structure* of the three propositions; we don't have to concern ourselves with the *meaning* of the terms that make them up.

Syllogistic logic is seen today as only one of the many systems comprising the field of formal logic. Modern logic, born in the nineteenth century, attempts to extend beyond the capabilities of syllogistics the range of discourse and the types of phenomena that can be represented formally. The benefits of a formal representation are well known: as with traditional mathematics, it allows us to build increasingly large and complex entities that are guaranteed to be valid – simply by combining hierarchically, level after level, entities whose validity is already established. Conversely, if confronted with an expression too complex to understand directly, we can determine its validity by reducing it to simpler entities, one level at a time, until we reach entities known to be valid. Formal logic, thus, permits us to apply the deductive methods of mathematics to any type of phenomena.

We also know what are the *limitations* of formal logic. We can reduce a phenomenon to an exact representation only when its links to other phenomena are weak enough to be ignored. If we recall the concept of simple and complex structures, logic systems allow us to create only *simple* structures; so they are useful only for phenomena that can be studied in isolation. While common in the natural sciences, this is rare for phenomena involving human minds and societies. In chapter 4, for example, we saw the attempts made by scientists to represent *knowledge* by means of logic systems. These attempts fail because the entities that make up knowledge are connected in many ways, not just through the hierarchical relations recognized by a particular logic system. These entities can only be represented, therefore, with a *complex* structure. To this day, few scientists are ready to admit that most human phenomena *cannot* be reduced to an exact, formal model. So they keep inventing one mechanistic theory after another, hoping to represent mathematically such phenomena as intelligence, language, and software.

Although differing in complexity and versatility, the modern systems of logic have a lot in common. To create a system of logic, we start by defining its basic entities – those entities that act as starting elements in the hierarchical structures created with that system: objects, propositions, etc. Logical *variables*

(single letters, usually) are used to represent these entities in definitions and expressions. Next, we define a set of logical *operations* – the means of creating the elements of one level by combining those from the lower level. We also need some *rules of inference* – principles that justify the various transformations performed when moving from one level to the next. These rules serve, in effect, to restrict the use of operations to those cases where the new element can be derived from the others only through logical deduction. (For example, the rule known as *modus ponens* states that, if we know that whenever $p$ is true $q$ is also true, then if $p$ is found to be true we must conclude that $q$ is true.) Lastly, we agree on a number of *axioms*. Axioms are assumptions taken to be valid by convention, and which can be employed therefore in logical expressions just as we do premises. (A common axiom, for example, is the assertion that any entity is identical to the negation of its negation.) The theorems of a logic system are the various assertions that can be proved deductively within the system by manipulating expressions. Clearly, increasingly complex expressions and theorems can be constructed by combining elements hierarchically, on higher and higher levels of abstraction.[2]

Despite their formality, there is considerable freedom in designing a logic system. For example, what is a rule in one system may be an axiom in another, and what is a theorem may be a rule. What matters is only that the system be *consistent*. A system is consistent when no contradictions can arise between the expressions derived by means of its operations, rules, and axioms. That is, if we can show that a certain expression or theorem is true, we should not be able to show in the same system, through a different deduction, that it is false. Another quality found in a correct logic system is that of *independence*: every one of its axioms and rules is necessary, and none can be derived from the others. To put this differently, if any one of them were omitted, we would no longer be able to determine the truth or falsity of some expressions or theorems.

The chief difference between logic systems, then, is in their basic entities, and in the way these entities are combined to create correct expressions (what is known as *well-formed formulas*). And, once we reach a hierarchical level where expressions can only yield truth values, *True* or *False*, the same operations can be used to manipulate them in any logic system. The starting elements themselves may be entities restricted to truth values, but many systems have starting elements of other types. In syllogistic logic, we saw, the

[2] Note that, when used with the simple structures created with logic systems, the term "complexity" is employed here (as it always is when discussing simple structures) to indicate the shift to a higher level within a structure, or to a structure with more levels. (The levels of complexity in a simple structure are its levels of abstraction.) So don't confuse this complexity with the complexity of complex structures, which is due to the interaction of structures.

starting elements are subjects and predicates (things and attributes), and only their combinations are propositions that can be true or false.

The most common logical operations, thus, are those that manipulate truth values. And among them, the best known are conjunction, disjunction, and negation (AND, OR, and NOT). Only conjunction and negation are usually defined as basic operations, though, since disjunction can be expressed in terms of them: *A* OR *B* is equivalent to NOT(NOT *A* AND NOT *B*). Additional operations (equivalence, implication, etc.) can be similarly defined in terms of conjunction and negation, or by combining previously defined operations. A *truth function* is an expression involving operands that have truth values, so its result is also a truth value. This result can then act as operand in other expressions.

One way of determining the result of a truth function is with *truth tables*. A truth table has a column for each operand used by the function, and a row for each possible combination of truth values (hence, two rows for one operand, four for two operands, eight for three operands, etc.). A final column depicts the truth value of the result, and there may be additional columns for intermediate results. (Figure 4-2, p. 330, illustrates the concept of truth tables.)

Another characteristic common to all logic systems is that the validity of their low-level elements, and of their axioms and premises, cannot be determined from *within* the system. A logic system only guarantees that, if certain expressions are known to be true or false, then the truth or falsity of other expressions – derived from the original ones strictly through the rules and operations permitted by the system – can be determined with certainty. It cannot verify for us whether the expressions we *start* with are true or not.

For example, a premise like "*A* is larger than *B*" could be used with numbers in one application and with animals in another. In either case, it would be true in some instances and false in others. But within the logic system, this statement appears simply as a symbol, say, *S*; and it is handled the same way regardless of what *A* and *B* stand for, or whether the statement is false while believed to be true. It is our responsibility to ensure that it is true – by means *external* to the system – before using it as premise in a particular application.

Logic systems, then, are only concerned with the *form* and *structure* of elements and expressions, not with their *interpretation*. Needless to say, though, both aspects are important in actual applications. If all we want is that the conclusion be sound logically, its correct deduction from premises is indeed sufficient. But for the system to be of practical value, the deduction *and* the premises must be correct.

Thus, along with their limitation to simple, isolated phenomena, their dependence on what is usually just an *informal* verification of premises and starting elements reduces considerably the usefulness of formal logic systems

in real-world applications. The delusions of the relational database model, for instance, stem from overlooking the severity of these limitations, as we will soon see.

❖

The simplest system of logic is the one known as *propositional calculus*. The basic elements in this system are whole propositions, and expressions are formed by combining propositions through logical operations, as described earlier. Although expressions of any complexity can be formed in this manner, this system is handicapped by its inability to analyze the individual propositions. For example, if two propositions comprise subject and predicate, as in syllogisms, the system cannot distinguish between the case where the propositions share their subject or predicate, and the case where they are unrelated. The chief quality of propositional calculus is its simplicity, so it is the system of choice in applications where the elements can be treated as either atomic entities or logical expressions built from these entities. The Boolean logic system, upon which digital circuits and many software concepts are founded, is a type of propositional calculus.

A more versatile system of logic, and the one that served as inspiration for the relational database model, is *predicate calculus*. The basic elements in this system are subjects and predicates, as in syllogistic logic, but a predicate can be shared by several subjects in one proposition. A predicate, in other words, is seen as an attribute that can be possessed by one, two, three, or generally *n* different elements. And when possessed by more than one, it serves not only as attribute, but also to relate them. Each set of *n* elements related through a predicate is known as an *n-tuple* (or *tuple*, for short).

An expression like *P(x,y,z)* – which says that the elements *x*, *y*, and *z* are related and form a 3-tuple through the predicate *P* – is a basic proposition in predicate calculus. Since the elements are represented with variables, the expression stands for any number of such tuples. Each element has its own domain of permissible values, and when we substitute actual values for the three variables, the relationship will generally hold for some combinations of values but not for others. So the expression will be true for some tuples and false for others. The totality of tuples that share a particular predicate (or, usually, just those for which the expression is true) is called a *relation*.

An example of a relation is the sets of three integers, *a*, *b*, and *c*, each one selected perhaps from a different range of values, and fulfilling the condition that *a* is greater than *b* and *b* is greater than *c*. An expression like *G(a,b,c)*, representing this relationship, is then true for some sets of values and false for others. Another example is the sets of five men, *p*, *c*, *b*, *n*, and *g*, who could

have been selected from various domains to act as crews in WWII B-25 bombers: pilot, co-pilot, bombardier, navigator, and gunner. An expression like *B(p,c,b,n,g)*, representing this relationship, is true only for those sets of men that formed actual crews.

Basic propositions can be combined by means of logical operations, in the usual manner, to form more complex propositions. Thus, we can form relations that are a logical function of other relations. Take, for example, these two relations: *P(x,y)* as the sets of two elements, *x* and *y*, related through *P*; and *Q(y,z)* as the sets of two elements, *y* and *z*, related through *Q*. The expression *P(x,y)* AND *Q(y,z)* may then be defined to mean, depending on the application, either the sets of two elements common to *P* and *Q*, or the sets of three elements, *x*, *y*, and *z*, for which both relations hold. Similarly, the expression *P(x,y)* OR *Q(y,z)* may be defined to mean either the sets of two elements that exist in either relation (excluding duplicates), or the sets of three elements for which either relation holds.

Additional flexibility can be achieved in expressions by binding each variable with the *universal* quantifier ∀ (which says that the relation holds for *all* instances of that variable) or with the *existential* quantifier ∃ (which says that the relation holds at least for *some* instances of that variable). These quantifiers become then part of the expression, and participate in operations and transformations, much like operators. Thus, if ∀ is applied to both *x* and *y* in the expression *R(x,y)*, the expression is true only if the relation *R* holds for all possible pairs of values of *x* and *y*; but if ∃ is applied to *x* and *y*, the expression is true even if the relation holds for just one pair of values.

❖

This brief review will suffice for our purpose, to assess the mathematical merits of the relational database model. It is worth mentioning, though, that many other systems of logic have been designed. The system we have just examined, for example, is called *first-order* predicate calculus, and is only the simplest of the predicate calculi. (In higher-order systems, the quantified variables and the predicates can themselves be logical expressions.) Some logic systems include special axioms, rules, and operations to deal with such imprecise concepts as necessity, possibility, and contingency, which lie outside the scope of propositional and predicate calculi. Other systems attempt to deal with propositions whose truth value changes over time, and some systems even attempt to reduce to logic such moral issues as belief, obligation, and responsibility.

As I have already stated, the motivation for these systems is to bring phenomena involving minds and societies into the range of phenomena that

can be represented with formal, mechanistic methods. And they have had very little success, because few human phenomena are simple enough to be reduced to a mechanistic representation.

Programming phenomena are largely human phenomena. So the relational model is, ultimately, an example of the attempts to find a mechanistic model for phenomena that are, in fact, too complex to represent mechanistically. Thus, apart from our interest in the theories of software engineering as pseudosciences in their own right, their analysis complements our study of mechanistic delusions, and serves to remind us of the degradation of the idea of research. We saw in chapters 3 and 4 the childish attempts made by some of our most famous scientists to represent behaviour, intelligence, and language with diagrams, or formulas, or logic. And the same fallacy is committed with *software* theories: the mechanists discover a model that explains *isolated aspects* of a complex phenomenon, and they interpret this trivial success as evidence that their theory is valid, and hence worth pursuing.

So, by invoking the official definition of science – which is simply the pursuit of mechanistic ideas, whether useful or not – academics can now spend their entire career developing worthless theories. Merely because mechanism works in fields like physics or chemistry, they feel justified to seek mechanistic explanations in psychology, or sociology, or linguistics, or economics, or programming. Then, because of our mechanistic culture, we admire and respect them, and regard their activities as serious research – even as we see that their theories never work, and that they resort to deception in order to defend them.

## 2

Let us see now how predicate calculus was adapted for database work. The inventor of the relational model is E. F. Codd, who presented his ideas in a series of papers starting in 1969.[3] We are not concerned here with the evolution of the model in the first few years, or with the specific contributions made by individual researchers, but only with the relational database ideas in general. And, in fact, apart from a few refinements, the theory presented by Codd in his original papers depicts quite accurately what became in the end the formal relational model. In 1981, Codd received the prestigious Turing award for his invention.

[3] The first paper was published in 1969 (as an IBM document), but it was only the second one, published the following year, that was widely read: E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* 13, no. 6 (1970): 377–387.

Recall the organization of data files as records, and fields within records. To represent a file by means of predicate calculus, the fields are seen as the basic elements of a logic system, and the records as tuples – sets of *n* elements, where *n* is the number of fields in a record. Each field can possess a value from a domain of permissible values. Thus, if the field is a part number, the domain is all valid part numbers; if a vendor name, the names of all possible vendors; if a quantity, all numeric values that are valid quantities; and so on. Generally, some combinations of values exist as actual records in the file, and others do not; and, *by convention*, the relationship that links these fields holds only for those combinations that exist. In other words, an expression representing this relationship is deemed to yield the value *True* when the tuple actually exists in the file, and the value *False* otherwise. As in logic, the totality of tuples (i.e., records) in the file is called a relation.

To define an employee file with four fields, for instance, we would use a logical expression like *E(a,b,c,d)*, where *a* is the employee number, *b* the name, *c* the hourly rate, and *d* the number of hours worked. *E* stands then for the predicate that relates the four fields. *E* says, in effect, that each set of four values, taken from the respective domains of permissible values (all possible employee numbers, names, rates, and hours), are related in such a way that they represent a potential employee. The expression is true for the sets that actually exist in the file, and false for the others.

To this basic system, which matches the system of predicate calculus, a number of features were added in order to make the relational model suitable for database work. One feature is the idea of *field names*. In predicate calculus, the elements are identified by their relative position within the tuple, but this is impractical for database fields. Fields, therefore, are given names (QUANTITY, VENDOR-NO, INVOICE-DATE, etc.); we can then refer to them by their name, so their relative position within the physical record (as they are stored on disk, for example) is immaterial. These names are sometimes described as a special tuple that exists in every file but does not take part in operations; its function is similar to the top row in a typical table – the row that contains the column headers.

Another feature is the idea of a *key*: one field in the record is designated as key, and its value in each record must be unique within the file; alternatively, the key can consist of several fields, and then their combined values must be unique. The key, therefore, can be used to identify a specific record within the file, or to order the records in a logical sequence (so the actual sequence of records, as they are stored on disk or as they were added to the file, is irrelevant to the application). It is often useful to have several keys for the same record; in this case, one is designated as the *primary* key, and the others are called *candidate* keys. Lastly, in order to relate files, a field (or a group of fields) can

be designated as a *foreign* key. This type of key is used to identify the records of another file, where that field usually functions as the primary key. The customer number in an invoice file, for example, is a foreign key that relates the invoice file to the customer file, where the customer number is the primary key. The values stored in a foreign key need not be unique in each record; thus, we can have several invoices with the same customer number.

It should be obvious, if you recall our discussion of indexed data files and the basic file operations, that the relational concepts we have examined so far are *identical* to the traditional data file concepts. The only difference is in the use of terms like "relation" and "tuple" instead of the terms traditionally associated with data files. The relational theory is rich in new terminology. Thus, in addition to the concepts and terms taken from logic, we are told that files are best perceived as tables: the rows of these tables are then the records, and the columns are the fields. Also, the term *attribute* is often used for columns. So the accepted relational terms are *tables* and *relations*, *rows* and *tuples*, *columns* and *attributes*.[4]

While tables still resemble the traditional data files, the way we access them is entirely different. The traditional file operations are based on indexes, and are used through the flow-control constructs of a programming language. The relational operations, on the other hand, are defined in the manner of logical operations. In predicate calculus, we saw, operations like AND and OR take relations as operands and produce a new relation; similarly, the relational operations take tables as operands and produce a new table.

In predicate calculus, the tuples in the resulting relation consist of variables that were elements in the tuples of the original relations. When the same values that made up the tuples of the original relations are substituted for the variables of the new tuples, the expression that represents the new relation may be true for some combinations and false for others; and the new relation is defined as those tuples for which the expression is true.

Similarly, the operations in the relational model are defined in such a manner that the columns of the resulting table are selected from among the columns of the original tables. Then, depending on the operation and the values present in the rows of the original tables, only *some* of the rows are retained in the new table. In other words, each operation has its own definition

[4] Generally, *tables*, *rows*, and *columns* are considered informal terms, while *relations*, *tuples*, and *attributes* are the formal ones. Since the entities described by these terms are identical to the traditional *files*, *records*, and *fields*, I am using both the new and the traditional terms in the discussion of relational databases.

of truth and falsity, and if we represent the rows with a logical expression, the new table is defined as those rows containing combinations of values for which the expression is true. For example, if we represent a customer table as *C(a,b,c,d)* and an orders table as *O(a,e,f)* (where the lower-case letters stand for columns, and *a* is the customer number), a particular operation could be defined as follows: create a new table *R(a,b,e)*, whose rows are those pairs of rows from the customer and orders tables where *a* has the same value in both. The expression *R(a,b,e)* is said in this case to be true for these rows (i.e., where the customer matches the invoice) and false for the others. This expression – that is, the new table – can then be combined with others in further operations.

There are five basic operations: The UNION of tables *A* and *B* is a table containing the rows present in either *A* or *B* or both (*A* and *B* must have the same number of columns, and rows common to *A* and *B* appear only once in the new table). The DIFFERENCE of tables *A* and *B* is a table containing those rows present in *A* but not in *B* (*A* and *B* must have the same number of columns). SELECTION takes one table, *A*, and produces a new table containing only those rows from *A* for which an expression involving one of the columns is evaluated as true (for example, only those rows where the value in a given column is greater than zero). PROJECTION takes one table, *A*, and produces a new table containing all the rows from *A*, but only some of its columns. The PRODUCT of tables *A* and *B* is a table whose columns are the columns of *A* plus those of *B*, and whose rows are every combination of rows from *A* and *B*; each row, thus, is built by taking a row from *A* and extending it with a row from *B* (so, for example, if *A* has 10 rows and *B* has 20 rows, the new table will have 200 rows).

Additional operations may be necessary in practice, but they can always be expressed as a combination of the five basic ones. For example, to reduce a table to only some of its rows and columns, we perform first a SELECTION to retain the specified rows, and then a PROJECTION on the resulting table to retain the specified columns. (Note that the order in which we perform these two operations is immaterial.) Most database systems, in line with their promise to give us higher levels of abstraction, provide some of the most common combinations in the form of built-in operations.

PRODUCT, in particular, is rarely useful on its own, and is normally employed as just the first step in a series of operations. JOIN, for instance, consists of PRODUCT followed by SELECTION and then by PROJECTION. JOIN selects from all the combinations of rows in tables *A* and *B* those rows where a particular column in *A* stands in a certain relationship to a particular column in *B*. Most often, JOIN is used to combine two tables on the basis of *equality* of values. For example, the JOIN of a customer table and an invoice table based on the customer number (present in both) will result in a table that has the combined

customer and invoice columns, but (through SELECTION) only the rows where the customer number was the same in both tables. JOIN, thus, will match invoices and customers: it will have one row for each invoice, and each row will include the customer columns in addition to the invoice columns. (The PROJECTION in the last step serves to eliminate one of the two columns containing the customer number, since they are identical.) JOIN can be performed on key columns as well as non-key columns, and its chief use is to relate files.

For most operations and combinations of operations, we can understand intuitively how the resulting table is derived from the original ones. It is possible, though, to define these operations formally, as transformations based on the operations of formal logic. There are several ways to do it: the relational *algebra* describes them as operations on tables, as we just saw; the relational *calculus* – of which there are two versions, *tuple* calculus and *domain* calculus – describes them with logical expressions similar to those used in predicate calculus. The operations are the same; only the way they are described differs.

The value of the relational model is due to expressing the data in the resulting table as a logical expression of the data in the original tables. Thus, if the original data is correct, the final data will also be correct. When permitted to combine records and fields at will – with the traditional file operations, for instance – a programmer may make mistakes and generate files that do not reflect correctly the original data, or generate files containing inconsistent data. This cannot happen with a relational database. Because we are restricted to operations on whole tables, and because the relational operations introduce no spurious dependencies between fields, we can be certain that the final table will express the same data and relationships as the tables we start with. In a database query, for instance, no matter how many operations and combinations of tables are involved, the final entities are guaranteed to be the same as the original ones – only arranged differently. It is impossible, in fact, to generate wrong or inconsistent data if we restrict ourselves to the relational operations.

The relational model permits us to implement any database requirements that we are likely to encounter in applications. The restriction to whole tables is not really a handicap, because any portion of a table – any subset of rows and columns – is itself, in effect, a table. We can even isolate a single row (with appropriate SELECTIONs), and that row is treated as a table and can be used in further operations. Even one column of that row can be isolated (with a PROJECTION), and that single element still is, as far as the relational operations are concerned, a table.

Note that the resulting table need not be a real entity. When using SELECTION to answer a query, for example, the database system may simply *display* the

selected rows, without actually creating a table. Generally, to perform a series of operations, the system may create some intermediate tables or use only the original ones, and may employ indexes or other expedients. But we don't have to concern ourselves with these details, because a good system will automatically discover the most effective alternative. All we need to do is specify, through the relational algebra or calculus, the original tables and the desired operations.

What we gain with the restriction to tables, then, is simplicity and accuracy: all we need now is a few operations, which are founded on formal logic and can be safely combined into more complex ones. Just as importantly, these operations permit us to view the data from a higher level of abstraction: we no longer need to access individual records, as in the traditional file systems; nor do we need file scanning loops, or intricate conditions to select records and to relate files. Whether our requirements involve single tables, or combinations of tables, or portions of tables, or just one row or column, all we need now is the high-level relational operations. Thus, a relational database is said to be "tables and nothing but tables."

## 3

It is not enough for the database *operations* to conform to a logic system. The relational theory also requires that the data be *stored* in a logical format. Specifically, the fields that make up the individual tuples must be simple, indivisible entities, with no unnecessary dependency between them. Tables that adhere to this format are said to be *normalized*, and the process of bringing them to this format is called *normalization*. There are several levels of normalization, each one a more stringent enforcement of these principles. The levels are known as first normal form, second normal form, third normal form, etc., and are abbreviated as 1NF, 2NF, 3NF, etc.

The fundamental requirement is 1NF. For a table to be in first normal form, each field must be a simple entity – a single, atomic value. In traditional data files, a field may consist of a series of values, or a multidimensional array of values, or a hierarchical structure of values. A twelve-month transaction history, for example, can be stored in one field of a customer record as an array of twelve rows by three columns – month, quantity, and amount. The relational model prohibits this format: data that comprises a set of related values must be stored in a separate table, where each value has its own column. Thus, to reduce the customer table just described to 1NF, we must create a separate table for the transaction history. The columns in this table will be the customer number, month, quantity, and amount, and the key will be the combination of customer

number and month. For each row in the customer table there will be twelve rows in the transaction history table.

It should be obvious why the first normal form is so important. The relational operations expect to find tuples, and cannot process multiple values – data stored, in effect, as tuples within tuples. To deal with this format we need a more complex database model, and operations that can process more than just rows and columns.

The other normal forms deal with the problem of *field dependency*; specifically, the dependency of one field on another within a tuple. Since such a relationship is likely to cause data *redundancy* and *inconsistencies*, the only type of relationship permitted between fields within a tuple is the obvious dependency of the tuple's fields on the tuple's unique key. All other field relationships must be implemented by moving the fields to other tables and linking the tables logically.

An example of misplaced dependency is a customer orders table where the key is the combination of customer number and order number, and the other fields are the customer name and address, and the order date, quantity, and amount. All these fields depend on the customer number; but, whereas order-specific data like quantity and amount must indeed be included in each order, fixed customer data like name and address must not. This design is wrong because, while a customer's name and address are the same for all his orders, we *repeat* them in every order. The faulty design, thus, will cause data redundancy. Worse still, it will cause various inconsistencies ("anomalies," in relational terminology) when we run the application: first, if a customer's name or address changes, we may have to update not one but several rows – all his outstanding orders; second, we can store a customer's name and address in the database only if that customer has at least one outstanding order.

The solution, of course, is to store the name and address in a separate table, where the key is the customer number and there is only one row per customer. From the order rows we can then access the appropriate name and address by using the customer number as link. The process of normalization, thus, consists in creating two tables from one. In general, we eliminate a misplaced dependency by increasing the number of tables: we extract the fields with repeated values and place them in a separate table, where we discard the duplicate rows; then we choose a field (or a combination of fields) with unique values to act as a key for the new table and as a link to the original one.

The redundancy and inconsistencies were caused, obviously, by an incorrect design – a design that did not match the application's requirements: the name and address are the same for all orders, and yet we repeated them in every order. With the fields in a separate table, the design matches the requirements, and consequently there is no redundancy or inconsistency.

The normalization theory, however, describes the problem of incorrect design as a problem of misplaced dependency: the name and address depended on only a portion of the key (the customer number), instead of depending on the whole key (the combination of customer and order numbers), as do the order date, quantity, and amount. And we correct this dependency by placing the name and address in a separate table – a table where the customer number is the whole key. Clearly, what we do is the same as before, match the design to the requirements; but the normalization theory describes this process as the elimination of misplaced dependencies.

Few people would design a database with the kind of redundancy we have just examined. The theorists, nevertheless, treat the subject of normalization very seriously. Various types of field dependencies are defined and studied in great detail, along with the steps required to eliminate them. Thus, five types were discovered, each one more rare and more subtle. Tables, we saw, are already in first normal form when their fields are single elements. Then, after eliminating one type of dependency, they are in second normal form (2NF). After eliminating a second type, they are in third normal form (3NF). This is followed by a level known as Boyce/Codd normal form (BCNF), and then by the fourth and fifth normal forms (4NF and 5NF). When in fifth normal form, tables are in the ultimate relational format, devoid of any misplaced dependencies. Very few databases, however, require all five levels of normalization. If an application is not complicated, tables will likely be in their highest possible normal form after just one or two levels, simply because there are no other dependencies. In any case, many experts consider 3NF or BCNF adequate, and don't even mention 4NF and 5NF.

The second and higher normal forms are in reality very similar, and their differences need not concern us here. The reason for having several types of normalization and a numbering system is largely historical: while the first normal form was described by Codd in his original papers, the others were incorporated into the relational theory later – as they were discovered, one by one. (More specifically, the higher normal forms became necessary when the relational model was expanded to include *updating* operations.) Thus, it is worth noting that the theorists needed several years, and innumerable papers and conferences, to discover what an experienced programmer could have told them from the beginning. For, the problems caused by misplaced dependencies, as well as their solutions, are *identical* in relational databases and in databases created with traditional data files; only the attempt to treat these problems formally is new. We will return to this subject later, in our discussion of the relational delusions.

# The Contradictions

## 1

To summarize, the relational model is an attempt to turn database programming, as well as database use, into an exact, formal activity. Since data records resemble the so-called tuples of predicate calculus, and since they can be manipulated with operations resembling logical operations, the theorists concluded that the rigour and exactness of mathematical logic can now be attained in database work. All we have to do is restrict the files and records to a certain format, and restrict the operations to a high level of abstraction; we have then a mathematical guarantee that the answers to queries will reflect accurately the data and relationships present in the database.

From the start, then, the relational theory was grounded on the curious principle that only *some* aspects of the database need to be covered by the formal, mathematical model; the others can remain informal. This principle is sometimes expressed with the statement that certain aspects lie *within* the scope of the formal model, while others are *outside* its scope. Thus, if we call "relational model" the whole body of relational principles and features, the "formal relational model" constitutes only a small part of it.

Within the scope of the formal model lie, as we just saw, the format of files and records, the concept of queries, and the use of high-level query operations. The theorists recognize, of course, that there is a lot more to databases and applications. So, while asking us to treat these aspects formally, they expect us to deal with the other aspects of the database *informally*: by relying on traditional programming methods and on personal skills.

In particular, operations that *update* the database – adding and deleting records, modifying the data in fields, creating and deleting files – cannot be treated formally, and therefore lie outside the scope of the formal model. Note that this is a necessary consequence of the model's mathematical foundation: predicate calculus is concerned with the logical expressions that *use* the tuples of a given relation, not with the way the tuples became part of that relation, or with the way the elements of these tuples acquired their current value. Thus, if the updating of tuples and relations lies outside the scope of predicate calculus, it must also be left out of the formal relational model.

Data normalization, too, is largely informal. Only the first normal form, which deals with the record format, is part of the formal model. The second and higher normal forms are only needed in order to prevent redundancy and inconsistencies when *updating* the files; thus, if the updating operations are informal, so must be the normalization. In any case, the process of normalization entails an *interpretation* of the application's requirements:

whether or not a certain field depends on another can be determined only from the way we intend to use them in the application, something that no formal system can know.

Another aspect of the database that cannot be formalized concerns *data integrity* – the countless rules that ensure the validity of the updating operations within the context of a particular application. Again, what is valid in one case may be invalid in another, and only *we* can decide how to interpret the result of a certain operation.

Lastly, the formal model does not include the means we use to *specify* the query and updating operations. These means – a set of commands, or a database language – can only be used informally. As is the case with any programming language, we can define with precision the commands or statements themselves, but not their effect when combined to perform a particular task in a given application.

In conclusion, the updating operations, the normalization process, the integrity rules, and the database language, even though needed in any application that uses relational databases, lie outside the scope of the formal relational model. So they can be no more formal or exact than they are in applications using traditional data files.

What, then, is the meaning of the relational model? What is the point, for instance, of including in the formal model the query operations while excluding the updating operations? Clearly, the two types of operations are equally important in an application. What is the value of a model that guarantees correct answers to queries while being unable to guarantee the correctness of the data upon which the queries are based?

The theorists acknowledge that the formal model is insufficient, that we must depend on some informal operations too, but they fail to appreciate the implication: if we must deal with certain aspects of the database by relying largely on personal knowledge, the inexactness of this method will annul the exactness of those aspects treated formally. The result of a process cannot be more exact than the least exact of its parts. The answer to a query may well be mathematically derivable from the original data, but this quality of the relational model has little value if we cannot prove that the original data is correct to begin with.

❖

We just saw how, in order to attain a practical relational model, the theorists were compelled to separate it into a formal and an informal part. But this is not all. There is one aspect of the database that is considered to lie, not only outside the scope of the *formal* model, but outside the scope of the relational

model altogether: the actual, physical implementation of the database and operations.

Like the logic system that inspired it, the relational model is a mathematical, and hence abstract, concept. This limitation, however, is interpreted by the theorists as a *quality*: thanks to its abstract nature, they say, we no longer need to be concerned with such issues as the system's *performance* (the time required to execute the database operations). In general, the independence of the logical database structures from their physical implementation permits us to access the data from a higher level of abstraction. Here are some statements expressing this view: "The ideas of the relational model apply at the external and conceptual levels of the system, not the internal level. To put this another way, the relational model represents a database system at a level of abstraction that is somewhat removed from the details of the underlying machine."[1] "The eight relational operators express functionality without concern for (or knowledge of) technical implementation. An obvious benefit is that relational users apply relational operators without concern for storage and access techniques."[2] "The aim of the relational model is to represent logically all relationships, and hence alleviate the user from physical implementation details."[3] "The relational data model removes the details of storage structure and access strategy from the user interface."[4]

The operations of a mathematical system are assumed to occur instantaneously; we don't think of addition or multiplication, for instance, as physical processes that may take some time. Similarly, the high-level operations of the relational model – selection, projection, join, and the rest – are assumed to be executed instantaneously by the database system. Incredibly, while presenting the relational model as the foundation of practical database systems, the theorists insisted that the subject of performance lies outside the scope of the model. Everyone knew, of course, that the application's performance is limited by the speed of the computer's processor, and that databases rely on physical devices like disk drives, which impose additional speed limits on data access. Nevertheless, the claim that it is possible to design real databases without having to concern ourselves with their performance was received with enthusiasm. All we need to do, promised the theorists, is specify the relational

[1] C. J. Date, *An Introduction to Database Systems*, 6th ed. (Reading, MA: Addison-Wesley, 1995), p. 98.

[2] Candace C. Fleming and Barbara von Halle, *Handbook of Relational Database Design* (Reading, MA: Addison-Wesley, 1989), p. 38.

[3] M. Papazoglou and W. Valder, *Relational Database Management: A Systems Programming Approach* (Hemel Hempstead, UK: Prentice Hall, 1989), p. 30.

[4] Ken S. Brathwaite, *Relational Databases: Concepts, Design, and Administration* (New York: McGraw-Hill, 1991), p. 26.

operations, just as we do in mathematics. The database system will analyze the request, determine the most efficient implementation, and then execute the necessary low-level operations.

Separating the performance issue from the relational model is just as illogical as separating the updating operations from the query operations. It shouldn't come as a surprise, therefore, that the relational database systems have proved to be incurably slow, and that, in addition, their users have remained as preoccupied with the performance issue as those who use the traditional file operations.

It is absurd to expect the database system itself to know what is the most efficient implementation of a high-level request. It is absurd because most requests do not depend on database structures alone, but also on such other structures (i.e., aspects) of the application as its various processes (see pp. 345–346). To discover the most efficient implementation we must link, therefore, the database structures with the other structures that make up the application. These links, moreover, occur usually at the low level of database fields, memory variables, and individual statements; so the only way to implement them is through traditional programming means.

The inefficiency caused by the lack of low-level links, then, was the main reason for the continued preoccupation with the performance issue. And this inefficiency was also the reason for annulling, in the end, two fundamental relational principles: the restriction to normalized files, and the restriction to high-level operations.

❖

We know, of course, why the theorists separated the relational model into formal and informal aspects: because this is the only way to attain a precise, mechanistic representation of the database and the database operations. If we want to represent an indeterministic phenomenon with a deterministic theory, we must exclude from the phenomenon those aspects that prevent such a representation.

Thus, we can start with any phenomenon, no matter how complex, and invent an exact theory – a mathematical model – that depicts what we *wish* the phenomenon to be. Then, we match the phenomenon to the theory by eliminating, one by one, those aspects that contradict the theory – by branding them as "informal" parts of the phenomenon. If we eliminate enough aspects, we are certain to reduce the phenomenon eventually to a version that is simple enough to match the theory.

But this is a trivial accomplishment – we knew all along that it could be done. It is impossible, in fact, to *fail* in this project, if we place no limit on the

number of aspects that we are willing to eliminate. Mechanistic projects of this nature are, therefore, intrinsically pseudoscientific. This is true because the concept of separating the phenomenon into aspects that are, and aspects that are not, within the scope of the model is unfalsifiable: since we are free at any moment to exclude any number of additional aspects in order to make the model work, there is no condition under which we can say that a mechanistic model *cannot* be found.

The issue, then, is not whether we can find a mathematical model for the phenomenon of a database, as this is always possible by simplifying the phenomenon. Rather, the issue is whether, by the time we simplify the phenomenon sufficiently to have an exact model, such a model is still meaningful. It is quite easy to discover mathematical models for *individual aspects* of software phenomena. The theorists happened to discover a database model grounded on predicate calculus, but with a little imagination we could find any number of other models. The real challenge, again, is not to discover a mathematical model by simplifying the phenomenon, but to discover a useful model for the original, complex phenomenon.

So, like all mechanistic delusions, the relational model failed because its mathematical foundation is insignificant: we can represent mathematically only a small fraction of the concepts involved in programming and using a database. If we were to rely on the original relational model, we would perhaps enjoy the promised benefits, but only with small and simple databases; we would be unable to develop the kind of databases we need in real applications.

Having failed as a practical concept, the relational model was rescued by expanding its *informal* aspects – precisely those aspects that had been excluded from the formal, mathematical model. The early works discuss in detail the formal model, including the various types of query operations, but mention only briefly the informal aspects – the updating operations, the integrity and performance problems, and the database language.[5] These aspects are presented merely as miscellaneous features needed to support the formal model in actual applications. In the end, however, it is precisely these features (the database language, in particular) that became the main concern of relational database systems, while the formal model declined in importance and became practically irrelevant.

Specifically, the restriction to normalized files and the restriction to high-level operations were both lifted; and no one, of course, is using databases through mathematical logic. Today's relational systems are promoted by praising the power of their programming language (usually SQL), the power of

[5] See, for example, Codd's original paper, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* 13, no. 6 (1970): 377–387. The other informal aspect (the second and higher normal forms) is not mentioned at all.

certain features described as integrity functions (but whose true role is to bypass the limitations of the high-level operations), and the power of a variety of new data formats and *low-level* file operations. In other words, while the power of the original model was said to derive from its formal, mathematical foundation, the power of what is seen today as the relational model derives entirely from *informal* concepts – concepts that are practically identical to the traditional ones. We will analyze this degradation in "The Third Delusion."

## 2

When studying the relational model and its evolution from a mechanistic fantasy to a pseudoscience, we can distinguish three major delusions. These delusions are summarized below; then, in the following subsections, we will study them in detail.

The first delusion is the belief that the relational model's mathematical background is an important quality. It is true that the model is grounded upon certain mathematical principles, and that these principles guarantee the soundness of certain database operations. But this quality constitutes an insignificant part of the phenomenon of a database: we can ground a database on mathematics only after limiting the data to a certain format, after separating the database from the rest of the application, and after restricting its use to queries expressed through high-level operations. Important aspects – the operations that modify the database, and the links to the rest of the application's logic – are not included in the mathematical model. Thus, we must deal with the most difficult aspects of database programming *informally*, just as we do when using the traditional file operations.

The second delusion is the belief that the principles of normalization are an essential part of the relational theory. In reality, data normalization is a totally useless concept, even *within* the relational model. It is a contrived theory that attempts to eliminate data redundancy and inconsistencies by identifying misplaced field dependencies. But misplaced dependencies occur only in an incorrectly designed database. So, in order to justify the need for normalization, the theorists ignore the application's requirements and deliberately create an incorrect database; then, they use normalization to convert it into a correct one. The theorists also delude themselves when claiming that they have turned database design into a formal, exact procedure. All they do, in fact, is discuss formally their invention, the various types of field dependencies; the design problem itself has remained as informal as before.

The third delusion emerged when the relational model was found to be impractical. In order to reduce them to a mathematical representation,

the database format, relationships, and operations were simplified so much that very few actual requirements could be implemented. Consequently, the theorists were compelled to "enhance" the model. And this consisted in restoring, one by one, those features that had been eliminated in the first two delusions in order to attain the exact, formal representation. By the time the model was versatile enough to be practical, there was nothing left of the preciseness of the original theory, not even in that narrow domain where the database operations had indeed been mathematical. The third delusion, thus, is in the belief that the original restrictions are not really necessary; in other words, the belief that we can enjoy the benefits of an exact theory without having to adhere to its principles.

Historically, the first two delusions can be said to make up the original idea, while the third one emerged when trying to implement that idea. The first two are a manifestation of the mechanistic fallacies; that is, attempting to represent a complex phenomenon with simple structures. And the third one is the consequence of this attempt. Since the only way to save a fallacious theory from refutation is by making it a pseudoscience, the software experts rescued the relational model by turning its falsifications into what they describe as new relational features. But what they are doing is merely to restore those features which they had eliminated previously in order to make the theory mechanistic. So, if the first two delusions demonstrate the naivety of the software experts, the third one demonstrates their dishonesty: they continue to praise the benefits of the relational model even while annulling the relational concepts and replacing them with the traditional ones.

The relational database theory is an excellent example of what I have called *the new pseudosciences*. Even better than structured programming or object-oriented programming, it can serve as a model of the modern mechanistic delusions.

Recall how these delusions evolve. The scientists start by noticing *one* aspect of a complex phenomenon; so they extract, from the *system* of structures that make up the phenomenon, the structure depicting that one aspect. Then, they enthusiastically announce a formal, mathematical theory based on this structure alone – claiming, in effect, that a complex structure can be reduced to a simple one. A further benefit of having only one structure, they say, is that we can choose our starting elements from higher levels of abstraction – an expedient that makes it even easier to represent the phenomenon.

But the theory does not represent the phenomenon accurately enough to be useful. So, instead of trying to understand the reason for its failure, the

scientists decide to "improve" it: they suppress the falsifications by reinstating, in the guise of new features, the very features they had previously excluded – features that must indeed be excluded if we seek a mechanistic theory. The purpose of the new features, thus, is to restore some of the original structures, and the links between them. In the end, the theory becomes useful only when enough of the old features are reinstated to allow us to represent the entire complex phenomenon again; that is, when we are allowed to represent it *informally*, the way we always did. The scientists, though, continue to praise the exact, mechanistic qualities of their theory – even as everyone can see that what made the theory useful is the *annulment* of these qualities, and their replacement with complex, indeterministic ones.

## The First Delusion

### 1

The first delusion is the belief in the mathematical merits of the relational model. For more than thirty years, we have been hearing the claim that the relational model is based on mathematical logic, and therefore relational databases benefit from the rigour and precision of mathematics. Although few people actually understand the connection between databases and mathematics, no one doubts this claim. After all, the mathematical benefits are being praised, not just by the vendors of relational systems, but also by university professors, database experts, and professional computer associations. In this subsection I want to show, however, that the claim is a fraud: relational databases do not benefit at all from mathematical logic.

❖

Mathematical systems, which include logic systems, are artificial models invented by us in order to represent with precision various aspects of the world. It is not difficult to invent a mathematical system (see pp. 694–695). Essentially, we define its basic elements, the operations that combine elements from one level of complexity to the next, the rules that control the use of these operations, and the axioms (those assumptions taken by convention to be valid assertions). We can then build increasingly complex expressions and theorems by combining elements on higher and higher levels. For the system to be useful mathematically, it must be consistent: no contradictions should be possible between the expressions or the theorems derived within the system. The basic elements vary with the system: numerical values for the

classical mathematical systems, or subjects, predicates, and propositions for the logic systems.

The more elaborate the system, the more complex its elements and operations. Differential calculus, for example, is more complex than arithmetic or algebra. Since mathematical systems are simple hierarchical structures, a higher complexity means only that the system can have more levels, and more intricate elements at the higher levels, within *one* structure. While still a simple structure, though, a more elaborate system allows us to represent more difficult phenomena.

To use a mathematical system as model, we start by translating the entities and processes that constitute the phenomenon into the entities and operations permitted by the system. Once this is accomplished, we can study the phenomenon by working strictly with the mathematical concepts. We create expressions and higher-level elements, manipulate them in various ways, and finally translate the results back into real entities and processes. With this method, we can explain and predict events that may be difficult or impossible to study directly.

A classic example of a mathematical model is Newton's theory of gravitation: if we represent with mathematical entities and operations the bodies that make up the solar system, their state at a given instant, and the natural laws that govern their motion, we can determine with accuracy their position at any other instant in the past or in the future. Clearly, it would be impossible to accomplish this without a mathematical model.

Imagine now a trivial system, a small subset of traditional mathematics: the basic elements in this system are integers, and the only operations are addition and subtraction. Thus, since expressions are limited to these two operations, the most complex elements possible are still integers. And, even though the system permits any number of levels and hence increasingly complex expressions, because of its simplicity it is unlikely that we will ever need more than a few levels. Nevertheless, while simple, this system is not without practical applications; we can employ it, for example, to create accounting models (if we agree to use only whole dollars). The chief difference between it and the mathematical systems of science and engineering is that the latter reach much higher levels, and much more complex elements and operations.[1]

Turning now to the relational model, we find a modification of the logic system known as predicate calculus. To this system, features like record keys and field names were added in order to adapt it for database work. The simplest

[1] It is worth stressing again that the term "complexity," when used with the simple structures of mathematical and logic systems, refers to *levels of complexity* (also known as levels of abstraction), and it must not be confused with the complexity of complex structures (which is due to the interaction of several simple structures).

elements in the relational model are the fields – called now columns, or attributes. Fields are combined to form records – called now rows, or tuples; and records are combined to form files – called now tables, or relations. Relations can then be combined into expressions by means of standard logical operations (AND, OR, and NOT) and some new, more complex operations (UNION, DIFFERENCE, SELECTION, PROJECTION, and PRODUCT). Relations can be combined in this manner to form increasingly high levels, but the result is still a relation. Relations, thus, are the most complex elements in a relational system. Although more intricate than our system of integers and two operations, it is still very simple – far simpler than the mathematical systems employed in science and engineering.

And herein lies the explanation for the first delusion, why the mathematical background of the relational model is irrelevant. It is true that the relational entities and operations can be defined rigorously, with the same methods and notation we use in mathematics. But this preciseness is specious. The relational definitions resemble perhaps the definitions found in the traditional mathematical systems, but, because the relational model is such a simple system, its formality is superfluous, even silly.

The operations of a mathematical system, and the rules that govern the use of these operations, determine how the elements that make up one level are combined to form the elements of the next level. These combinations become the theorems and expressions possible in the system, and, ultimately, the mathematical representation of a phenomenon when the system is used as model. A formal definition of entities, operations, and rules is important in the *traditional* systems, therefore, because this formality is our only guarantee that the theorems and expressions remain valid as we move to higher levels of complexity. But if in a relational system all we have is some simple elements and operations – some simple transformations of one file into another, or of two files into one – and if we rarely need more than a few levels, the formality is hardly necessary. The concept of files, records, and fields is so simple that we can accomplish the same tasks using nothing more than personal skills.



Let us divide the use of a mathematical system into two parts, *translation* and *manipulation*. The translation is the work required to convert into a mathematical representation the entities and processes that make up the phenomenon, and to convert the mathematical entities back into real entities and processes. The manipulation is the work performed *within* the system, with the mathematical entities alone.

When praising the power of mathematics, it is the manipulation that we

have in mind, not the translation. The translation – an effort to represent a complex world with a neat, artificial system – is necessarily informal, and cannot benefit from the exactness of the mathematical system itself. Thus, there is no way to guarantee that we selected the right system and operations to model a particular phenomenon, or represented the phenomenon accurately, or interpreted the results correctly. All we have to guide us in the translation is our skills.

To take a simple example, we can use mathematics to model the relationship between the speed of a car and the distance traveled in a period of time. All we need to do is represent these entities with appropriate values, perform the operation of multiplication or division, and then translate the result back into an actual entity. So, if we know the car's speed, the mathematical model lets us predict the distance it will travel in a given period of time, or the time required to travel a given distance. But mathematics cannot verify for us that we employ the formulas correctly. It cannot stop us, for instance, from using an incorrect speed, or from measuring the speed in miles per hour and the distance in kilometers. We praise the power of mathematics to predict the distance or time, but, in reality, mathematics only guarantees that the higher-level element (the result) is indeed the product or quotient of the two elements we started with.

No mathematical system can also be a substitute for the expertise required to use it. Although no less important than the system itself, the work involved in using it – particularly the translation from real entities into mathematical ones and back – is largely informal, and hence open to errors despite the exactness of the system.

In relational systems, we saw earlier, this problem led to the separation between the formal and the informal aspects – those aspects deemed to be within, and those deemed to be outside, the scope of the formal model. The formal aspects, we see now, correspond to the manipulation, while the informal ones form the translation. The manipulation includes the definition of fields and tuples, and the query operations. And the translation includes everything else: the updating operations, the normalization, the integrity rules, and the database language. This separation is artificial, of course, since all aspects of the database are equally important. But it is inevitable if we want to have a mathematical model: if we must exclude from the model any database aspect that is too complex to treat formally, that aspect is bound to end up as part of the translation, where we can deal with it informally.

This limitation – the need to treat the translation informally – is inherent in all mathematical systems. No matter how rigorous and exact is the manipulation, we depend largely on personal skills when selecting a particular system for a given phenomenon, and when translating the real entities into

mathematical ones. And the relational model is no different. What makes it silly, then, is not this limitation, which is universal, but the fact that it consists almost entirely of the informal translation. In the end, what is for the traditional mathematical systems the ultimate purpose – the formal, exact manipulation – plays in relational systems an insignificant part.

## 2

Recall the predicate calculus system, the logical foundation of the relational model. A logical expression like *P(x,y,z)* describes the tuples of elements *x*, *y*, and *z* related through predicate *P*. When we substitute actual values for the three elements, the expression will be evaluated as *True* for some tuples and *False* for others. We retain then, usually, the set of tuples for which the expression is true (a relation); and, using logical operations, we combine it with other sets, which are based on different predicates and elements. Such a combination is an expression that describes a new set of tuples – a set that relates in a different and perhaps more complex way the elements of the original tuples. The new set may then be combined with others, and so on, to create higher levels of complexity.

Like all logic systems, predicate calculus is concerned with the *structure* of variables and expressions, not their meaning. All it can guarantee is that, if we restrict ourselves to combinations based on the operations permitted by the system, the result of each combination will be correct within the definition of the system. Thus, if we know that the tuples in the original sets are true, we can determine with certainty, level after level, whether those in the resulting sets are true or false.[2] The system guarantees the validity of the manipulation, but the translation remains our responsibility: it is up to us to determine – by means external to the system – whether the tuples we start with are true or false, and whether the expressions that define the tuples, along with the operations that combine them from one level to the next, match the relations between the entities we want to model in that system. A logic system, in the end, is only a tool. It is up to us to judge whether it is the right tool in a given situation, and to use it correctly.

The fallacy, thus, lies in the belief that if the database model resembles a logic system, database work will become an exact, mathematical activity. In reality, this new tool is inappropriate for database programming, because the

[2] Strictly speaking, it is not the tuples that are true or false, but the result of the logical expression that defines the tuples; the more accurate description, though, would make these sentences too complicated.

database structures are closely linked to the other structures that make up the application. Mathematical and logic systems can only model phenomena that can be represented with a simple structure. So their ability to model database structures has little value if these structures must interact with others.

Like the predicate calculus system, a relational system guarantees only that the sets of tuples generated from the previous ones, as we move from one level to the next, are correct within the system's definition. The elements are now fields, the tuples are records, the sets of tuples are files, and the logical expressions depict combinations of files or portions of files. The database and the relational operations can be represented, therefore, with the same formality and preciseness we enjoy in predicate calculus. An expression like *F(a,b,c)* defines a file by stating that the fields *a*, *b*, and *c* are related through the predicate *F*. When actual values are stored in these fields, the expression will be true for some tuples (i.e., records) and false for others; and the actual file consists of those tuples that are true.

In other words, we use simply the *existence* of the tuple to determine the truth or falsity of the expression that defines the tuple: a tuple is deemed "true" if it currently exists as a record in the file, and "false" if it does not. Thus, the definition of truth and falsity in a relational database is merely a *convention*. The convention states in effect that *the data in the original files is valid by default*, simply because the records present in the file are deemed to be "true." The absurdity of this convention is the root of the relational model's mathematical delusion, as we will see in a moment.

The relational operations are designed to create a new set of tuples from one or two existing sets; that is, to create a new file by selecting and combining records (or portions of records) from one or two existing files. So, if we restrict ourselves to the relational operations, the validity of the existing data guarantees the validity of the combinations: since the original records are true by default, the records in the new file will also be true. The new file can then be combined with others to create a higher level of complexity, and so on. No matter how we use the relational operations, we can be sure that the final result will reflect the data we started with. There can be no false records in the resulting files, because there were no false records in the original ones.

It should now be obvious why logic systems are inappropriate as database models. A logic system like predicate calculus cannot control the addition and deletion of tuples in the original sets, nor the modification of their elements. All it can do is create new sets of tuples from existing ones; that is, *read* the original data. And if predicate calculus is limited to reading its data, so must

be the relational model. With a database, the limitation to reading is, of course, the limitation to queries. So the fact that the relational model is limited to queries follows *necessarily* from its logical grounding. In general-purpose applications, though, adding, deleting, and modifying records are as important as queries. It is absurd, therefore, to ground a database system on predicate calculus.

The great weakness of the relational model, then, is the need to ensure by *informal* means that the original tuples are true. Since it is the truth or falsity of each tuple that determines whether it can be a record in the file, what is merely the truth value of a logical expression in predicate calculus becomes the critical issue of data integrity in a relational system. This means that, before we add a new record to a file, we must ensure somehow that the record is true; similarly, when we modify the data in an existing record, we must ensure that it continues to be true; and before we delete a record, we must ensure that it is indeed false.

But the only way to perform these validity checks is by accessing the tuples from *outside* the relational model, with *traditional* programming methods. The software theorists underrate the significance of this weakness: they casually say that the database modifications and the associated integrity issue lie outside the scope of the formal model, as if this limitation were just a minor implementation detail.

Clearly, what the model offers us – the assurance that the files resulting from relational operations are correct if the original ones are – is meaningful only if we can be sure that the data in the database is correct at all times. But the values stored in database fields are not right or wrong in an absolute sense. Their validity can only be assessed within the context of the running application; that is, by performing certain operations that link the database structures with some of the other structures that make up the application. The database is one aspect of a complex structure, and the validation process cannot be represented with a formal, mechanistic model.

So, if the validation is an informal, error-prone process, how can anyone claim that the relational model guarantees the correctness of the resulting files? All it can guarantee is that the data in the resulting files reflects accurately the data in the original ones. Consequently, the validity of the resulting data can be no more certain than the validity of the original data. And ascertaining *that* validity is no different in a relational system than it is for traditional data files. Thus, since the ultimate precision of a system is limited by its least precise part, the belief that a relational database is more precise than a traditional one is a delusion.

Here are some of the mistakes that can be committed in applications based on a relational system – mistakes that would not be detected by the system:

adding a new record that has invalid values in fields like address, phone number, price, or part description; placing invalid values in the fields of an existing record; retrieving a part record using the vendor number as part number (record keys for parts, vendors, employees, customers, etc., may well share a common range of values, so this mistake would not result in an invalid key, and the wrong record would indeed be retrieved); adding the quantity purchased to the quantity in stock of a part, instead of subtracting it; deleting a record that must not, in fact, be deleted – and, generally, omitting a record that *should* be in the file (the convention that the records present in the file are true does not imply that all those not in the file are false, so the model cannot determine which records are *missing*).

The reason a relational system does not prevent us from performing such operations is that, within the relational model, these operations are perfectly correct. The only way to discover the mistakes is by performing these operations together with some *other* operations, which take into account both the database structures and the other aspects of the application; in particular, the business rules implemented in the application. In other words, we can only discover the mistakes by checking the data from *outside* the model.

It is not surprising, therefore, that the relational model had to be "enhanced" with informal means that allow us to discover such mistakes. We will study these enhancements under the third delusion, but it is worth noting at this point that, despite some new and impressive terminology, what we are doing with the new features – linking the database structures with the other structures of the application – is exactly what we had been doing all along, in a much simpler way, with ordinary programming languages.

## 3

Let us return to the formal model. To explain the relational theory, textbooks give us page after page of definitions and expressions in mathematical logic. Yet, as we just saw, this formality and preciseness cannot stop us from committing outrageous mistakes. No matter how rigorous the relational model is from a mathematical perspective, the only part that is formal and precise is the definition of database entities and operations; specifically, how we combine tuples into files, and files into other files, as we move from one level of complexity to the next. And these entities and operations are so simple that we can use them just as effectively without the formal definitions.

Recall the simple system that can handle only integers and two operations, addition and subtraction. In this system, all that mathematics does is ensure that the integer of the next level is indeed the sum or difference of the integers

of the current level. Thus, the formal definitions in such a system offer very little beyond what we can accomplish using just common sense. For, we can also add and subtract integers correctly by replacing the formal definitions with an informal method, and carefully following that method. Because this system is so simple, the formal and the informal alternatives are equally practical. Even more importantly, the formal system cannot prevent us from committing such mistakes as using wrong values when translating an actual phenomenon into integers, or adding two integers when in fact we ought to subtract them. Whether we choose the formal system or an informal method, we must deal with these problems informally.

And the same is true of the relational model. All that mathematics does is assure us that each operation combines elements just as its definition says. It assures us, for example, that the selection operation indeed selects the specified records. Thus, an expression like *G(a,b,c) = F(a,b,c)* AND *a>k* defines a selection operation by saying that the new file *G* includes those tuples *(a,b,c)* which satisfy two conditions: they are true (i.e., are actual records) in file *F*, and the element *a* is greater than a certain value *k*. This formal definition is very impressive, but it is also very silly. Because record selection is such a simple concept, we can easily perform this operation by relying on common sense alone: we describe informally what we mean by record selection, and then carefully implement this concept using the basic file operations and a programming language (see figure 7-13, p. 680).

All mathematical systems appear silly, of course, if we study only the low levels. The low-level elements and operations are usually simple enough to understand intuitively, so the rigour and preciseness of their definition appear superfluous. But there is a reason for this formality. In serious mathematical systems there are *many* levels of complexity. We always start with simple elements, but we combine them so many times that we end up with very intricate ones at the higher levels. Since the elements and operations at these levels can no longer be understood intuitively, the formal definitions are our only assurance that the system functions correctly.

In a simple system, on the other hand, the elements do not increase in complexity as we move to higher levels, so there is no benefit in combining them more than a few times. With a simple system, therefore, we rarely create more than a few levels, and we can only model simple phenomena. In the system of integers and two operations, for instance, the sum or difference of two integers is still an integer. And there aren't many applications where all we need is to add or subtract integers while repeating these operations endlessly, level after level. Applications that require many levels also require an increase in the complexity of elements and operations.

In the relational model, too, the same type of elements (tuples and files) and

the same type of operations (SELECTION, UNION, PRODUCT, etc.) are found at both the lowest and the highest levels. And as a result, there is no benefit in combining elements more than a few times. Thus, few requirements involve more than three or four files, or more than three or four operations, in the creation of a new file. It is difficult to imagine a situation where we have to perform a series of a dozen selections, products, and projections, each operation starting with the result of the previous ones.

The software theorists claim that the relational model offers benefits similar to those of the traditional mathematical systems, but this is not true. In science and engineering we start with simple elements like integers, and simple operations like addition, but after many levels we end up with such concepts as calculus and analytic geometry. The complexity of the elements, as well as the complexity of the operations, keeps increasing as we move to higher levels. With the traditional mathematical systems, therefore, we derive important benefits when adding levels; in particular, the higher complexity permits us to model more complex phenomena. Evidence of these benefits is also found in that the formality and preciseness are now critical: unlike the selection of records in a database, we can hardly replace concepts like differential equations with methods based on common sense alone.

In the relational model, it is the restriction to high-level operations that prevents us from using more than a few levels. Software applications do have many levels of complexity, starting with simple entities like statements and database fields, and ending with the logic of a whole business system. But these are not the levels of a simple structure. Unlike mathematical systems, which can be represented with one structure, software applications comprise *many* structures – structures that must share their elements if they are to model our affairs accurately. And it is often low-level elements like statements and database fields that must be shared.

Among these structures are also the database structures, but with a relational system the lowest-level elements that can be shared are the files. If we take the fields, records, and files to be the three lowest levels of a database structure, the relational operations only permit us to access files. Starting with files we can then create even higher levels – that is, further files. But the interactions with the other structures are not as versatile as those we could create by starting with records and fields. Most interactions, in fact, are now too awkward or inefficient to be practical.

Thus, we see no benefits in creating more than a few levels of relational operations, not because we do not *need* higher levels, but because the restriction to operations on whole files prevents us from creating the combinations of software entities needed to attain those levels. This is why the original model was useless, and why the features added later serve mainly to bypass the

restriction to whole files: they restore the means to link the database structures to the rest of the application through lower-level elements (through individual records and fields), thus permitting more alternatives at the high levels.

## 4

With any mathematical system, we must perform the translation in order to attain the precise format required for the manipulation. But in itself the translation is a detriment: not only does it constitute additional work, but its informality detracts from the exactness of the manipulation. We can justify the use of a mathematical system, therefore, only if the manipulation confers significant benefits; that is, if it permits us to perform some important and difficult tasks. And this is indeed the case for the mathematical systems used in science and engineering: the manipulation in these systems is very elaborate, with many levels of complexity, while the translation may be as simple as converting things like weight, voltage, or time into numerical values.

In a relational system, the opposite is true: the manipulation is trivial, and it is the translation that ends up very elaborate. In order to have a mathematical database model, the part that is the manipulation had to be restricted so much that it involves in the end only trivial mathematics. The most difficult aspects of database programming – updating operations, integrity rules, the second and higher normal forms, the database language – were left out of the model and became part of the translation. They were left out, not because they do not entail manipulation, but because *that* manipulation cannot be represented mathematically.

So the manipulation includes only queries, and the queries permit only high-level operations on whole files. In any case, these queries are so simple that they can be implemented without mathematics. The basic file operations, we saw earlier, allow us to scan and relate files, and to select records and fields. Thus, the operations permitted by the relational model – selection, union, product, and the rest – can be easily programmed with ordinary iterative and conditional constructs.

To deal with those aspects of the database that make up the translation, and which were left out of the formal model, we need programming skills. So in the end we use the power of mathematics for the relatively simple manipulation, which hardly requires a formal system, while depending on informal programming methods for the difficult tasks.

Unlike the mathematical systems used in science and engineering, then, the relational model confers no benefits. In the traditional fields, mathematics permits us to accomplish tasks that are impossible without a formal system; so

the translation, with its drawbacks, is worthwhile. Relational mathematics, on the other hand, is so simple that it can be replaced with a few lines of programming; so the drawbacks of the translation exceed the benefits of the manipulation. The idea behind the relational model is, therefore, senseless. What is the point in seeking a formal system for the query operations, if all the work required to prepare the database for these queries must remain informal? Since we must continue to depend on programming for the difficult translation, we may as well use programming also for the relatively simple manipulation.

The theorists promote the relational model by pointing to its mathematics, and implying that it provides the same benefits as the models of science and engineering. But if the relational model uses only trivial mathematics, the claim is a fraud. In reality, very little of the phenomenon of a database is amenable to an exact, mechanistic representation. Mathematics is useful for phenomena where changes are rare. Take the bodies in the solar system, for instance: we can represent their motion mathematically because their properties are fixed; so, with only a small investment in the translation, we gain the great benefits of the manipulation. In the database phenomenon, however, changes are very common. These changes include adding or deleting records, and modifying the data stored in fields. Each change produces a slightly different database – different data, and hence different relationships. No mathematical system can accurately represent such a changeable phenomenon, and it is for this reason that the theorists exclude the updating operations from the formal model.

The relational idea is worthless because we have to leave too much out of the manipulation in order to represent the database functions mathematically. What we leave out is far more than what we leave out in the traditional uses of mathematics. We must leave out of the formal model the database changes and all the related issues; in particular, the integrity rules and most of the normalization. These features are in reality as much part of the application as are the queries. So, for the model to be truly useful, they would have to be included in the manipulation. Only if we decide that databases are mainly query systems can we treat issues like updating, integrity, and normalization as part of the informal translation, rather than the formal manipulation. But then we must no longer claim that the relational model is useful for general applications.

The fact that so much had to be left out of their formal model ought to have worried the theorists. This was an opportunity to realize that the relational

concept is fallacious, that databases cannot be usefully represented with a mathematical model. Instead, fascinated by the little that *could* be represented mathematically, they saw in the relational concept the beginning of a new science.

But an even greater deficiency than the separation of query operations from updating operations is the separation of the query operations from the other operations performed by the application. As we saw, the relational operations are restricted to manipulating whole files, rather than individual records and fields, like the traditional file operations. And, while in principle individual records and fields can be treated as tiny files and accessed with the relational operations, this method is far too awkward and inefficient to be practical. In effect, then, the relational model does not permit us to manipulate freely the low-level database entities. The immediate consequence of this limitation is that it is impossible to link, at the level of records and fields, the structures formed by database entities and operations with the structures formed by the other aspects of the application. And no serious application can be developed without these links.

Even for query systems, therefore, the relational model cannot be said to work if the only queries that can be implemented are those possible through operations on whole files. To be truly useful, a database system must allow us to manipulate database entities in any conceivable way.

In conclusion, the relational model is indeed a revolution in database concepts, in that it imparts to database programming the rigour and exactness of mathematics; but only *if* we restrict ourselves to queries; and *if* we restrict ourselves to queries that can be expressed through certain parameters (so that the database can be separated from the application and accessed only through operations performed on whole files); and *if* we restrict ourselves to normalized files (although it is possible, in principle, to implement any queries using normalized files, this is usually too complicated or too slow to be practical); and *if* we can ensure that all the operations that modify the database are performed correctly (so that the data upon which the queries are based is valid at all times).

Note that these restrictions describe the *original* model; so this model, absurd as it is, is in fact optimistic. As no practical uses were found for it, means had to be provided eventually to link the database structures with the other structures of the application. And the only way to do this was by permitting *low-level* database operations, which bypass the original restrictions. But if those restrictions are essential in order to attain an exact model, if we bypass them we will no longer enjoy the benefits of mathematics, not even in a narrow range of applications. So those benefits, which were insignificant in the original model already, were reduced in the end to zero.

# 5

There is no better way to conclude our discussion of the first delusion than by showing how the relational model is presented to the public. We just saw that there are no mathematical benefits in using a relational database. The database experts, however, promote the relational systems by praising *precisely* their mathematical background. Here are some examples: "The mathematical concept underlying the relational model is the set-theoretic *relation*."[3] "The relational model is founded on the mathematical disciplines of predicate calculus and set theory."[4] "The relational data model is based on the well developed mathematical theory of relations. The rigorous method of designing a data base (using normalization …) gives this model a solid foundation. This kind of foundation does not exist for the other data models."[5] "The relational approach is based on the mathematical theory of relations.… The results of relational mathematics can be applied directly to relational data bases, and hence operations on data can be described with precision."[6] "The relational model is based on the mathematical notion of a relation. Codd and others have extended the notion to apply to database design."[7] "The solid theoretical foundation guarantees that results of relational requests are well defined and, therefore, predictable."[8] "One of the benefits of working with the relational approach to databases is that it can be couched within the formalism of first-order predicate logic. As a result a mathematical foundation is available for dealing with database issues when databases are all relational."[9] "The reason we could define rigorous approaches to relational database design is that the relational data model rests on a firm mathematical foundation."[10] "In the

[3] Jeffrey D. Ullman, *Principles of Database Systems* (Potomac, MD: Computer Science Press, 1980), p. 73.

[4] Anthony Ralston and Edwin D. Reilly, eds., *Encyclopedia of Computer Science*, 3rd ed. (New York: Van Nostrand Reinhold, 1993), p. 1161.

[5] Shaku Atre, *Data Base: Structured Techniques for Design, Performance, and Management*, 2nd ed. (New York: John Wiley and Sons, 1988), p. 90.

[6] James Martin, *Computer Data-Base Organization*, 2nd ed. (Englewood Cliffs, NJ: Prentice Hall, 1977), p. 204.

[7] Catherine M. Ricardo, *Database Systems: Principles, Design, and Implementation* (New York: Macmillan, 1990), p. 177.

[8] Candace C. Fleming and Barbara von Halle, *Handbook of Relational Database Design* (Reading, MA: Addison-Wesley, 1989), p. 32.

[9] Barry E. Jacobs, *Applied Database Logic*, vol. 1, *Fundamental Database Issues* (Englewood Cliffs, NJ: Prentice Hall, 1985), p. 9.

[10] Henry F. Korth and Abraham Silberschatz, *Database System Concepts*, 2nd ed. (New York: McGraw-Hill, 1991), p. 209.

formulation of relational data models, the mathematical theory of relations is extended logically where required to meet data management objectives. The mathematical foundation of relational data models permits elegant and concise definition and deduction of their properties."[11]

As we saw, it is not difficult to show that the model's mathematical foundation is irrelevant. Yet no one in the academic world – not the mathematicians, not the philosophers, not the engineers – ever challenged these claims. Nor did anyone challenge the other software theories. The computer scientists can invent any theories, thus, no matter how absurd, confident that the academic community, the software practitioners, and the rest of society will accept them unquestioningly.

Most people trust and respect the universities, without realizing that what the academics are promoting is not ideas that are useful, but ideas that help them maintain their privileged position even if useless; in particular, the idea that science means simply the pursuit of mechanistic theories – whether sound or not, whether useful or not.

Moreover, by fostering the mechanistic ideology, universities make it possible for the software companies to promote fraudulent concepts. The mechanistic ideology benefits incompetents and charlatans, therefore, by making their activities look like serious research, or like legitimate business.

## The Second Delusion

### 1

The second delusion is the idea of normalization: the belief that, within the relational model, the problem of database design has been turned into a formal theory. In reality, the principles of normalization do not constitute an exact procedure, but one that can only be implemented informally. (The concept of normalization was introduced earlier; see pp. 704–706.)

The delusion of normalization can be summarized by saying that it is an attempt to replace the simple process of *avoiding* incorrect file relationships, with the complicated process of *eliminating* them after allowing them into the database. To justify the need for normalization, the theorists misrepresent the design problem. Traditionally, we used methods that helped us to create a correct database, and thereby avoided data inconsistencies. Now we are expected to ignore those methods, deliberately create an incorrect database,

[11] Dionysios C. Tsichritzis and Frederick H. Lochovsky, *Data Models* (Englewood Cliffs, NJ: Prentice Hall, 1982), p. 93.

discover the consequent problems, and then use normalization to convert the incorrect database into a correct one.

In the end, it is only this contrived, absurd procedure that the theorists managed to formalize, not the *actual* problem of database design. As we will see presently, even under normalization the correct database structures can only be discovered *informally*, by studying the application's requirements – just as we do when following the traditional design methods. The concept of normalization, thus, is a fraud. By inventing pompous terms to describe what are in fact senseless principles, and by discussing these principles with great seriousness, the relational experts delude themselves that they have turned database design into an exact theory.

Note that here, in the discussion of the second delusion, I am using the term "normalization" to refer only to the second and higher normal forms; that is, to the problem of field dependency. The *first* normal form (which restricts fields to single values) is unrelated to the higher ones; it is part of the formal relational model, and hence part of the first delusion.

Note also that, although the mathematical pretences of the second and higher normal forms resemble the first delusion, these transformations are not required at all by the formal model. A database, in other words, does not have to be normalized in order to satisfy the mathematical restrictions of the formal model (I will return to this point later). The higher normal forms are only needed in order to prevent problems that arise when *updating* the database; and the updating operations lie outside the scope of the formal model.

Thus, it would be wrong to treat the higher normal forms as part of the first delusion. Their *annulment* (the process known as *denormalization*) does constitute, however, the same kind of delusion as the annulment of the first normal form, or the annulment of the other aspects of the relational model. All annulments, therefore, are discussed under the third delusion.

Let us review the concept of normalization – how the relational theorists present the problem of data inconsistencies, and its solution.

Each piece of information in the database should exist in only one place, because data that is duplicated may cause various inconsistencies when records are added, deleted, or modified. The relational theorists call these inconsistencies "update anomalies." The unnecessary repetition of data also wastes storage space, but it is the anomalies that are the main reason for

normalization. In fact, depending on the size of the duplicated fields and the number of records involved, normalization sometimes *increases* storage requirements. Still, the theorists say, the benefits are so important that we should normalize our files even at the cost of increased storage space.

It must also be noted that there is always an alternative to normalization: the inconsistencies can be avoided by performing additional operations in the application (additional checks and, when required, additional updating). Normalizing the files is generally a simpler and more efficient solution, but sometimes those operations are the better alternative. In the relational model, though, this too is unacceptable: we must *always* normalize our files.

Duplication, and hence redundancy, occurs when we store some data in several records in a certain file while that data could be stored in only one record in another file: repeating customer data like name or address in every order belonging to that customer, repeating product data like description or price in every order line with that product, and so on. Clearly, fixed data should be stored in a separate file – a customer file or a product file, in this case. Only data specific to an order should be stored in the orders file, and only data specific to an order line in the order lines file. A field in the orders file will contain the customer number, and a field in the order lines file will contain the product number. These fields will serve as links to the customer and product files. Thus, when processing an order, we can access the customer data by reading the customer record; and when processing an order line, we can access the product data by reading the product record. The same is true of an invoice file, a transaction file, a sales history file, and any other file that needs customer or product data.

With data separated in this manner, when there is a change in a customer's name or address, or in a product's description or price, we only need to modify the customer or product record, rather than all the orders for that customer, or all the order lines with that product. If there were no customer and product files, an anomaly would occur if we modified the customer data in an order, or the product data in an order line: any other orders for that customer, or any other lines with that product, would continue to have the old, and hence wrong, values. Another anomaly would occur if we had to store data for a customer that has no outstanding orders, or for a product that is not currently on order: we would have to create a dummy order just so that we had a place to store customer or product data.

Data redundancy can also be viewed as the result of a mistaken relationship between two fields in the same record; specifically, a *misplaced dependency* of one field on another. A field should depend only on the field or fields that make up the record's key. There is no need for other relationships *within* a record; and if such a relationship exists, some data will be redundant. This is

true because, if one field can be determined from another, its value will be repeated unnecessarily in all the records where the other field has a particular value. We only need to specify the dependency between the two fields in one place. So the correct way to store this information is as a single record, in a separate file.

Generally, to eliminate the redundancy associated with one misplaced dependency (the dependency of one or several fields on a given field), we must create one extra file in the database.[1] Each level of normalization – the levels known as second, third, Boyce/Codd, fourth, and fifth normal forms – is a more stringent implementation of this principle. Each level, that is, will eliminate a more subtle type of dependency. These types – known as functional dependency, transitive dependency, multivalued dependency, and join dependency – differ in the types and combinations of fields that form the misplaced dependency: a non-key field depending on only some of the fields that make up the key (instead of depending on the whole key), or a non-key field depending on another non-key field, or a key field depending on another key field, or more than two fields depending on one another. The classification of the normal forms is such that, in addition to being more subtle and more rare, each level represents a broader category of misplaced dependencies – a category that includes as a special case the one at the next lower level.

## 2

One aspect of the second delusion is the belief that, because the ideals of normalization are discussed only with the relational model, they are exclusive to relational databases. The theorists present the concept of normalization as if no one had been aware of the problem of data redundancy and inconsistencies before we had relational databases, and as if the relational model and the normalization principles were the only way to deal with this problem. They never mention the fact that this problem and its solution are *identical* to their counterparts in databases created with the traditional file operations. And they are identical because they are concerned with files, records, fields, and keys – elements that are identical (despite the new terminology) in relational and in traditional databases. With one type of database or the other, redundancy and inconsistencies indicate a faulty design, a database that does not match the application's requirements. And the solution is to modify the design so as to satisfy the requirements.

[1] Several extra files are required when the relationship involves three or more interrelated fields (the kind of dependency resolved by the fifth normal form).

The issue of normalization, then, is perceived as an important part of the relational model while being, for all practical purposes, a separate theory. It was tacked on to the relational model because it was invented by the same theorists, but it could be applied to any database model that uses files, records, fields, and keys. For, what we are asked is simply to replace the traditional principle of designing a database so as to *avoid* redundancy and inconsistencies, with the absurd principle that we must start with a faulty design and then modify it so as to *eliminate* the redundancy and inconsistencies. Thus, nothing stops us from employing this absurd principle with a *traditional* database. All we have to do is deliberately create an incorrect database, and then normalize it in order to eliminate the consequent problems. The final, correct database would be identical to the one we create now simply by following the traditional design principle.

It is also worth noting that we can attain the ideals sought by normalization more effectively with traditional databases than we can with relational ones. Ironically, while the relational theory makes the problem of redundancy and inconsistencies look like a new discovery, insists on strict normalization, and overwhelms us with formality and new terminology, its restriction to high-level operations often *prevents* us from solving this problem. And it is with the traditional file operations, where we don't even use terms like "normal form," that we can more easily deal with it. This is true because those operations are more versatile and more efficient than JOIN, the operation that combines files in the relational model. Since the process of normalization separates fields by creating additional files, we must read and combine more files and more records later, when *accessing* the database. And it is when the performance degradation caused by these additional operations becomes unacceptable that we must leave some data unnormalized. Thus, since the traditional file operations permit us to combine files and records more efficiently than does JOIN, we can afford to separate more fields – and hence attain a higher level of normalization – in a traditional database than in a relational one.

We can appreciate even better why the problem of redundancy and inconsistencies is not part of the relational theory by recalling the mathematical foundation of the relational model, predicate calculus (see pp. 697–698). The relation described by the logical expression *P(x,y,z)*, for instance, consists of those tuples of elements *x*, *y*, and *z* that are related through the predicate *P*. Specifically, we substitute for the three elements certain values selected from their respective domains of permissible values, and we retain those combinations of values for which the expression yields *True*.

Now, there is nothing in this definition of a relation to prevent two elements in a tuple from forming an additional relationship. For example, if $y$ depends on $x$ in such a way that we can always derive its value from that of $x$, $y$ is in effect redundant. But this redundancy is harmless; that is, if we combine this expression with other expressions, the redundancy will be reflected perhaps in the final result, but it will not cause a logical inconsistency.

The original, formal relational model is similar: even if mistaken, the dependency of one field on another in a given file does not cause an inconsistency when that file is combined with others through relational operations. The formal model is concerned only with the *structure* and *combination* of files. Thus, even if there is a misplaced dependency in that file, the consequent redundancy is harmless. All that will happen is that some fields in the resulting file will also be related through a misplaced dependency.

The reason we can have misplaced dependencies in predicate calculus and in the formal relational model is that these systems do not include *updating* operations. Predicate calculus is not concerned with the way tuples ended up in the relation, or the way elements acquired their current value, but only with the operations that *use* the tuples. And the formal relational model is not concerned with the way records are created, deleted, or modified, but only with the operations that *read* the records. Thus, since only updating operations can cause inconsistencies, we need not worry about misplaced dependencies when we restrict ourselves to the formal model. (The relational theorists acknowledge this fact by calling the inconsistencies *update* anomalies.) To put it differently, if we restrict ourselves to queries, and particularly to queries expressed through the relational operations, we need not worry about misplaced field dependencies.

So the theory of normalization is irrelevant for applications restricted to the formal model. It is only for the broader model, which includes various informal aspects, that it has any significance. The original papers mentioned only briefly the updating operations that would be required in an application, and the language through which they would be specified.[2] It was assumed that these operations, along with the problems they might cause and the checks needed to avoid these problems, would be similar to those used in other database systems. No one tried to extend the formal model by including, say, a formal set of updating operations. It was assumed, in other words, that the exact, formal model would provide all the important database operations. The updating operations, as well as the operations needed to protect the database from redundancy and inconsistencies, were seen as a minor issue; so the plan

[2] See, for example, E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* 13, no. 6 (1970): 377–387.

was to implement them informally, just as they were implemented in other systems. We find evidence for this interpretation in that the second and higher normal forms are not mentioned at all in the original papers. The terms "normal form" and "normalization" in these papers refer only to what is called now the *first* normal form.

It was when the theorists turned the formal model – originally meant only for queries – into the basis of general database systems, that the idea of normalization had to be extended. By calling the new normal forms "second," "third," etc., the theorists made them look like a natural extension of the first one, although they are unrelated. While the first one is concerned with eliminating data structures *within* fields, the higher ones are concerned with eliminating misplaced dependencies *between* fields. The first one is needed in order to base the formal model on predicate calculus, but the higher ones are needed only if we perform updating operations. By making the latter look like an extension of the first one, though, the theorists managed to mask the fact that the relational model was changing from an exact theory into a collection of informal concepts. While everyone thought that the precision of the formal model was being extended to cover all aspects of database work, in reality the exact opposite was taking place: what had been originally the *informal* aspects of the model – the updating operations, the higher normalization, the integrity rules, the database language – was becoming the actual model, and the *formal* part was becoming irrelevant.

One wonders, if the updating operations constitute an informal aspect of the relational model, why is it so important to formalize the normalization process? Why do the theorists attempt to reduce the problem of redundancy and inconsistencies to an exact model, if the problem only concerns the updating operations, which are informal in any case? The answer is that the theorists saw in the normalization principles an extension of the original, formal model. The precision which that model offered for queries, they thought, can now be extended to the design phase, and to the updating operations; so we will soon have a mathematical model for the whole database concept.

As we will see later, the formality of the normalization process is specious. The theorists are indeed discussing the subject of dependencies in a formal manner, but, ultimately, we can determine the relationship between two given fields only by studying and interpreting the application's requirements; that is, informally. As is the case with all mechanistic pseudosciences, the relational theorists noticed a few patterns and regularities (the normal forms and the field dependencies), and jumped to the conclusion that an exact theory is possible for the design of file relationships. The same naivety that led earlier to the belief that the resemblance of records to the tuples of predicate calculus can

be the basis of a practical database model, led now to the belief that a neat classification of field dependencies can be the basis of a formal model for database design.

## 3

When studying the problem of data redundancy and inconsistencies, we notice a marked discrepancy between the way it is presented by the relational theorists and its *actual* difficulty. The theorists discuss this subject with great seriousness, the way one would discuss the most difficult problems a programmer can encounter. In reality, this is one of the simplest programming problems. And it is a problem that does not lend itself to formal treatment, so an exact theory has no practical value in any case.

It is hard to think of anyone designing a database where the "anomalies" so seriously discussed by the theorists could occur at all. Even a novice can recognize the absurdity of storing fixed customer information only in the invoice records, or repeating fixed product information in every order line with that product. And if mistakes like these go undetected and end up in the working application, it is hard to imagine a place where the programmers or the users fail to understand why customer data is lost when an invoice is paid, or why two order lines with the same product show different descriptions. Then, once they understand the problem, it is hard to imagine them failing to discover the solution; that is, keeping the fixed data in a separate file. To put this differently, a person incapable of dealing with this simple problem would be unable to deal with any other programming problem. His applications wouldn't work, and those anomalies would be the least of his worries. Thus, it is highly unlikely that a place can exist at all where the theory of normalization can confer any benefits.

And indeed, before it was brought into the limelight by the relational experts, we treated the problem of redundancy and inconsistencies as we did every other programming problem: we recognized its importance, but we never tried to explain it with an exact theory, or to solve it with a formal method. As evidence of its simplicity, we didn't even think that the process of solving it needed a special name; it is only for the relational theory that terms like "normalization" and "normal form" had to be introduced. And for those of us who have continued to use the traditional database design method, the attempt to turn this subject into an exact theory has had no significance whatever. We are treating the problem of redundancy and inconsistencies exactly as we did thirty or forty years ago, simply because the theory of normalization is irrelevant.

Relational database books devote at least one chapter to the subject of normalization. And the more thorough among them intimidate us with their formal tone and lengthy explanations, the countless definitions and theorems, and the new terms and symbols. Thus, if we ignore its content and judge it solely by its style, the discussion of normalization in a database book resembles the kind of discussions found in engineering books. Readers new to the relational theory are impressed by this formality and expect to learn some important facts. Invariably, though, they find the discussion hard to follow. Then, when the book illustrates the theory with actual examples of unnormalized files and their conversion to normalized ones, these readers react by exclaiming, "But this is how I would have designed the database in the first place!" So it is only through actual examples that we can comprehend the theory of normalization at all, and at that point the reason for the earlier difficulty becomes clear: since we would intuitively create the correct, normalized files to begin with, we struggle to understand what is the problem that the theory is trying to solve. To most of us it doesn't even occur – until we read a relational book – that anyone would design a database by repeating, say, the customer address fields for each order, or the product description field for each order line.

The difficulty, then, is not in understanding the principles of database design, but in understanding the theory of normalization: the attempt to reduce database design to formal and exact methods. We must make an effort to understand the normalization problems and their solutions because we normally don't think in a way that can create these problems. The problems are contrived, unreal. They were *invented* by the theorists, in order to have a reason for seeking a formal solution.

Typically, the books start by showing us an incorrect design and its drawbacks. They continue then by showing us how to convert it into a correct design. But what is the point of this discussion if hardly anyone would even *consider* the incorrect alternative? The theorists are defining, classifying, and explaining in the style of mathematical analysis some implausible situations – situations we never encounter in real life. With just common sense and a little practice, we already know how to create correct databases. The normalization theory, on the other hand, asks us to study some strange problems (the difference between the second and third normal forms, why we have the so-called Boyce/Codd normal form between the third and fourth, how to convert a file from first to second or from second to third, etc.) and to assimilate an endless list of strange concepts (superkey, dependency preservation, nonloss decomposition, left-irreducible functional dependency, etc.).

It is the attempt to formalize the problem of field dependency, data redundancy, and data inconsistencies, and the need to fit the incorrect designs into

the classification of normal forms, that we find hard to understand – not the actual principles of database design. And when we finally understand the new concepts, we realize that in practice we never encounter these problems. When we learn to program we don't learn two things – how to design incorrect databases, and how to convert incorrect databases into correct ones; we simply learn how to design correct ones. Thus, since the problems studied by the theory of normalization concern mostly the transition from bad to good design, it is not surprising that the theory is, for all practical purposes, irrelevant.

## 4

We saw earlier, in "The Basic File Operations," that the concept of records, fields, and keys allows us to implement any file relationships we need in our applications. And we also saw that this concept is identical in traditional and in relational databases. The difference lies mainly in the new terminology and in the way the files are used. In traditional databases, we use the basic file operations through the flow-control constructs of a programming language; and we specify, through indexes, the individual records. In relational databases, we use only the high-level relational operations; and, rather than individual records, we specify whole files or logical portions of files. But in both cases we must create the same files and fields, and the same relationships, in order to implement a particular set of requirements.

Thus, although our earlier discussion concerned traditional databases, the same design principles apply to relational ones. With one type of database or the other, the traditional principles permit us to create correct – that is, normalized – databases directly from the application's requirements. To appreciate the absurdity of the normalization theory, then, let us review the traditional design concepts.

The decision we must make when designing a database is what files, fields, and keys are needed; that is, what data to store in the database, how to distribute it among files, and how to relate the files, in order to satisfy the application's requirements. Thus, since the files depend on the application's logic, we usually implement them together with the various parts of the application. It is the file *relationships* that pose the greatest challenge. For, if all we needed were isolated files (a customer file, a product file, a history file, etc., with no links between them), designing the database would be trivial, little more than creating the respective fields.

Files are related through the values present in their fields. Typically, identifiers and codes are used to relate files (product number, invoice number,

category, etc.). A relationship is established when two files use such a field, and some records in both files contain the same value in this field.[3] Depending on how the relationship is used in the application, we may use either key fields or non-key fields. Often, a combination of several fields, rather than a single field, is needed to relate files.

And it is the correct choice of relationships that ultimately determines whether or not there will be redundancy or inconsistencies in the database – what the theory of normalization is concerned with. The various normal forms, as we will see shortly, are nothing but a complicated way of expressing these relationships. In reality, all we have to do is create a database that correctly represents the application's requirements; and if we do this, there will be no redundancy or inconsistencies. In other words, if we understand the application's requirements, and if we implement them correctly, we don't need a theory of normalization (because we create "normalized" files from the start); and if we don't understand the requirements, or fail to implement them correctly, no theory can help us.

❖

Four types of file relationships are possible between two files: one-to-one, one-to-many, many-to-one, and many-to-many. The terms "one" and "many" refer to the number of records in the first and second file that are logically linked.

Two files are in a *one-to-one* relationship when one record in the first file is related to no more than one record in the second file. Thus, the two files will have the same number of records when each record in the first one has a corresponding record in the second one, and they will have a different number of records when some records in either file have no corresponding records in the other. Files that are in a one-to-one relationship can always be combined into a single file, where each record comprises the two corresponding records: we simply merge their fields, and when there is no corresponding record we assign null values or default values to the respective fields. For practical reasons, though, it is sometimes preferable to have two files rather than one. For example, if a file has many fields but some operations involve only a few, we may decide to keep these fields in a separate, smaller record, in order to improve the application's performance.

An example of one-to-one relationship is an employee file and a special functions file, with the condition that a function may be performed by only one

[3] Relations based on field equality are the most common, but, strictly speaking, any values can be used to relate files. With a date field, for example, we can create a relationship where a record containing a certain date in the first file is logically linked to those records in the second file where a date is up to one year earlier.

employee, and an employee may select no more than one function. At some point in time we may have, say, 80 records in the employee file and 30 records in the functions file, but only 20 functions actually selected; thus, 60 employees will have no corresponding function, and 10 functions no corresponding employee. The two files are linked by adding a function number field to the employee record, or an employee number field to the function record (or both, if we need two-way links).

The most common relationship is *one-to-many*. Two files are in a one-to-many relationship when one record in the first file (the "one" file) is related to one, several, or no records in the second file (the "many" file), while each record in the second file is related to one or no records in the first file. Here are some examples: customer file and customer orders file (one customer may have one, several, or no outstanding orders, but each order belongs to one customer); orders file and order lines file (one order may include one or several order lines, but each line belongs to one order); employee file and payment history file (each employee has one record in the history file for each pay period, but each pay period record belongs to one employee). A one-to-many relationship is also a *many-to-one* relationship, when seen from the perspective of the second file: several orders are related to the same customer, several order lines to the same order, several pay periods to the same employee.

The one-to-many relationship is implemented by making the "many" file's key a combination of both files' identifying fields. For example, if we make the key in the orders file the combination of customer number and order number, we will be able to select for a given customer any one of the corresponding records in the orders file, and for a given order the single, corresponding record in the customer file. However, when the relationship is seen as many-to-one and a direct link is not required from the "one" file to the "many" file, the "one" file's identifying field can be just a non-key field in the "many" file. Thus, the key in the orders file would be just the order number, and we would access the customer records by including the customer number as a non-key field.

Two files are in a *many-to-many* relationship when one record in the first file is related to one, several, or no records in the second file, and at the same time one record in the second file is related to one, several, or no records in the first file. To implement such a relationship, we create a service file to act as a link between the main files. The service file has only key fields, and its key is simply the combination of the two main keys. For example, if some vendors supply several products, and certain products are supplied by several vendors, the vendor and product files form a many-to-many relationship. The key in the service file is the combination of vendor and product numbers, and we implement the two-way links between files (vendor to product, and product to vendor) by providing *both* sorting sequences: products within vendors, and

vendors within products. (If using traditional file operations, we accomplish this by creating two indexes for the service file.) We can then select for a given vendor the corresponding records in the product file, and for a given product the corresponding records in the vendor file.

The four types of relationships can be combined to link more than two files. Thus, a set of *several* files can form a one-to-one relationship, when any two files in the set are in a one-to-one relationship. Also, a file can be in two many-to-many relationships at the same time: with one file through one field, and with another file through another field.

The most versatile relationship, however, is one-to-many. One way to combine one-to-many relationships is by having several "many" files share the "one" file, through the same field or through different fields. The customer file, for example, can be related through the customer number to both the orders and the sales history files. One-to-many relationships can also be combined to form hierarchies of more than two levels, by using the "many" file of one relationship as the "one" file of another. For example, for each order in the orders file we can have several lines. We store then the line-related data in an order lines file, and use the combination of customer, order, and line numbers as the key. The order records will function, at the same time, as "many" in their relationship with the customer records, and as "one" in their relationship with the order lines records. Most applications require a mixture of combinations: several levels, and several files on each level. Thus, several "many" files may share the "one" file while acting at the same time as "one" files in other relationships.

It is also possible for two "one" files to share the "many" file. For example, if a customer purchases several products and a product is purchased by several customers, there will be a set of records in the sales history file for each customer record, and another set for each product record. But these sets will overlap: each history record will be related at the same time to a certain customer and to a certain product. Thus, in addition to being the "many" file for both the customer and the product files, the history file serves to create a many-to-many relationship between them. (The many-to-many relationship, we see now, is merely a special case of two one-to-many relationships that share the "many" file – the case where this file's sole purpose is to link the "one" files.)

Although the four types of relationships are usually described as *file* relationships, they are also *field* relationships. When two files are related as one-to-one, or one-to-many, or many-to-many, it is through their records that the

relationship exists: one or several records in one file correspond to one or several records in the other. But records are made up of fields, so the same correspondence exists between fields: the relationship between files is reflected in each pair of fields. Thus, when two files are related as one-to-one, each field in the first file is in a one-to-one relationship with each field in the second file; in addition, fields that belong to the same file are in effect in a one-to-one relationship with one another. When two files are related as one-to-many, each field in the first file is in a one-to-many relationship with each field in the second file. And when two files are related as many-to-many, each field in the first file is in a many-to-many relationship with each field in the second file.

For example, if the product and the orders files are related as one-to-many, a field like the product description or price in the former will be related as one-to-many to fields like the order date or quantity in the latter. What this means in practice is that the same product description or price may be associated with several order dates and quantities.

We can regard the four types of relationships, therefore, as either file or field relationships. So, rather than saying that two files are related and the field relationships reflect the file relationship, we can say that it is the fields that must be related, and the file relationship will reflect the field relationships. We design a database by creating relationships that match the requirements. In some situations we think in terms of *file* relationships; and once we create the files, it is obvious to which file each field must be assigned. In other situations it is better to think in terms of *field* relationships; and we implement the files and file links that will allow us to relate those fields as required.

Consider this example. We want to store some information about our products, so we start with a file that contains just the key field, the product number. If the requirements say that there may be several orders for each product, the product number is related as one-to-many to the order number. The order number must be, therefore, in a separate file, so we create an orders file with two fields: the order number as the key, and the product number as the link to the product file. Next, we need a product description field, which is always the same for a given product; it is related as one-to-one, therefore, to the product number, so we assign it to the product file. We then need an order date field, which is always the same for a given order; it is related as one-to-one to the order number, so we assign it to the orders file. (This also relates it as many-to-one to the product number and description, which is what we want.) Next, we need an order quantity field; like the date, it is related as one-to-one to the order number, so we assign it to the orders file.

This process, clearly, can be continued for each new field. And, since most requirements reflect common needs, an experienced programmer will easily design a correct database. Only in unusual situations do we have to analyze carefully the requirements to determine how to treat a new field.

The foregoing example, while very simple, already demonstrates that it is the application's requirements, not some database principles, that determine what is a correct database. Thus, if the requirements changed and the product description were permitted to differ from one order to the next, the description field would have to be in the orders file rather than the product file (because it would now be related as one-to-one to the order number, date, and quantity, and as many-to-one to the product number). Similarly, if the requirements permitted several lines in an order, the product number and quantity would be related as many-to-one to the order number. So they would be assigned to a separate file, order lines, where the key is the combination of order number and line number, and several records correspond to one order record. The order date, though, would stay in the orders file, because it continues to be related as one-to-one to the order number.

We know that relationships can be one-to-one, one-to-many (or many-to-one, if seen in reverse), many-to-many, and combinations of these. So, if we understand the role that a new field must play in the application, we already know what relationship to create, and hence to which file to assign it. (Key fields duplicated in another file in order to relate the two files are treated differently, of course.) All we need in order to design a correct database is to study the application's requirements. Then, we use an appropriate combination of relationships to represent these requirements. In other words, we create the database that matches the requirements – one field at a time. Ultimately, if we understand the requirements, we are bound to create a correct database.

And when we create a correct database, the problem of redundancy and inconsistencies does not arise. (The only time we must deal with this problem is when we *deliberately* introduce redundancy into the database; that is, when avoiding it would make the application too slow.) This is true because in a correct database all field relationships reflect actual requirements. Thus, in the foregoing example we assigned the product description to the product file because the requirements stated that it was the same for all orders. If we assigned it to the orders file instead, we would end up with unwanted duplication: a product's description would be repeated unnecessarily in each order that includes the product. The duplication can be explained by noting that this relationship does not reflect the requirements: the description field would be related as many-to-one to the fields in the product record, while the requirements called for a one-to-one relationship. (Alternatively, the error can be described as a one-to-one relationship with the fields in the

orders record, while the requirements called for a one-to-many relationship with these fields.)

The most important lesson from this analysis is that data redundancy and inconsistencies can only be defined within the context of a particular set of requirements. So this is not a problem that can be solved by means of a formal database theory. This is a *programming* problem, one that can be solved only by taking into account both the database structures and the other structures that make up the application. It is the way we plan to use the files that determines what are the correct relationships. And with correct relationships, there will be no redundancy or inconsistencies.

A database, then, can be correct only for a specific set of requirements. With just a small change in requirements, the same database would no longer be correct. The incorrectness may manifest itself in the form of wrong values or unnecessarily duplicated values. In the earlier example, storing the description in the product record is correct if it must be the same in all orders, and wrong if it must change; conversely, storing it with each order is correct if it must change, and wrong if it must be the same in all orders. The presence of redundancy and inconsistencies, therefore, when unintended, is similar to any other programming error: we neglected the requirements, and consequently the application malfunctions. The error, in this case, is a discrepancy between the required file relationships and the actual ones.

It is worth repeating: the concept of file and field relationships applies to relational databases *exactly* as it does to traditional ones, because both types are based on files, records, fields, and keys. Thus, even those programmers who prefer the relational model can benefit from the traditional design methods. They too can avoid data redundancy and inconsistencies by creating a correct database directly from requirements. Even with a relational database system, therefore, there is no need for a theory of normalization – because, if we create correct relationships, there is no redundancy or inconsistency to eliminate. As is the case with the traditional databases, we simply need to understand the application's requirements and the four types of relationships.

We can appreciate even better the connection between file relationships and the application's requirements if we remember that requirements are in effect rules, or restrictions. Specifically, from all the operations that the application can perform, and from all possible values that memory variables and database fields can take, only a few must be permitted if the application is to run correctly. One type of restrictions concerns the *combinations* of values that the database fields will display at run time: how the value of one field depends on

the value of another. And it is through the four types of file relationships that we implement these restrictions.

Two fields are related as one-to-one when they can have any combination of values; that is, when neither field depends on the other. Two fields are related as one-to-many when one field is restricted to a specific value by a series of values in the other. (Many-to-one is the same relationship seen in reverse.) And two fields are related as many-to-many when there are two simultaneous one-to-many restrictions: one field is restricted by the values of the other, and at the same time possesses values that restrict the other.

By interpreting the requirements as restrictions, we can explain the problem of redundancy as follows: we provide for all possible combinations of values in a situation where only a few can actually occur. If the requirement is for one-to-many and we place the two fields by mistake in the same file, they will be related as one-to-one. We provide for *any* combination of values when, in fact, the first field will have the *same* value for a series of values in the second. So that one value will be repeated unnecessarily every time the second field's value is in that series. We only need to specify their relationship once, and yet we do it several times.

There is an obvious correspondence between the various file relationships and the normal forms of the normalization theory: the relationship that is correct for a given requirement corresponds to the highest normal form attainable for that requirement (the one for which the files are deemed to be fully normalized). The relational theorists avoid the subject of file relationships – perhaps because this would reveal the shallowness of the normalization theory. Let us take a moment, though, to study this correspondence.

The first normal form is the highest one attainable when the application's requirements place no restriction on the combinations of values that two fields can take. From the perspective of the normalization theory, this means that there is no dependency between the two fields; so they can be assigned to the same file (or to separate files if those files are in a one-to-one relationship).

The second and higher normal forms can be attained when the application's requirements place some restrictions on the combinations of values. Because of these restrictions, the correct relationship is now one-to-many; and if we create one-to-one instead (by placing the fields in the same file), we will have a relationship that permits *any* combinations, while the actual data includes in fact only *some* combinations. The normalization theory describes this problem as a misplaced dependency: the only dependency permitted within a tuple is that of a non-key field on the field or fields that make up the key. We also note

the mistake in that the file is only in first normal form, while a higher normal form is now attainable. The solution is to place its fields in separate files, thereby creating files that are in second, third, or Boyce/Codd normal form. (Which form is actually attainable depends on the combination of field types, key or non-key, that constitutes the misplaced dependency.) In traditional terms, what we do when using two files instead of one is replace the incorrect one-to-one relationship with a one-to-many relationship, which is what the requirements had called for to begin with.

Combinations of these three normal forms correspond to combinations of one-to-many relationships: two "many" files sharing the same "one" file, or two "one" files sharing the same "many" file. They also correspond, therefore, to a many-to-many relationship between two files. The more complicated fourth and fifth normal forms correspond to various many-to-many relationships involving three or more files, when some of the two-file relationships are restricted.

But this correspondence, while perhaps interesting, is irrelevant; for, in practice we don't need to know anything about field dependencies, or about the notion of normal forms. We can create the correct relationships directly from requirements, as we saw earlier. We don't have to start with an incorrect, one-to-one relationship (as the normalization theory says), note the redundancy and inconsistencies, and then try to attain the correct relationship by discovering misplaced dependencies.

## 5

We are now in a position to explain the fallacies behind the delusion of normalization. We saw that all we need in order to create correct file relationships is to understand the application's requirements. We can avoid data redundancy and inconsistencies, therefore, simply by implementing relationships that match the requirements. But, while not especially difficult, this task demands skills that most programmers lack.

Without exception, the mechanistic software theories attempt to solve the problem of programming incompetence, not by encouraging programmers to improve their skills, but by providing *substitutes* for skills. The relational theory, in particular, was meant to obviate the need for database programming skills. Instead of the traditional file operations, which must be used through a programming language, programmers will only need to understand the high-level relational operations. Moreover, the mathematical foundation of the theory will guarantee data correctness: since the relational operations are as exact as mathematical functions, and since any database requirement

can be expressed as a combination of these operations, even inexperienced programmers will create correct database structures.

But, as we saw under the first delusion, the mathematical database model is a fantasy. To attain such a model, we must restrict it so much that it loses all practical value. If we divide the use of a mathematical system into translation (the conversion of the actual phenomenon into its mathematical representation) and manipulation (the work performed with the mathematical entities within the system), only the manipulation can be formal and exact. The translation entails an *interpretation* of the phenomenon, so it is necessarily informal. The relational model is senseless because it consists almost entirely of the translation. The manipulation, while indeed exact, forms a very small part of the model; and, besides, it is so simple that we can implement the same operations by relying on common sense alone. The theorists praise the mathematical benefits of the model, but these benefits can only help us to deal with a few, simple aspects of database work. Most work, including the most difficult aspects, lie outside the scope of the formal model. So, in the end, we need the same programming skills as before.

If the manipulation includes only the little that can be reduced to an exact representation, every other aspect of database work must become part of the translation. This includes the *design* of the database; that is, discovering the combinations of files and fields that correctly represent the real entities and the relationships between them. With a traditional database or a relational one, this is an informal activity: using our knowledge and experience, we study the application's requirements and ensure that the database entities and relationships match the real ones. And if we accomplish this, there will be no redundancy or inconsistencies. The relational theory never promised to replace this activity with an exact method; it simply left the issue out of the formal model (along with such other issues as integrity rules, updating operations, database language, and database performance).

The relational theory, thus, failed to eliminate the need for programming skills. Programmers continued to create incorrect database structures, but the theorists did not recognize this problem – the fact that so much had to be left out of the formal model – as a falsification of the relational concept. So, instead of studying the problem, they introduced an additional concept – the normalization theory. Their attitude, in other words, did not change: confronted with the evidence that mechanistic theories cannot be a substitute for expertise, they hoped to contend with the persisting incompetence by inventing yet another substitute. The second relational delusion (the delusion of normalization) emerged, therefore, because the theorists refused to face the first one (the delusion of a formal database model).

The normalization theory differs from the original relational theory in that

it promises us exact methods for identifying the incorrect file relationships, not *before*, but *after* they are implemented. Rather than invoking the power of mathematics to *prevent* a bad design (something that everyone now agrees is impossible), we are told that the same power can be invoked to *correct* a bad design. Clearly, the theorists do not see the absurdity of this idea. For, were it possible to discover formally the incorrect relationships in an *existing* database, we could also discover them formally *while designing* the database. The phenomenon is the same in both cases: file relationships that do not match the application's requirements.

So the theorists still fail to understand why the original model could not help us to design correct file relationships. This is not a technical problem that might be solved with an additional theory, but a fundamental limitation: it is only through an informal interpretation of the requirements that we can determine what *are* the correct relationships. Thus, there is no difference between determining this *while* designing the database or *after*. In both cases, we must process the database structures together with the other structures that make up the application; in particular, the business practices reflected in the application. In both cases, then, we must deal with the complex structure that is the whole application, and this is something that only minds can do.

The normalization theory claims to eliminate the need for expertise by eliminating the need to design correct databases. Unlike the traditional design methods, which expect us to create file relationships that match the requirements, the new method permits us to create relationships that are as incorrect as we like. To take an extreme case, we can ignore the need for file relationships altogether: we create a database that consists of just one file, and assign *all* the fields to this file, regardless of their actual relationships. We can do this because the database we create now is only a starting point. By applying the principles of normalization, we will be able to transform the incorrect database, step by step, into a correct one.

As we know, files created within the formal model are already in first normal form. To attain the higher normal forms, we must modify the database by discovering and eliminating the misplaced field dependencies. And this can be accomplished, we are told, through the formal methods provided by the normalization theory. Through one procedure we eliminate one type of dependency, and thereby convert the files from first to second normal form; then, through another procedure we eliminate a different type of dependency, and convert them from second to third normal form; and so on. We continue this process until we find at a certain level – a level that varies from one

database to another – that there are no misplaced dependencies left. At that point, the database is fully normalized. By eliminating all misplaced dependencies, we eliminated the possibility for any data redundancy or inconsistencies to emerge later, when the database is used.

The normalization theory, thus, claims to have solved the problem of programming incompetence by replacing the challenge of designing a correct database, with an easier challenge: eliminating the errors found in an existing, incorrect database. This shift, the theorists believe, reduces database design to a series of simple, mechanical activities. Their naivety is so great that, although the logic is the same (matching the file relationships to the application's requirements), and although the ultimate database is the same, they believe that the new principles are formal and exact while the old ones are not.

In the end, the problem of design became the problem of dependency: an elaborate system for defining, analyzing, and classifying the field dependencies found in a database. Date describes this shift perfectly: "The fact is, the theory of normalization and related topics – now usually known as *dependency theory* – has grown into a very considerable field in its own right, with several distinct (though of course interrelated) aspects and with a very extensive literature. Research in the area is continuing, and indeed flourishing."[4] But this research is a fraud: the theorists are distorting and complicating the problem of database design in order to have a reason for seeking an alternative. The delusion is not so much in the shift from design to dependency, as in the belief that this shift has turned the problem into a formal theory; specifically, the belief that we have now exact methods to prevent redundancy and inconsistencies.

In reality, redundancy, inconsistencies, and misplaced dependencies are different aspects of the same phenomenon: a discrepancy between the file relationships and the application's requirements. Thus, whether we wish to avoid redundancy and inconsistencies, or to eliminate misplaced dependencies, the only way to do it is by interpreting the requirements correctly; and this task cannot be formalized. What the theorists did is *add* to this task a complicated system of principles and procedures – the theory of normalization. And it is only this theory that is formal and exact. Their "research," then, is merely a preoccupation with this theory, with the problems they invented themselves. The real problem – creating a correct database – is as informal as before. So, if the normalization principles did not replace the original problem, if we continue to assess dependencies informally, the normalization theory is fraudulent.

[4] C. J. Date, *An Introduction to Database Systems*, 6th ed. (Reading, MA: Addison-Wesley, 1995), p. 337.

To repeat, dependency is indeed part of the same phenomenon that causes redundancy and inconsistencies. So the shift from design principles to dependency principles is wrong only because it is unnecessary, because it complicates the problem without providing any benefits in return. Recalling the earlier examples, repeating the unchangeable product description in every order entails redundancy. We can describe this redundancy as the result of an incorrect relationship: we created a one-to-one relationship between the description field and fields like order number, when their required relationship is one-to-many. But we can also describe the redundancy as the result of a misplaced dependency: the description depends on the part number, which is not the key in the orders file. Regardless of how we describe the redundancy, though, it is the incorrect relationship between the product description and the other fields that is the root of the problem. And in both cases it is this relationship that must be modified in order to solve the problem.

Generally, with the traditional design concept we create the correct relationships from the start. With the normalization theory, we start by creating one-to-one relationships – which are usually wrong, because most relationships are one-to-many or many-to-many; we then search for misplaced dependencies, which direct us to the incorrect relationships; and finally, we modify the relationships in order to eliminate those dependencies, and with them the redundancy and inconsistencies.

But with both the traditional method and the new one, we always reach the point where we must decide, for a given field, whether it must be in the same file as some other fields, or in another file. With the traditional method, this decision is also the design. With the normalization theory, this decision is only a small part in a long and complicated process. For, now we must also identify the current normal form, determine the type of dependency between fields and the higher normal form that would eliminate it, and convert the files to that normal form.

The decision itself, however, entails the same challenge: interpreting the application's requirements correctly. Thus, what is the critical step with both design methods – discovering the correct relationship between two fields – is necessarily an *informal* process. So the formality of the normalization theory is silly if normalization depends ultimately on an informal process, just like the traditional method. Before, we made that decision in order to create a correct file relationship. Now we make it in order to correct an incorrect one. But, if in the end it is only through our interpretation of the requirements that we can determine what *is* the correct relationship, we may as well use the traditional method, which is so much simpler.

To conclude, there are two stages to the delusion of normalization. The first stage is the belief that we need a theory of normalization at all; namely, that preventing redundancy and inconsistencies is a special problem, which demands a formal theory. This problem, though, is no different from all the other problems that make up the challenge of programming. Regardless of which aspect of the application we are dealing with, we must create structures of software entities that correctly represent the structures of real entities. And to accomplish this task we must understand the application's requirements and the means of implementing them. Moreover, a given requirement usually affects *several* aspects of the application, and we cannot deal with them separately. The database structures, in particular, are always linked to the other structures that make up the application. Searching for a formal, mechanistic theory of database design is an absurd and futile quest.

The theorists assume that it is impossible, or very difficult, to design a correct database directly from requirements; that programmers cannot attain the necessary expertise, so this task must be replaced with a method which they can follow mechanically; and that it is possible to discover such a method. But, quite apart from the fact that no formal method can exist, the traditional design principles already provide a fairly simple method for creating correct databases. All we need to do is determine, for each new field, the appropriate relationship with the existing fields (one-to-one, one-to-many, many-to-one, or many-to-many). If we do this, we will end up with a correct database – a database that matches the requirements. And, among the many benefits of a correct design, there will be no redundancy or inconsistencies.

The second stage in the delusion of normalization is the belief that the body of principles that make up this theory constitutes indeed a formal solution to the problem of database design. In reality, the database structures are still based on the relationships between fields, and we can only determine the correct relationships by interpreting the requirements; in other words, informally, just as before. The theorists think that studying field dependencies rather than field relationships has resulted in a method that is formal and exact, but what is formal and exact is only the new principles. These principles did not replace the informal task of understanding the requirements; so that task – upon which the correctness of the database ultimately depends – has remained unchanged.

If we divide the design process into two parts, formal and informal, the traditional method is almost entirely informal, while the new one is almost entirely formal. But this improvement is an illusion. What confuses the theorists is that the part which they invented, and which is indeed formal, keeps growing, while the traditional part (understanding the requirements) remains the same. Recalling an earlier quotation, research in this area is flourishing. Thus, the more preoccupied they are with the dependency theory,

the smaller the informal part appears to be. The informal part, after all, consists simply in determining, for a given field, its relationship with the other fields. In the end, though, this decision is the only thing that matters – what will make the database correct or incorrect – with both the traditional method and the new one. But, while this decision is practically the whole design process with the traditional method, with the new method it is such a small part that it goes unnoticed. So the theorists delude themselves that the new method is entirely formal.

The formal part, thus, did not eliminate the informal one in the new method; it is *additional* to it. The formal part, while impressive, is absurd if the correctness of the database depends ultimately on the small part that is informal – on the part that, with the traditional method, is the only thing we need.

So the conclusion must be that the concept of normalization is worthless. It is an artificial, unnecessary theory. The critical part is still the informal task of determining what field relationships match the application's requirements. But by spending most of their time with formal and complicated procedures, and only moments with that informal task, the relational enthusiasts can claim that database design is now an exact science.

We examined earlier the first stage of the delusion of normalization: the belief that we need some new, formal principles, when in fact the traditional concepts provide an excellent and relatively simple design method. In the following pages we examine the second stage: the belief that the principles of normalization provide indeed a formal design method, when in fact the critical part is as informal as before.

## 6

Like predicate calculus, which inspired it, the formal relational model is a true mathematical system, complete with operations and formulas. Its weakness, we saw under the first delusion, is only that it is irrelevant to database work: when we depict the use of a relational system as the *translation* of database entities into mathematical ones and their *manipulation* within the system, we find that the manipulation – the most important aspect in other mathematical systems, and the reason for performing the translation – plays an insignificant part.

The normalization theory, on the other hand, is not a mathematical system at all. The theorists discuss it as seriously as they do the formal relational model, but on closer analysis we discover that all they do is *present* it formally. There are no true operations or formulas in this theory, as there are in the formal model; all we have is a study of field dependencies, expressed through

formal notation. The theory of normalization, in other words, consists *entirely* of a process of translation: from the real entities into the relational ones. There is no manipulation at all. The only operations available are those we had under the formal relational model.

An example of the specious mathematics of the normalization theory is found in a long paper written by E. F. Codd – a paper generally regarded as the most rigorous treatment of the second and third normal forms.[5] The paper provides an exhaustive analysis of field dependencies and their elimination, but despite the formal tone and terminology, this is not a mathematical theory. The paper describes various combinations of data elements, and represents their relationships and dependencies by means of a formal system of notation. The resulting expressions *look* perhaps like mathematical formulas, but they serve no purpose beyond this representation. Page after page of expressions are, in reality, only the *translation* of files and fields into the new notation. Once the translation is complete, we have no way to *manipulate* the expressions. All the system does, then, is represent field dependencies formally. Were this a true mathematical system, we would have some new relational operations, to replace the original ones.

We find the same style in thousands of other writings. What is described as mathematics is merely a system of definitions and theorems expressing in formal notation various issues pertaining to the subject of field dependency. Typically, the papers introduce new terms and define them through references to other terms, show how to derive certain parts of the system from other parts, prove that if certain conditions hold then other conditions will also hold, and so forth. And this is where the mathematics ends.

It is the introduction of new terms that the authors are especially fond of. The relational theory in general overwhelms us with new terminology, but the principles of normalization in particular seem to require some new terms at every step. Thus, along with the formal tone, the rich terminology helps to make the normalization theory appear important, no matter how shallow it actually is. But, while the mathematical style of these writings impresses naive readers, an intelligent person merely finds the writings incomprehensible. The reason is that, since we know that the whole theory is unnecessary, we have little motivation to assimilate the countless terms and definitions; and without understanding the new concepts it is impossible to follow the author's discussion.

To convey the flavour of this style, I will quote a few lines from Date's book (out of the seventy pages devoted to the subject of normalization). After

[5] E. F. Codd, "Further Normalization of the Data Base Relational Model," in *Data Base Systems*, ed. Randall Rustin (Englewood Cliffs, NJ: Prentice Hall, 1972), pp. 33–64.

presenting several related theorems, Date defines the fourth normal form as follows: "Relation *R* is in 4NF if and only if, whenever there exist subsets *A* and *B* of the attributes of *R* such that the (nontrivial) MVD $A \twoheadrightarrow B$ is satisfied, then all attributes of *R* are also *functionally* dependent on *A*."[6] Concepts like "nontrivial," "MVD," and "functionally dependent," used in this definition, are explained on previous pages. For example, MVD (multivalued dependency) is defined as follows: "Let *R* be a relation, and let *A*, *B*, and *C* be arbitrary subsets of the set of attributes of *R*. Then we say that *B* is *multidependent* on *A* – in symbols, $A \twoheadrightarrow B$ (read '*A* multidetermines *B*,' or simply '*A* double-arrow *B*') – if and only if the set of *B*-values matching a given (*A*-value, *C*-value) pair in *R* depends only on the *A*-value and is independent of the *C*-value."[7]

It is also worth mentioning the following warning: "We stress the point that the discussions that follow are intended to explain a *formal theory*, albeit in a fairly informal manner."[8] In other words, definitions and explanations like those quoted above, and the endless formulas and diagrams, are not the actual theory but a *simplified* version. For the *really* formal discussion we must consult the original papers, in academic journals.

To summarize, all that the normalization theory does is represent formally the relationships between fields. A true mathematical system would provide operations that combine entities to create increasingly high levels, as do the systems used in engineering. There are no such operations here, so the normalization theory does not describe a mathematical system. What it describes is *a formal system of representation*. This system may have its uses, but not in the way a mathematical system has. In the end, the only mathematical manipulation remains the one provided by the original relational model. The normalization theory is not a true enhancement of that model.

So what the relational theorists invented is akin to a game. The normalization work is *additional* to the task of studying and implementing the application's requirements. That task has remained as important – and as informal – as before. It is only the game that is formal and exact. This is a sophisticated and difficult game, demanding a special kind of knowledge. It is not surprising, therefore, that the academics who invented it, and the practitioners who learn it, feel that their normalization work is a sign of expertise. This is expertise in playing a game, though, not in designing databases.

[6] Date, *Database Systems*, p. 329. [7] Ibid., p. 328. [8] Ibid., p. 327.

# 7

The reason we cannot have a formal *and* useful theory of normalization is that the dependency of one field on another is not a *database* problem, but part of the application's logic. Formal normalization principles can only deal with the *database* structures. They cannot take into account the other structures that make up the application – the business practices, for instance. And it is these other structures that determine, ultimately, the relationships between database fields. A formal theory, thus, can deal with such issues as the definition and classification of dependencies, or the conversion from one normal form to another; but it cannot tell us whether the relationships are correct. In particular, no formal theory can tell us to which file to assign a given field. Only our knowledge of the application can do this.

Recalling the earlier examples, assigning the product description to the same file as the product number is not right or wrong in an absolute sense, but only relative to the requirements being implemented: if the description is fixed, it should be in the same file; if changeable, in the other file. We *must* understand the requirements. And when we do, we already know how to implement them: as a one-to-one or as a one-to-many relationship. Thus, a formal theory cannot replace the need to study the requirements, and is unnecessary once we understand them. It is, in other words, useless.

Let us take another example. An employee file usually includes such fields as department, position, salary, seniority code, and vacation code. Now, these fields may be related in one company, and unrelated in another. The salary, for instance, may be independent, or the same for all the employees with a particular position; the vacation code may be independent, or the same for all the employees with a particular seniority; the position and salary may be independent, or the same for all the employees in a particular department. Some of these fields, therefore, may be dependent on others, in which case they should be moved into separate files: a salary file where the key is the position, a vacation code file where the key is the seniority code, and so on. But only *we* can know whether a given field is or is not independent; and we would know this in the same way we know the other requirements that define the payroll application. The same application, in fact, may be used by two companies while a certain field is independent in one but not in the other. So, just like the business practices that make up an application, the normalization requirements may be different in each case; and as a result, a database that is deemed to be normalized for one company may not be for the other. Again, since it is only *we* that can discover the field relationships, a formal theory is useless.

Another situation where the need for normalization is determined largely by our knowledge of the application occurs when files are updated only under certain conditions. Thus, some files may be used by the application in such a way that a relationship of dependency between two fields in the same record would be harmless. For example, records may be added but not modified or deleted; or those fields alone may never be modified. Also, there are situations where it may be simpler or more efficient to deal with the problem of dependency through the application's logic, rather than through database restrictions. In all these situations, what we do is simplify the application or improve its performance by noting that not all conceivable database operations will *actually* be performed. Clearly, no formal theory can include such knowledge.

The only formal theory of normalization possible is one that assumes the worst case; namely, the case where *every* field may depend on another field. As we saw, we eliminate each dependency by separating the two fields: we place one field in a new file, where the records are linked through their key to the field left in the first file. Thus, if we want to be absolutely certain that there are no dependencies, and if we don't want to rely on an interpretation of the requirements, we must separate in this manner every field, in every file. In the end, every file in the database will have only one non-key field. This is an exact, formal procedure – a procedure that can even be automated. However, because many of the separated fields must be put back together in the running application, this overnormalization would make the application too complicated and too slow; so no one seriously suggests that we follow it. (In fact, as we will see under the third delusion, even minimal normalization – separating just a few fields – is often impractical and must be forsaken.)

A database where the smallest necessary number of fields (rather than an arbitrarily large number) were separated in an attempt to eliminate all dependencies is said to be in optimal second normal form. This sounds like a precise definition, but in reality it is only informally, through our knowledge of the application, that we can determine whether or not the normalization of a given database is "optimal." Again, the only way to have a formal theory is by separating *every* field in the database, regardless of how it is used in the application.

But even if we succeeded somehow in developing an exact and complete theory of normalization, it would still be inadequate. This is true because normalization deals only with dependencies that can be eliminated by separating fields. There are many other types of field dependencies in an application, all a natural

part of the application's logic. Every application includes operations that relate fields in the same record, or fields in separate files. Some of these fields, therefore, depend on others; so they are, strictly speaking, unnecessary. But we cannot eliminate these dependencies through normalization, by separating fields.

Let us examine a simple example of the type of dependency that cannot be eliminated through normalization – the classic case of aged balances. The customer balance, for instance, is usually stored in several fields in the customer record: current, thirty-day, sixty-day, and ninety-day balances. And there is usually an additional field, for the total balance, which is the sum of the other four. But if the total balance is always the sum of the aged balances, its field can be eliminated. Instead of having a separate field, we can calculate the total balance (by adding the other fields) wherever we need it in the application. The reason we usually retain the total balance field is that this is simpler than calculating it: in most applications we modify it in only a couple of places (typically, when invoicing the customer and when receiving payments), but we show it in dozens of inquiries and reports. So it is simpler to update the total balance in the few places where an aged balance changes, and merely to *read* it in the other places.

It is obvious that the dependency of the total balance on the aged balances cannot be eliminated through normalization, by moving the total balance into a new file. What we do for this type of dependency, therefore, is similar to what we do when we decide *not* to normalize in situations where normalization *is* possible: we anticipate the problems that may be caused by the updating operations, and we add to the application's logic the necessary steps to prevent them. Thus, in the case of balances, we must remember to update the total balance too, when one of the aged ones is updated. And if we neglect this, we will face "update anomalies" (the total balance will no longer equal the sum of the aged ones) not unlike those that occur in unnormalized files when we ignore the effect of updating operations.

To continue this example, in most applications the aged balances themselves can be calculated, using a transactions file: we read the records belonging to a particular customer, and total the invoice and payment amounts under four different periods. So the aged balance fields too are dependent on other fields, and hence unnecessary (although the original data is now in another file). Also like the previous dependency, this dependency cannot be resolved through normalization. To prevent "update anomalies" (balance fields different from the sum of the transactions), we must either eliminate the balance fields, or ensure that they are updated whenever a record is added to the transactions file. (In this case, though, eliminating the fields is rarely practical, because it is too inefficient to calculate them by reading the transaction records every time.)

So what is the point in seeking a formal theory of normalization, if this theory would eliminate only *some* dependencies? Clearly, there is no limit to the types of field dependencies that can exist in an application – types like the ones we have just examined. In fact, we don't even think of these dependencies as a database problem, but as various aspects of the application's logic. Since most software requirements involve database fields – fields belonging to one file or to several files – it is natural to find relationships of dependency between fields. And it would be absurd to eliminate these relationships solely in order to avoid redundancy, or to avoid inconsistencies in updating operations. What we do in each case is seek the most effective design: we eliminate the dependency when practical, and deal with the updating problems as part of the application's logic when this is simpler or makes the application faster.

In the end, all field dependencies cause similar problems, and we can only deal with these problems by taking into account not just the database structures but *all* the structures that make up the application. These are not *database* problems but ordinary programming problems, similar to the many other problems we face when developing an application. And it is just as futile to search for an exact and complete theory of field dependency as it is to search for an exact and complete theory of programming. The relational theorists isolated *one type* of dependency – the type that can be eliminated by separating fields; and they naively concluded that, if we eliminate this one type, we will eliminate *all* the problems caused by dependency (or, at least, the most common problems).

This belief is reflected in the relational vocabulary (terms like "normalize" and "normal form" imply a particular, proper data format) and in the numbering system (the fifth normal form is said to be the last and most stringent one). Hardly ever are the other types of dependencies mentioned at all. Date discusses them briefly: "5NF is the *ultimate* normal form with respect to projection and join.… That is, a relation in 5NF is *guaranteed to be free of anomalies* that can be eliminated by taking projections [i.e., by separating fields].… Of course, this remark does not mean that the relation is free of *all possible* anomalies. It just means (to repeat) that it is free of anomalies that can be removed by taking projections."[9] Most authors, however, depict the process of normalization as a final refinement, as a guarantee of database validity.

Thus, by emphasizing the few dependencies that can be eliminated through normalization while disregarding the many that cannot, the relational experts make the normalization principles appear more important than they really are. Then, they use this misrepresentation to rationalize their search for a theory of normalization.

[9] Ibid., p. 334 and footnote.

# 8

If the theory of normalization is unnecessary, if the traditional design method permits us to avoid redundancy and inconsistencies simply by understanding the application's requirements, how do the theorists justify their lengthy discussions? By distorting the problem of database design. They describe some contrived database structures that are incorrect but hardly ever occur in practice, and then they show us how to turn them into correct ones.

The only theory they can offer us is one that studies the so-called normal forms and gives us methods to convert files from one form to another. But we need such a theory only if we normally create incorrect databases. The theorists present the incorrect databases as a common occurrence, and the concept of normalization appears then important. In reality, we can create correct databases from the start, by selecting file relationships that match the application's requirements. So the classification of normal forms and the conversion procedures have no practical value.

I will illustrate this distortion now with a few examples taken from database books. In all these situations, we will see, the correct design can be easily determined from the requirements. The authors, however, *ignore* the requirements, and start with a *deliberately incorrect* design: a single file, when several are needed. They start, that is, with a one-to-one relationship when the requirements call for one-to-many or many-to-many. They point to the problems caused by the incorrect design, and *then* they study the requirements and show us how to arrive at the correct one: through normalization.

The examples, in other words, are presented so as to make the theory of normalization, which in reality is totally unnecessary, look like an indispensable concept in database design. Moreover, their method is so lengthy and complicated that the reader is likely to miss the fact that its preciseness and formality are specious: the most important decisions – identifying the misplaced field dependencies – are still being made, not mathematically, but through an *informal interpretation* of the requirements.

Brathwaite demonstrates the second normal form with this simple problem:[10] we want to store some information about students and about the classes they

[10] Ken S. Brathwaite, *Relational Databases: Concepts, Design, and Administration* (New York: McGraw-Hill, 1991), pp. 76–77.

attend; students are identified by a student number, and we must record their name and major; classes are identified by a class number, and we must record the class location and time; a student may attend several classes, and we must be able to identify these classes.

Ignoring all we know about normalization, we note that the students and classes form a many-to-many relationship (a student attends several classes, and a class is attended by several students). So the student number and class number must be in separate files: a student file, where the student number is the key, and a class file, where the class number is the key. The student name and major are both related as one-to-one to the student number, so they must be non-key fields in the student file. Similarly, the class location and time are related as one-to-one to the class number, so they must be non-key fields in the class file. Lastly, to link the two files, we need a service file where the key is the combination of student number and class number. In a traditional database, the service file could then have two indexes: class number within student number (to select the class records associated with a student), and student number within class number (to select the student records associated with a class). But the requirements call only for the link from student to classes, so we need in fact only the first index. (It is worth noting that in a real application the link file wouldn't be just a service file; it would also have some non-key fields, for data that is related as one-to-one to its key – the student's grade, for instance.)

Brathwaite, though, attempts to implement the requirements with *one* file: the combination of student number and class number is the key, while the student name and major, and the class location and time, are non-key fields. Then, he notes the problems caused by this design: no information can be stored about a particular student unless the student is enrolled in at least one class, or about a particular class unless at least one student attends it. Also, a certain name and major will be repeated for every class attended by that student, and a certain location and time will be repeated for every student attending that class; so if these values change, several records would have to be updated.

What causes these problems, Brathwaite explains, is the dependency of non-key fields on *part* of the key: while the key includes both the student and the class numbers, the student name and major depend only on the student number, and the class location and time only on the class number. Non-key fields must depend on the whole key, so the solution is to create a separate file for the two student-related fields, with the student number alone as the key, and another file for the two class-related fields, with the class number alone as the key. What will be left in the original file is just its key, the student and class numbers. This design eliminates all the aforementioned problems.

The final database, thus, is identical to the one we created earlier, directly from the requirements. We knew all along that it was correct, simply because it reflects accurately the requirements. Now, however, we are told that it is correct because the files are in second normal form (whereas the original file, with all fields bundled together, was only in first normal form).

What is the point of this approach? Starting with one file would make sense, perhaps, if the method used to reach the final design were indeed formal and exact (in which case we could even automate the design process). But the misplaced dependencies were discovered *informally*, by interpreting the requirements. For instance, when noting that the name and major depend only on the student number, we used the same information and the same logic as we used earlier, when noting that they are related as one-to-one to the student number. With normalization as much as with the traditional method, we relied on skill and common sense, not on mathematics. Thus, if we know how to determine the relationship between two fields, we may as well use this knowledge directly to assign them to the proper files. Why bundle them first in one file, and then use this knowledge to *separate* them?

So the part that is formal – the classification of field dependencies – did not replace the need for, nor the importance of, the part that is informal. The correctness of the normalization depends, ultimately, on the correct interpretation of the requirements. The fancy terminology makes the process of normalization seem more exact than the traditional method, when in reality it is merely more complicated.

❖

Date starts his discussion of the second and third normal forms with the following problem.[11] Let us imagine that we purchase parts from a number of suppliers, located in different cities and identified by a supplier number; the cities are identified by the city name, and each city has a status associated with it; several suppliers may be located in the same city; a supplier can sell different parts, which are identified by a part number; and we want to record our purchase orders by storing for each order the supplier number, part number, and quantity. (The requirements assume, for the sake of simplicity, that only one order exists at a given time for each combination of supplier and part number, so we don't need order numbers. Also, the requirements call for the capability to identify directly the city of a given supplier, but not the suppliers in a given city.)

With our knowledge of file and field relationships, we can translate these

[11] Date, *Database Systems*, pp. 297–303.

requirements into the following design. We note first that the city is related as one-to-many to the supplier, so we need two files: a city file, where the key is the city name, and a supplier file. In one-to-many relationships, the key of the "many" file includes usually the "one" file's key; so here it would be the combination of city name and supplier number. But the present requirements do not call for selecting the suppliers in a given city, so the key in the supplier file can be just the supplier number. We do have to select the city associated with a supplier, though, so we include the city name as a non-key field. The status is related as one-to-one to the city, so we add it as a non-key field to the city file. The supplier number is related as one-to-many to the order-related fields, part number and quantity; so these fields must be in a third file, orders, where the key is the combination of supplier number and part number.

Date, however, says nothing about these relationships. He starts by bundling all five fields (supplier number, status, city name, part number, and quantity) in one file: the orders file, where the key is the combination of supplier number and part number. And immediately he notes the consequent redundancy and anomalies: Since there must be a record in this file for every order, the information that a certain supplier is located in a certain city will be repeated for every order from that supplier; so, if the supplier relocates to another city, we will have to modify several records. Similarly, the information that a certain city has a certain status will be repeated for every order from every supplier in that city; so, if the status changes, we will have to modify several records. Lastly, we cannot store the information that a certain supplier is located in a certain city unless an order exists for that supplier.

Date then presents the solution. The first step is to separate the fields by creating a new file: the supplier file, where the key is the supplier number, and the city name and status are non-key fields. The quantity is left in the orders file. Since each combination of supplier and city appears now in only one record, the redundancy associated with the city, along with the update anomalies, has been eliminated. The solution can be expressed in terms of misplaced dependencies: while non-key fields must depend on the whole key, the city and status in the original file were dependent only on the supplier (they are the same for all the orders from a given supplier). In terms of normalization, the problem was solved because the new files are in second normal form, while the original one was only in first normal form.

But this still leaves the other redundancy: the status of a certain city is repeated in the supplier file for every supplier located in that city. Although not as bad as in the original file (where the repetition was for every *order* from every supplier in that city), this redundancy will nevertheless cause the same kind of problems. The misplaced dependency that must be eliminated now is between the status and the city (two non-key fields). So we create a new file:

the city file, where the key is the city name, and the status is a non-key field. The supplier file will then be left with only the city as a non-key field. In terms of normalization, the problem was solved because these two files are in third normal form. In other words, while the second is the highest normal form attainable for the orders file, we can attain the third for the supplier file by creating a separate city file; and a database is fully normalized only when each file is in its highest attainable normal form. (The difference between the second and third is in the type of misplaced dependency that is eliminated: on only a portion of the key, and on a non-key field.)

So by the time he is done, Date ends up with exactly the same database as the one we created directly from requirements with the traditional design method. The normalization method is more complicated, and we *still* depend on an informal decision: we identify the misplaced field dependencies by interpreting the requirements, the same way we identified the correct field relationships before. What is formal is only the *analysis* of these dependencies and the *conversion* from one normal form to another; that is, the work that is *additional* to the task of identifying them.

Carter uses the example of an employee file to demonstrate the fourth normal form.[12] Specifically, we have to store for each employee, in addition to his name, some data about his children and about his salary history. Thus, we need a set of fields for each child (identified by the child's name), and a set of fields for the salary of each past year (identified by the year). We will have an employee file where the employee number is key, and the name (related as one-to-one to the number) is a non-key field. And we will have two one-to-many relationships, with the employee file acting as shared "one" file: between employee and children, and between employee and salary history. In the children file, the key will be the combination of employee number and child name; and in the salary history file, the combination of employee number and year. We will then be able to select for a given employee the corresponding child records and history records; and for a given child or year, the corresponding employee record.

Carter, however, starts by showing us what would happen if we placed the child and salary history fields in the same file – a file where the key is the combination of employee number, child name, and year: we would have to repeat the entire salary history for each child. For instance, for an employee with 3 children and 10 years of history, there would be 30 records in this file:

[12] John Carter, *The Relational Database* (London: Chapman and Hall, 1995), pp. 135–150.

one record for each combination of child and year. This design, therefore, would cause redundancy and anomalies: to add or modify the data for one child, we would have to add or modify 10 records (because the same child data is stored for each year); and to add or modify the history data for one year, we would have to add or modify 3 records (because the same history data is stored for each child).

Now, *no one* would try to combine child data and salary history in one file. Carter must start with this absurd design in order to demonstrate the benefits of normalization. It is pointless to describe his actual analysis – fifteen pages of complicated principles, definitions, and diagrams related to the fourth normal form, not to mention nearly forty prior pages dealing with the lower normal forms. Briefly, that file suffers from multivalued dependencies (i.e., several fields dependent on one another). The solution is to separate it into two files, one for child data and the other for salary history – which is exactly how *we* designed the database to begin with.

The redundancy and anomalies were eliminated, we are told, because these files are in fourth normal form, while the original file was only in Boyce/Codd normal form. But *we* know that the database is correct simply because it expresses two one-to-many relationships, which is what the requirements actually called for. Carter needs an enormously complicated procedure to reach the same design that *we* reached simply by implementing, directly from requirements, the appropriate file relationships. Moreover, the critical observation that the child data and history data must be separated could only be made *informally*, by studying the requirements – just as we identified the file relationships with the traditional design method.

Date explains the fourth normal form with a more difficult example.[13] We are asked to design a database to express the relationships between the courses, teachers, and textbooks in a certain school, with the following requirements: a particular course may be taught by one or more teachers, and a teacher may teach one or more courses; a particular course may use one or more textbooks, and a textbook may be used in one or more courses; a particular course always uses the same textbooks, regardless of the teacher.

Studying the requirements, we note two many-to-many relationships: between courses and teachers, and between courses and textbooks. We need, therefore, three main files (courses, teachers, and textbooks) linked through two service files. To satisfy the requirement that a teacher may teach several

[13] Date, *Database Systems*, pp. 325–329.

courses and at the same time a course may be taught by several teachers, we create a service file where the key is the combination of course and teacher; and to satisfy the requirement that a course may use several textbooks and at the same time a textbook may be used in several courses, we create a service file where the key is the combination of course and textbook.

As usual, in order to implement the two-way links between files (course to teacher and teacher to course, course to textbook and textbook to course), the service files must provide *both* sorting sequences: teachers within courses and courses within teachers, textbooks within courses and courses within textbooks. (Thus, if we use a traditional database, there will be two indexes for each service file.) We will then be able to select for a given course the corresponding records in the teachers file, and for a given teacher the corresponding records in the courses file; and we will also be able to select for a given course the corresponding records in the textbooks file, and for a given textbook the corresponding records in the courses file.

This, then, is how a sensible database book would present the example – the problem and the solution. Let us see now how Date presents it. He starts by attempting to implement all the relationships with *one* service file – a file where the key is the combination of course, teacher, and textbook. (So there is one record in the file for each combination of values in the three fields.) But this design is absurd; it is deliberately incorrect in order to demonstrate the transition from one normal form to another. The file, Date explains, is only in Boyce/Codd normal form, and this gives rise to redundancy and anomalies. For instance, if a particular course uses two textbooks, we will need two records for every teacher who teaches that course, although all teachers use the same textbooks. In addition to this duplication, we would have to add, delete, or modify several records (one for each teacher) when adding, deleting, or modifying the information about a textbook. Expressing the problem in terms of dependencies, the design is incorrect because it permits a multivalued dependency.

But this is a gross simplification of Date's actual explanation – four pages of complicated pseudo-mathematical analysis, which is in fact incomprehensible without a good understanding of some fifty prior pages on the subject of normalization.

The solution, Date concludes, is to have two service files rather than one, and to separate the three key fields into two sets of two fields.[14] More specifically, it is the teacher and textbook fields that must be separated.

[14] It must be noted that Date does not call these files *service* files, thus suggesting that they are the main data files (i.e., tables). A real application, though, would also require some non-key fields, to store details about courses, teachers, and textbooks; and such fields would not be added to these files, because that would cause much redundancy.

The result, needless to say, is the two service files we created previously, when we implemented the database as two many-to-many relationships. The redundancy and anomalies were eliminated, we learn now, because these files are in fourth normal form.

The design method based on file relationships, we saw, leads directly to the correct database. Date describes a situation that is a good example of a fundamental database concept, the many-to-many relationship. But instead of discussing this concept, he presents a silly, deliberately incorrect design. Then, he uses this design to justify the need for the normalization theory.

And, as in the previous examples, the complexity of the normalization masks the fact that the critical step (the observation that it is the teachers and textbooks fields that must be separated) was based on an *informal* interpretation of the requirements – exactly the same interpretation that helped us to determine the correct relationships with the traditional method.

## 9

We saw earlier that the principles of normalization are not, in fact, required by the original relational model: they are not an extension of the formal model, but an attempt to formalize the process of database design (see pp. 732–734). The normalization theory is, in effect, an independent theory – a theory that can be applied to any system based on records, fields, and keys. Thus, we can study the normalization theory on its own, ignoring the relational model altogether. And when doing so, its character as a mechanistic delusion becomes even clearer. By way of summary, therefore, I want to recapitulate the normalization fallacies and to show how they arose from the mechanistic way of thinking that pervades the academic world.

Mechanists attempt to explain a complex phenomenon, which can only be represented with a complex structure, by breaking it down into simpler phenomena: they extract smaller and smaller aspects of it, until they reach an aspect that can be represented with a simple structure. And at that point they discover an exact theory – a theory based on that aspect alone. But this discovery is a trivial, predictable achievement; for, if we keep reifying *any* phenomenon, we are bound to reach, eventually, aspects simple enough to allow an exact theory. The discovery, nevertheless, generates a great deal of excitement, so the mechanists initiate a research program. The more elaborate their research becomes, the more confident they are about its importance. Although it is obvious to everyone that the theory explains only that one isolated aspect, the mechanists promote it as if what it explained were the original, complex phenomenon.

The phenomenon of a database comprises many aspects, of which the most important are the application's *requirements* and the *file relationships*; that is, the *actual* entities and relationships, and their *representation in software*. And these two aspects consist, in their turn, of many aspects. Among the other aspects of this phenomenon are the field dependencies, the data redundancy, and the inconsistencies (the so-called update anomalies).

The aim of the normalization theory is to find a formal, exact method for designing the file relationships from a knowledge of the requirements (or, at least, for determining whether a given set of relationships matches the requirements). Now, it may be possible to represent with one structure the relationships on their own, or the dependencies, or the redundancy, or the inconsistencies, or perhaps even a combination of them. But the database phenomenon as a whole is complex, because these aspects interact with the requirements, which in turn interact with many other aspects of the application. Thus, no mechanistic theory can represent the system that consists of the file relationships *plus* the requirements. No formal method can exist, therefore, to determine whether or not a given set of relationships matches the requirements.

Because they could not discover a theory for the *actual* database phenomenon, the software mechanists tried to discover a theory by breaking down the phenomenon into simpler ones. They noticed that the inconsistencies occur when the file relationships are incorrect; and they also noticed that the inconsistencies are related to data redundancy and to field dependencies. It is the misplaced dependencies, they concluded, that cause redundancy and inconsistencies. And since this one aspect of the original phenomenon is simple enough to represent with an exact theory, they made *it* their subject of research. The *dependency* theory is believed to be the answer to the original problem: if we study, analyze, and classify the various types of field dependencies, the mechanists say, we will discover a formal method for avoiding misplaced ones; this will then prevent data redundancy and inconsistencies; and the lack of redundancy and inconsistencies will indicate that the file relationships match the requirements.

But this logic is fallacious. The dependencies, like the redundancy and the inconsistencies, are merely one aspect of the database phenomenon. They are not the *cause* of correct or incorrect file relationships, but just a different way of viewing them. So it is absurd to study the dependencies in the hope of determining from them the correct relationships. The *requirements* are the real determinant in this phenomenon. It is only from the requirements, therefore, that we can determine other aspects of the phenomenon: when there is no discrepancy between the requirements and the file relationships, there are no misplaced dependencies, no redundancy, and no inconsistencies; and when

there is a discrepancy, we note misplaced dependencies, redundancy, and inconsistencies.

It is indeed possible to explain the relationships, the redundancy, and the inconsistencies in terms of dependencies; but this is true because they are closely related aspects of the same phenomenon, not because the dependencies *cause* the other aspects. Thus, instead of a dependency theory we could develop an equally elaborate redundancy theory, to study, analyze, and classify the various types of data redundancy; or an inconsistency theory, for the various types of data inconsistency; or a relationship theory, for the various types of file relationships. And each theory could then be used to "explain" the other three aspects, just as the dependency theory is said to explain the redundancy, the inconsistencies, and the relationships.

From the requirements, then, we can determine the other aspects, but not the other way around. The mechanists base their theory on dependencies because they mistakenly interpret them as the cause of correct or incorrect file relationships. The dependencies on their own, though, are meaningless; for, we cannot decide from a dependency alone whether or not it is misplaced. Similarly, the redundancy or inconsistencies or relationships on their own, or all aspects together, are meaningless. The real cause – what can explain all four aspects – is the requirements. The dependency theory, thus, suffers from the fallacy of confusing cause and effect. It is fundamentally wrong.

Each aspect of the phenomenon of a database has its own representation: the requirements are represented by means of business practices, the file relationships by means of diagrams or programming languages, and the dependencies by means of a system of notation peculiar to the normalization theory. Similar systems could be invented to represent the redundancy and the inconsistencies, if we wanted. Each aspect provides a different view of the database, but neither is complete; only a system embodying *all* these aspects, plus those aspects we are not even discussing here, can represent the phenomenon of a database accurately. Thus, because they form a complex phenomenon, it is impossible to describe these aspects and their relationships exactly and completely. We *can* design correct databases, but this is largely an informal procedure.

Database design entails the conversion from one system of representation to another. What we want to attain, of course, is the *software* representation; that is, the file relationships. So, if it is the requirements that ultimately determine what are the correct relationships, the only conversion worth studying is the traditional one, from requirements to relationships. Because

they failed to discover a formal and exact procedure for *this* conversion, the relational mechanists shifted their attention to the study of field dependencies. Their theory does offer a formal and exact conversion, but only from dependencies to relationships. Its exactness is illusory, therefore, because to benefit from it we must ensure first that we have correct dependencies. And the only way to attain the correct dependencies is by performing the conversion from requirements to dependencies, which is as informal as the traditional one, from requirements to relationships (see figure 7-17).

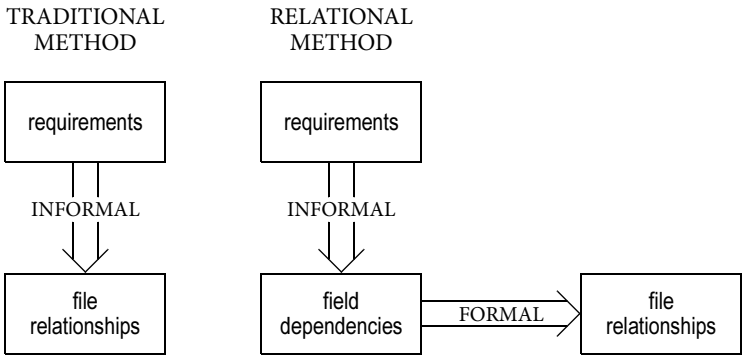


Figure 7-17

The dependency theory may appear impressive to the casual observer, but in reality an exact theory that explains relationships in terms of dependencies is a trivial accomplishment. It is not surprising that one aspect of a phenomenon can be shown to depend on another, if they are closely related. Thus, we could also discover similar theories to explain relationships in terms of redundancy, inconsistencies in terms of dependencies, dependencies in terms of redundancy, and so on. But that first step – from requirements to one of the other aspects – is always necessary, and is always informal. So we may as well use the traditional method, which entails *only* that one step – from requirements to relationships.

Of the five aspects of this phenomenon, the requirements and the file relationships are the most intuitive, and the field dependencies are the least intuitive. This is why, before the relational theory, we had no interest in dependencies; we only studied the requirements and the relationships, and sometimes the redundancy and the inconsistencies. We are asked now to replace what is the simplest method – the intuitive conversion from requirements to relationships – with a method that involves two steps, each one more complicated than our one-step method: the conversion from requirements

to dependencies – which the mechanists must perform but don't like to discuss, because it is informal – is less intuitive and already more difficult than requirements to relationships; and in addition, we have now the intricate dependency theory, for the conversion from dependencies to relationships. (It is, perhaps, precisely because the study of field dependencies is so complicated that the mechanists think it is an important discovery.)

The dependency theory is typical of the mechanistic pseudosciences. The relational mechanists settled for a dependency theory only because this is one narrow aspect of the database phenomenon for which they *could* find an exact explanation. They never proved that this theory could model the whole phenomenon. But then they forgot this limitation, and proceeded to treat the theory as if it *did* provide a formal method for database design.

In conclusion, the dependency theory – a major, thirty-year-old research program involving thousands of academics and generating a vast literature – is a worthless, senseless pursuit. No matter how exact it is, it cannot help us to determine what are the correct relationships. Thus, recalling an earlier example, the theory cannot tell us whether to assign the product description field to the product file or to the orders file. We must decide which alternative is correct during the conversion from requirements to dependencies, before we even get to use the theory. The theory may well offer us a formal, faultless conversion from dependencies to relationships, but we can only apply it after determining – informally – what *are* the correct dependencies.

## The Third Delusion

### 1

The third delusion consists of those modifications to the relational model that are presented as *enhancements*, while being in reality *reversals* of the relational principles. These modifications were introduced when it was discovered that the model worked only with small and simple databases, and was totally impractical for serious applications. Thus, while the need for reversals constitutes an obvious refutation of the relational model, the theorists describe these reversals as *new relational features*.

The original theory defined a complete database model, and, although generally worthless, was a falsifiable concept. So, had it remained an academic treatise, it could have been regarded perhaps as a serious study. But because its supporters believed that it could have practical applications, the theory had to be modified again and again. The modifications, as we will see shortly, serve largely to restore the low-level capabilities of the traditional file operations –

capabilities which the relational model had attempted to replace with high-level features. Clearly, by the time we restore these capabilities we no longer have a relational model. The third delusion is in the belief that we can continue to enjoy the benefits promised by the original model even while reversing its principles.

The relational theory, thus, was turned into a pseudoscience when its supporters, instead of admitting that it had been refuted, decided to "improve" it: they suppressed the falsifications, one by one, by incorporating them into the model in the guise of new features. This practice rendered the theory unfalsifiable. (We examined earlier the pseudoscientific nature of the relational theory; see pp. 710–712, 713–714.) The relational model was indeed rescued, but this was accomplished by annulling the relational principles and reinstating the traditional ones. And because they were reinstated *within* the relational model, the traditional principles are now far more complicated than they were on their own. Moreover, relational systems still lack the flexibility and efficiency we enjoy with the traditional file operations.

From its simple origin, and from its mathematical ambitions, the relational theory was degraded in the end to a complicated and messy concept. What is perceived today as the relational model has little to do with the original ideas. And, although we still see the claim that the model is founded upon mathematical principles, relational systems are promoted now on the strength of features that were described originally as *informal* aspects of the model. Today's relational systems consist of large, cumbersome, inefficient, and expensive development environments, which include special programming languages and an endless list of features, definitions, principles, standards, rules, and procedures that we must assimilate. And what is the purpose of this complexity? To provide a substitute for what any programmer should be able to do by using just the six basic file operations.

## 2

Let us start with the concept of normalization. There are two kinds of normalization: the first normal form (1NF), and the second and higher normal forms (2NF, 3NF, etc.). 1NF was, from the start, part of the *formal* relational model; its purpose is to restrict the data stored in each field to a single item, so that the records and files match the tuples and relations of predicate calculus. The second and higher normal forms were added later, and belong to the *informal* aspects of the relational model; their purpose is to eliminate data redundancy and inconsistencies.

As we saw, whether the goal is to avoid multiple items in a field or to

eliminate redundancy and inconsistencies, we must separate the fields of the file in question into two sets, and move one set into a new file. Each normalizing step will generally increase by one the number of files in the database. Thus, although it is quite easy to normalize files, this process makes it more difficult to *access* the data. For, we must read more files and more records, in order to put back together the fields that were separated by normalization.

The idea of separating and recombining fields looks neat when presented as mathematical logic; that is, when we assume that data records can be accessed instantaneously, just like the tuples of predicate calculus. And the additional complexity caused by the separations and combinations can be justified by invoking the ultimate benefits of normalization. In real applications, however, even if we are willing to accept the additional complexity, normalization is often impractical, because of the excessive time needed to access the data.

Whether the fields were separated in order to attain the first or the higher normal forms, the only way to recombine them is with the JOIN relational operation (see pp. 702–703). JOIN creates one file from two: it combines the records of the two files, retaining only those records where certain fields relate the files in a particular way. But, while easy to use as a high-level operation, JOIN is very inefficient and hard to optimize. This may go unnoticed with small files, but in most databases its execution takes far too long to be practical. Also, applications usually need *many* normalization steps, and hence *many* JOINs later. Even a simple query may need two or three JOINs, and perhaps hundreds of times the number of disk accesses that the traditional file operations would need.[1]

So the idea of strict normalization had to be abandoned. But the theorists refer to this reversal with such euphemisms as database "optimization," or "tuning," or "tailoring." They discuss now the benefits of *denormalization* with the same seriousness, and with the same technical, impressive language, as they did the benefits of normalization before. This makes the reversal appear like progress, like an *enhancement* of the relational model. No one mentions the fact that the abandonment of strict normalization means simply a return to the informal design principles we had followed *before* the relational model: we compare in each situation the benefits and drawbacks of keeping data together in one file, with those of using two files, and we choose the more effective alternative. This is what we routinely do when creating databases with the traditional file operations.

[1] As I remarked earlier (see p. 732), we can attain the ideals sought by normalization more effectively with *traditional* databases. As a result, what is perceived as a fundamental relational principle – normalized files – is found more often in applications using the traditional file operations than in applications using relational databases.

❖

The abandonment of the first normal form comes by way of a feature called – incredibly – *non-first-normal-form*. Abbreviated with scientific-looking terms like non-1NF, NFNF, and $NF^2$, this feature is so advanced that only a few database systems support it. Those that do are known as *extended relational* systems.

The name chosen by the experts for the new feature betrays their attitude: instead of simply stating that the first normal form – one of the fundamental principles of the relational model – has been abandoned, they present the abandonment as a new principle; and they call this principle, literally, the opposite of the original one. 1NF is still important, but now we need to impose this restriction only when convenient. Thus, the experts suppressed the falsification of an important principle by introducing a new one. In effect, the two principles, 1NF and non-1NF, cancel each other; that is, taken together they cannot possibly be serious principles. So the first normal form is now just an informal recommendation. But the experts describe this falsification of the relational model as a new, advanced relational feature.

To appreciate the significance of non-1NF, recall the 1NF restriction and its implications. For a file to be in first normal form, its fields must contain single, atomic values. Each field, in other words, must contain only one value at a time – not a list of items, or an array, or any other structure. This restriction is usually expressed by saying that the columns of a relational table must not contain *repeating groups*. The restriction to a single item per field is critical if we want to base the relational model on mathematical logic (because the elements of a tuple in predicate calculus are single items).

In most applications, however, we encounter sets of values that are so closely related that the most effective way to store and use them is as a list, or array. For example, in a file of purchased parts, we may want to store for each part a list of up to three vendor numbers, or three vendors and their selling price, or three vendors with their last price and purchase date. With the traditional file operations and a language like COBOL, we define these values, respectively, as an array of $3\times1$, or $3\times2$, or $3\times3$ elements. In the part record, the whole array will be treated as one field. It will be read into memory or written to disk along with the record, and, when in memory, its elements can be conveniently accessed with the same operations that programming languages provide for manipulating *memory* arrays. Thus, we can easily display or update one element or a subset of the elements, compare the three prices, change the relative position of the vendors, and so on.

In a relational database, the only way to store these values is as a separate file. The fields in the new file will be, for instance, the vendor number, price,

and date; and the key will consist of the part number and a sequence number, 1 to 3. For each record in the part file, there will be up to three records in the new file. Operations like comparing prices or exchanging the relative position of vendors, which can be performed with a couple of statements in a traditional database, will now be small programming projects (since we must combine the two files with JOINs, access the three sets of elements as separate rows but save them somehow so that we can use them together, and so on). What is worse, these operations will now take longer to execute, because of the additional disk accesses.

For a few fields, it is possible to bypass the 1NF principle; and the simplest way to do it is by simulating arrays with ordinary fields. In the previous example, we would add to the part record three, six, or nine fields, each one with its own name, and access them through whatever means a relational system provides for accessing individual fields. This method obviates the need for a second file and separate records, and solves therefore the performance problem; but it makes programming even more complicated. Simulating arrays with ordinary fields, thus, is an awkward trick that programmers must employ if they want to bypass the 1NF principle while pretending to like the relational model.[2]

The fact that we have to resort to tricks in order to avoid the inefficiency of a relational principle constitutes a falsification of the relational theory. And the final abandonment of 1NF, after thirty years of struggling to fit real-world problems into relational systems, is in effect an acknowledgment of this falsification. Presenting non-1NF as a new relational feature is how the relational charlatans suppress the falsification.

With non-1NF, a field in one file acts as a pointer to records in a second file. For example, if the first file contains customer records, one field may be used for that customer's invoices. But the field itself contains no information. It only points to another file: an invoice file, where the records are identified through the combination of customer and invoice numbers, and the set of invoice records associated with a particular customer record are those with the same customer number.

[2] The 1NF principle is impractical, not because it requires a second file, but because it requires a second file in *any* situation. In contrast, with the traditional operations we are free to choose, in each situation, the most effective alternative. Thus, we may decide to use a second file even to replace a *small* array, if the application must access those elements in such a way that the use of indexed data records is simpler. Conversely, if access time is critical, we may decide to use an array even if this results in a very large record size.

With this method, a record in the first file can point to any number of records in the second file. In some database systems, more than one field can act as a pointer to another file; for example, in addition to the invoice field, we can have an order field and a history field in the customer record, pointing to records in an orders file and a sales history file, respectively.

Non-1NF allows us to relate files hierarchically, by logically *nesting* one file within another. Thus, databases that utilize this feature are also known as *nested relational* databases. Nesting is not limited to one level: fields in the second file can act as pointers to further files, which become then logically nested within the second one, and so on. Non-1NF allows us, therefore, to create hierarchical file structures. And, since the original relational model does not support these relationships, new relational operations were introduced for defining and accessing the records of nested files.

The concept of file nesting, however, is not new; it is practically identical, in fact, to the way we relate files when using the traditional file operations (see pp. 683–686). The only real difference is the higher level of abstraction of the non-1NF operations. What this means in practice is that, instead of creating explicit file scanning loops like those in figures 7-15 and 7-16, we invoke some built-in functions that generate the loops for us.

But, as we know, a higher level of abstraction also has drawbacks: we are restricted to fewer alternatives. So in the end, even with non-1NF, the relational systems are not as flexible or efficient as the basic file operations. For example, with the basic operations we can nest – in different places in the application, through different fields – the same files in different ways; we can create, therefore, several relationships between the same files. Also, with the basic operations we still have the option of storing arrays directly in a record – a method that is both simpler and faster than file nesting.

The main objection to non-1NF, however, is that it is presented as a new feature while being an abandonment of the relational file-relating method and a reinstatement of the traditional one. Even the term "nesting" is old: with the traditional operations, the files are nested by nesting their scanning loops; with relational systems, the files are nested through *implicit* scanning loops. The logical relationship between files is the same.

❖

The term "non-1NF," then, is not only silly but also misleading. For, the intent of the new feature is not to avoid the problems caused by the 1NF principle, but to replace the impractical JOIN operation. Let us examine this misrepresentation more closely.

To promote non-1NF, the experts point to the inefficiency of certain file

combinations in the original relational model. But the combinations they describe were never thought to be a consequence of the 1NF restriction. Specifically, non-1NF is recommended for files of any size, not just as a substitute for the small arrays that we may want to store directly in a record. Thus, referring to the earlier examples, we can use file nesting not just to replace an array of three vendors associated with one part, but also for a whole invoice file, where hundreds of invoices may be associated with one customer. For this type of data, though, we have *always* resorted to a second file, even with the traditional file operations, because this is the only practical way to store it. The difference between non-1NF and 1NF, then, is simply in the way we combine files: through nesting instead of JOINs. So what the experts are recommending in reality is not the replacement of 1NF with non-1NF, but the replacement of JOIN operations with the traditional concept of file nesting.

Non-1NF, in other words, is not promoted as a solution to the inefficiency of 1NF, but as a solution to the inefficiency of JOIN; that is, for any situation where we have to combine files. Thus, if we adopt non-1NF we can dispose of the JOIN operation altogether. If we want, we can replace with nested files every situation that would normally require JOINs: not just files that would be created when enforcing the first normal form, but also files that would be created when enforcing the second and higher normal forms, and even files that would be kept separate in any case. Non-1NF eliminates, therefore, the inefficiency caused by combining *any* files in a relational database. So, if it is a general substitute for the relational way of combining files, what we have now is *a different database model.*

Far from being just a new feature, then, non-1NF cancels the whole relational model. To understand this, let us take a moment and recall the importance of the first normal form. And there is no better way to start than by citing the experts themselves.

Date says that 1NF is so fundamental that the term "normalized," when unqualified, means "first normal form": "It follows that *every* normalized relation is in first normal form …; it is this fact that accounts for the term 'first.' In other words, 'normalized' and '1NF' mean *exactly the same thing.*"[3] In Codd's original papers, too, the term "normalized" means what we call now first normal form;[4] the higher normal forms are not even mentioned. Recall also that the first normal form is the only one that is part of the *formal* relational model.

[3] C. J. Date, *An Introduction to Database Systems*, 6th ed. (Reading, MA: Addison-Wesley, 1995), pp. 289–290.

[4] See, for example, E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* 13, no. 6 (1970): 377–387.

Here are some additional statements: "At each intersection of a row and column there is exactly one value. This is the principle of *first normal form*, fundamental in the relational model."[5] "This property implies that columns do not contain repeating groups. Often, such tables are referred to as 'normalized' or as being in 'first normal form (1NF).' It is important that you understand the significance and effects of this property because it is a cornerstone of the relational data structure."[6] "Occasionally there might be good reasons for flouting the principles of normalization.… The only hard requirement is that relations be in at least first normal form."[7] "All data in a relational database is represented in *one and only one way*, namely by explicit value (this feature is sometimes referred to as 'the basic principle of the relational model' …). In particular, logical connections within and across relations are represented by such explicit values."[8]

It is not difficult to see why the first normal form is so important to the relational model – why it is "fundamental," a "cornerstone," a "hard requirement," and a "basic principle." It is not so much the restriction to single values that is important, as the *purpose* of this restriction. By preventing us from creating any data structures within a record, 1NF forces us to keep all data in the form of tables. And if the data is restricted to tables, the methods used to access and combine the data can be restricted to operations on tables; that is, to high-level operations based on mathematical logic.

Accordingly, by annulling 1NF we also annul these restrictions: we can store, access, and combine data in other ways too. In effect, we have regained some of the freedom we enjoyed when using files through the traditional file operations: we can now relate them through the versatile hierarchical concept, as data within data. And we can use this method, not just with small arrays or structures, but with files of any size, and on any number of nesting levels. In the end, annulling 1NF permits us to create database structures that are more flexible and more efficient than those possible with the relational model.

In conclusion, the restriction imposed by the first normal form is far more significant than what it appears to be – merely preventing multiple values in a field. Its annulment, therefore, means far more than just permitting multiple values; it means the annulment of the relational model. It also demonstrates the pseudoscientific nature of this theory, as well as the dishonesty of its supporters: the impracticality of 1NF, along with the impracticality of JOIN, is

[5] Anthony Ralston and Edwin D. Reilly, eds., *Encyclopedia of Computer Science*, 3rd ed. (New York: Van Nostrand Reinhold, 1993), p. 1162.

[6] Candace C. Fleming and Barbara von Halle, *Handbook of Relational Database Design* (Reading, MA: Addison-Wesley, 1989), pp. 32–33.

[7] Date, *Database Systems*, p. 291. [8] Ibid., p. 99.

a falsification of the model; but instead of being abandoned, the theory is expanded – by turning this falsification into a new relational feature, non-1NF.

❖

The mathematical foundation of the original model was predicate calculus, with its relations and tuples. Thus, if our databases no longer consist of this type of relations and tuples, it is absurd to continue to call them relational. Terms like "extended relational" and "nested relational" are simply incorrect if the new model is not "relational." The term "relational" derives from the mathematical concept of a relation; namely, a set of tuples, where each tuple is composed of single elements. And in predicate calculus the only operations are those performed on such sets through mathematical logic. It is these relations, tuples, and elements that become the files, records, and fields of a relational database. So, if we want a different database organization, or different operations, we need a different model.

As we saw under the first delusion, the mathematical claims of the relational model were tenuous in any case, since only a small part (those aspects that constitute the formal model) had indeed a mathematical grounding. And with the annulments we are discussing in this subsection – non-1NF, in particular – even that small part has disappeared. What we have now is neither an enhanced nor an extended relational model. What we have is not a relational model at all.

Non-1NF systems, then, are indeed as useful as their promoters claim; but they are useful because they are no longer relational. This is why some experts, embarrassed perhaps by this fraud, suggest terms like "post-relational," and even "object-relational," for the database systems that include non-1NF or a similar "enhancement."

Still, if not predicate calculus, perhaps another mathematical system can serve as a foundation for the new database model. And indeed, some theorists have attempted to extend the formal relational model to include non-1NF. But this is silly. For, if a mathematical system could guarantee the correctness of nested databases, then the same system would also guarantee the correctness of the nesting performed through the traditional file operations – which is identical, logically.

This, of course, is true for the original model too: nothing stops us from using the traditional data files and operations while limiting ourselves to the subset of features that parallel the relational model; and our databases would then be founded on predicate calculus, just like the relational ones.

The conclusion must be that, no matter how rigorous a formal database model is, it offers no mathematical benefits that we do not also enjoy with the

informal traditional operations.[9] The answer to this apparent contradiction is that the formal part plays such a small role in a database system that it is practically irrelevant. So, for the application as a whole, the mathematical benefits are about the same with a formal database model as they are without one. (This is the essence of the first delusion.)

❖

By way of summary, I want to show how the software elites are presenting the non-1NF feature. A good example is the white paper published by IBM to promote one of their new database systems.[10] This paper, we are told, "discusses technical advances represented by nested relational database technology."[11] And just in case we were not sufficiently impressed by this statement, a few sentences later we are reminded that nested relational databases represent an "advanced technology."

Now, the advanced technology that is file nesting has been available since about 1970 to anyone capable of writing a few lines of COBOL. So it is clear that IBM addresses individuals who, while being perhaps programmers or managers, have very little programming knowledge. These incompetents try to develop applications, not through programming, but by buying programming substitutes. They can be impressed by a feature like non-1NF because they are always dependent on the elites for solutions to their software problems. They have problems now because they trusted the elites in the past and adopted a relational system. But they believe that the solution must also come from the elites, in the form of a new system.

The paper continues by describing the problems caused by the restriction to 1NF: "Database conformance with 1NF often increases the amount of storage used, makes maintenance more difficult, and most importantly greatly increases the processing required to produce results, while still making the schema more complex.... For some potential users of relational databases, the joins [i.e., JOIN operations] that would be required to resolve relationship relations [i.e., cross-references] in 1NF databases would affect performance

[9] Because they are restricted to higher levels, the relational operations are logically a subset of the basic, traditional file operations. Thus, we can always simulate a relational database system using a traditional file system, but not vice versa. Many relational systems, in fact, are designed simply as a high-level environment based on an underlying file management system: the relational operations are implemented as subroutines that employ the basic file operations in conjunction with appropriate loops and conditions.

[10] IBM Corporation, *Nested Relational Databases*, white paper (2001).

[11] Ibid., p. 3. Note, again, the slogan "technology," used to make something appear more important than it really is.

enough to preclude the use of relational databases.... Apart from performance considerations, 1NF relational databases also have practical limitations for many applications."[12]

This is an excellent description of the restrictions imposed by the first normal form, and by the relational model in general. Reading this, one is liable to forget that the same institutions that are so harshly attacking this model now had been promoting it for the previous thirty years as an expression of database science, and as an important aspect of software engineering. These problems had been noticed from the start, of course. So how were the millions of programmers and users who had adopted relational systems coping all these years? By constantly seeking ways to bypass the restrictions; by spending most of their time dealing with these spurious problems instead of the actual business problems; and, ultimately, by being content with inadequate and inefficient applications.

The nested relational model, the paper tells us, eliminates the 1NF problems. Non-1NF is such an important feature, in fact, that all relational systems will soon support it: "Because of the limitations of 1NF relational databases, especially for storing complex data structures, all commercial relational databases have begun adopting extended relational technology; however, IBM has a technological lead of several years over its closest competitor."[13]

The shallowness of the non-1NF issue is seen in the pretentious description of file nesting. For example, one of the reasons why IBM's "extended relational technology" is more advanced than the competing ones is that "the IBM nested relational implementation, unlike others, is not limited to a single nested table."[14] With the basic file operations, as we know, it is just as easy to nest several file scanning loops as it is to nest one, simply because programming languages allow us to combine file scanning loops in any way we like. But with nested relational databases, this trivial capability is presented as a major technological advance, currently available only from IBM. Again, only ignorant practitioners can be impressed by such claims.

Finally, the paper reminds us (three times[15]) that the relational model has a rigorous mathematical foundation, which guarantees correct results when using the relational operations. And, the paper assures us, research has shown that this guarantee is not compromised by the annulment of the 1NF principle: "Analysis has proven that the resulting model is equally robust."[16] Such analysis and proof are senseless, though, because the relational model is *not* robust even *with* 1NF. As we saw under the first delusion, its mathematical foundation is irrelevant in practice. It is precisely because the mathematical foundation is

[12] Ibid., p. 7. [13] Ibid., p. 14. [14] Ibid. [15] Ibid., pp. 3, 7, 14.
[16] Ibid., p. 7. The paper cites several sources, where presumably the proof can be found.

irrelevant that annulling an important principle like 1NF indeed makes no difference.

And we are expected to feel even better after reading that nested relational databases have been "accepted by the academic community as adhering to a valid relational model."[17] But we saw that it is wrong even to *call* the new model relational. In any case, this statement is hardly reassuring if we remember that the same academic community also advocated other theories that failed (structured programming and object-oriented programming, in particular), and that, just like the relational model, those theories were rescued by being turned into pseudosciences.

Thus, by promoting pseudoscientific software theories, the universities help software companies to sell worthless development systems, and help incompetent programmers and managers to control corporate computing.

## 3

I began the discussion of the third delusion with the non-1NF issue because this is the most flagrant of the relational reversals – a reversal that marks, in effect, the end of the relational theory. But 1NF is merely the latest principle to be annulled. At this point, most relational principles had already been forsaken, because, like 1NF, they had been found to be impractical. In the remainder of this subsection, I propose to study the other reversals.

The abandonment of the second and higher normal forms (2NF, 3NF, etc.) came by way of a new relational principle, called *denormalization*. At first, database designers and programmers simply ignored the stipulation to fully normalize their files, when this was too complicated or too inefficient. But the theorists were condemning this practice. Before long, though, even they realized that strict normalization is impractical, and that the decision whether or not to normalize a particular set of files depends ultimately on the situation: on the type of data stored in these files, on the file relationships, and on the way we plan to use the files in the application.[18]

But instead of admitting that the idea of strict normalization had failed, the theorists reacted, as pseudoscientists do, by turning this falsification of the

[17] Ibid., p. 14.

[18] The term "normalization" refers usually to *all* normal forms; but here, in the discussion of the second and higher normal forms, I use "normalization" to refer only to them.

relational model into a new relational principle – denormalization. The new principle says that we must first normalize all files, as before; then, we must denormalize (that is, restore to their previous state) those files that should not have been normalized in the first place.

Both the principle and the term, "denormalization," are absurd. All we needed was a statement acknowledging that normalization was annulled as a relational principle and is now just an informal concept. The very term "normalization" should have been abandoned, in fact. After all, normalizing some files and not others is what we had been doing all along, with the traditional file operations, and we didn't need a special term to describe this activity. With the relational model we have now *two* principles for this activity, and *two* terms. We are told that normalization is as important as before, and that denormalization is the process of *improving* the results of normalization.

Clearly, the theorists invented the second principle in order to suppress the fact that the first one had failed. The two principles, normalization and denormalization, in effect cancel each other. But the theorists managed to make this return to what we had before the relational model look like an *enhancement* of the model.

Here is a typical explanation of the new principle: "Denormalization is the 'undoing' of the normalization process. It does not, however, imply omission of the normalization process. Rather, *denormalization* is the process whereby, after defining a stable, fully normalized data structure, you selectively introduce duplicate data to facilitate specific performance requirements."[19] What this sophistic verbiage is trying to say is that, while normalization is generally desirable, strict normalization is impractical; in other words, what we always knew. Now, however, we can no longer simply allow some data duplication from the start (when we know from experience that the application would otherwise be too slow). Instead, we must first normalize the whole database, and then "selectively introduce duplicate data to facilitate specific performance requirements." Actually, in both cases we address the same problem and end up with the same database. The pompous language serves to mask the fact that the principle of strict normalization – a fundamental relational requirement – has been falsified.

Here is how two other experts present this reversal: "The general idea of normalization is that the database designer should aim for relations in the 'ultimate' normal form (5NF). However, this recommendation should not be

[19] Fleming and von Halle, *Relational Database Design*, p. 440.

construed as law. Occasionally there might be good reasons for flouting the principles of normalization."[20] "There are, however, exceptions to [strict normalization].... We recommend that data models *always* be designed in third normal form, but that the physical data-base designer be permitted to deviate from it if he has good reasons and if the data administrator agrees that no serious harm will be done."[21]

A critical aspect of the idea of denormalization, then, and what the experts keep stressing, is that denormalization does *not* constitute the annulment of normalization. Normalization remains as important as before, and what we must do is both normalize and denormalize the database.

Here is another example of this doubletalk: "Data denormalization is constrained so that it does not alter the basic structure of the conceptual schema. It only makes adjustments to the basic structure for operational efficiency."[22] Denormalization, thus, consists in *adjusting* the database design, but without *altering* it. This is silly, of course, since adjusting something will also alter it. A database either is or is not normalized; so, if we denormalize a normalized database we necessarily end up with an unnormalized one, regardless of whether we call this process "adjustment" or "alteration." Not so, says Brackett: "A common misconception about data denormalization is that it results in a return to the unnormalized business schema that began the data normalization process.... However, this is not the situation. Data denormalization produces denormalized data, not unnormalized data."[23] In reality, there is no difference between the two: both "denormalized" and "unnormalized" mean simply data that is not fully normalized, violating therefore this relational principle.

The theorists, thus, are defending their deviation from strict normalization by claiming that denormalizing the database after fully normalizing it is different from simply leaving some of the files unnormalized in the first place. One method, they tell us, constitutes an exact design process, while the other is merely an informal decision. But this would be true if denormalization were indeed an exact process. In practice, though, the decision to denormalize a file can be no more exact than the decision to leave a file unnormalized to begin with. Recall the previous quotations: "[the designer is] permitted to deviate from [strict normalization] if he has good reasons and if the data administrator agrees that no serious harm will be done," and "occasionally there might be good reasons for flouting the principles of normalization."

[20] Date, *Database Systems*, p. 291.

[21] James Martin, *Managing the Data-Base Environment* (Englewood Cliffs, NJ: Prentice Hall, 1983), p. 216.

[22] Michael H. Brackett, *Practical Data Design* (Englewood Cliffs, NJ: Prentice Hall, 1990), pp. 155–156.

[23] Ibid., p. 156.

Informal comments like these can hardly be described as an exact method of denormalization.

Brackett starts by promising us an exact method: "Conceptual schema are converted to internal schema through a denormalization process following a precise set of rules depending on the physical operating environment."[24] But the "precise set of rules" never materializes. All we find on the subsequent pages is a list of cases where denormalization is beneficial, and a reminder to deal carefully with the consequent problem (redundancy and inconsistencies). For instance, this is how Brackett describes one of the cases of denormalization: "This situation creates redundant data and those redundant data must be consistently updated or the quality of the database will deteriorate rapidly.… Other data entities may be denormalized for operational efficiency based on these criteria.… Each situation must be carefully evaluated to assure that the logical model is not compromised and that any redundant data are routinely and consistently updated."[25]

So what Brackett is describing as denormalization is not "a precise set of rules" but an informal process – a process no different from what we do with *traditional* databases: we study the application's requirements, allow redundancy and inconsistencies when it is impractical to eliminate them, and deal with the consequent problems by adding special checks and operations to the application's logic.

Thus, to cover up the failure of strict normalization, the theorists were compelled to invent the absurd principle that we must first normalize the database and then denormalize it. And they defended the principle with the absurd claim that this method is exact while the traditional, simpler method – creating the correct database directly from the requirements – is not. In reality, both methods entail the same decisions and result in the same design.

We saw under the second delusion that the process of normalization is presented by the theorists as a formal design method, while being in fact as informal as the traditional method. It is informal because it must be based, ultimately, on the same decisions as those we make when designing the database directly from the requirements. Now we see that the process of denormalization too is informal, despite the claims that it is exact. Only we, by studying and interpreting the requirements, can determine whether strict normalization is practical in a given situation, and, if not, what operations must be added to maintain data integrity.

[24] Ibid., p. 155. [25] Ibid., pp. 157–158.

In conclusion, both normalization and denormalization are perceived as formal design methods, when in fact both are informal. So, to appreciate the new delusion, denormalization, we must ignore the previous one: we must believe, with the theorists, that normalization is indeed an exact process. Judging it from *their* perspective, therefore, denormalization is a delusion; for they did not stop promoting normalization when they introduced the concept of denormalization. They continue their research in what they believe to be formal and exact concepts – the dependency theory, the classification of normal forms – even while praising the virtues of denormalization, which is informal. They are oblivious to the absurdity of promoting these two methods at the same time: no matter how exact is the process of normalization, when we modify its result by adding the inexact process of denormalization the final result is bound to be inexact. So what is the point in seeking a formal and exact normalization theory while also permitting denormalization?

It is in order to resolve this self-contradiction that the theorists introduced the principle that we must denormalize the database only after fully normalizing it. This principle appears to justify the need for both processes, when in reality it shows that we need neither.

Earlier, to justify the need for normalization, the theorists distorted the problem of database design. Instead of determining the correct design simply by studying the application's requirements, we were asked to do two things: create a deliberately incorrect database, and then normalize it to make it correct. And now, to justify both normalization and denormalization, we are asked to do *three* things: create an incorrect database, normalize it to make it correct, and, finally, denormalize it to make it practical.

The traditional design method allows us to create, not only correct databases, but also efficient ones. For, the same skills that help us to create a correct, fully normalized database also help us to decide when this would be inefficient. Thus, we can create a correct *and* efficient database at the same time, directly from the requirements. We don't need a denormalization theory any more than we need a normalization one.

Finally, and quite apart from the delusions already discussed, the need for denormalization means that we are again preoccupied with the *efficiency* of the database operations – contrary to the claim that the relational model shields us from the physical implementation of the database. We must study each situation and seek the most effective solution, instead of implementing the requirements through formal methods and high-level operations, as the relational theory had promised us. We must accept, rather than avoid, the "update anomalies"; and we must add special checks and operations to deal with them. In other words, we have returned to what we had been doing all along, with the traditional databases. The theorists describe denormalization

as database "optimization"; but if the optimization consists in a deviation from fundamental relational principles, this description is merely a way of denying that the relational model has failed.

## 4

One of the relational model's promises was that we could restrict ourselves, in all database work, to the high-level relational operations. And this promise too had to be annulled. In the end, the relational systems became practical only after reinstating the low-level capabilities of the traditional file operations; specifically, the means to manipulate fields and records through traditional programming methods, and the means to link them to other low-level entities in the application. Let us examine this reversal.

Recall the original relational model. The database, we are told, must be perceived as "tables and nothing but tables." The relational operations can be assumed to occur instantaneously, and can therefore be treated like the operations of mathematical logic: all we have to do is reduce the database requirements to logical expressions where the operands are tables, and the relational operations (along with standard logical operations) combine in various ways tables and portions of tables. No matter how large or how small, a data file can be treated simply as a table with a number of rows and columns. In particular, if we need just one record of a given file, we must create a new table with just one row; and if we need one field, we must create a table with one row and one column. Also, there is no way to modify the tables. Updating the database was thought to be a relatively simple and infrequent aspect of database work, so the operations that add, delete, or modify records were expected to be informal, like the traditional ones. We must be careful when modifying the database, of course; but we don't need the formality and precision of mathematics, as we do for queries.

The original model, thus, permits only database queries. Consequently, the only database language we need is one that provides the means to formulate queries through the relational operations. In their naivety, the theorists believed that a model shown to satisfy some simple queries on small files could serve as the foundation of practical database systems: for applications with files of any size and queries of any complexity. Moreover, they later believed that the same model could be extended to cover all aspects of database work, including the updating operations and the design process. The fact that simple queries look neat when expressed as mathematical logic was enough to convince the theorists that all database programming could be restricted to high-level operations and to the notion of tables.

Today, after all the reversals, the relational systems are no longer restricted to "tables and nothing but tables." Rather, they provide, in a very complicated manner, the means to link individual fields and records to the other entities in the application. In addition, the database language, SQL, has grown from a set of simple query operations into an elaborate (although quite primitive) programming language. The relational systems, thus, have restored the means to manipulate, through programming, the low-level database entities. So they have restored exactly what the traditional file operations and programming languages had been doing all along, in a much simpler way – and what the original relational model had claimed to be unnecessary.

❖

We need to access low-level database entities for two reasons: because this is the only way to implement the *details* of a database operation, and because this is the only way to *link* the database structures to the other structures that make up the application. It is obvious, therefore, why the traditional file operations are indispensable: in addition to allowing us to access the low-level database entities, they can be used from a programming language; and through this language we can create the critical, low-level links between database entities and the other types of entities in the application.

These two qualities are both necessary and sufficient for implementing any database requirement; and it is precisely these two qualities that are lost in the relational model. Thus, since it is impossible to implement serious applications without accessing and linking the low-level elements of the application, it is not surprising that the modifications needed to make the relational systems practical consisted in restoring both the low-level operations and the means to use these operations through a programming language.

So, like all systems that offer us high-level starting elements, the relational systems became in the end a fraud. When promising us higher levels, the software charlatans tempt us to commit the two mechanistic fallacies, reification and abstraction (see "The Delusion of High Levels" in chapter 6). In the case of relational systems, the claim was that we could separate the database structures from the other structures that make up the application; this would allow us to start from higher levels of abstraction within these structures, greatly simplifying database work.

With the traditional development method, all we need is a programming language and a few libraries of subroutines (for mathematical functions, display operations, database management, and the like). The software charlatans have replaced this simple concept with the concept of *development environments*: large and complicated systems that lure ignorant practitioners with the promise

of high-level, built-in operations. These operations, we are told, function as prefabricated software subassemblies: they already contain within them many of the low-level operations that we would otherwise have to program ourselves. But, in fact, only trivial requirements can be implemented by combining high-level operations. So the systems must be continually enhanced, with more and more features. And what are these features? They are means to deal with low-level entities, precisely what the systems had originally attempted to eliminate.

Thus, instead of admitting that the restriction to high-level operations failed as a substitute for traditional programming, the software charlatans rescue these systems by turning their falsifications, one by one, into new features. The systems keep growing and appear to become more and more "powerful," but this power derives from reinstating the low-level, traditional concepts. By the time enough of these concepts are reinstated to make the systems practical, there is nothing left of the original promise. For now we must deal with the low levels again. What is worse, because the low levels were introduced within the high-level environment, they are much more complicated than they are when available directly, through a traditional language. So, in the end, programming is even more difficult than before.



Returning to the relational systems, the need for low levels emerged when the notion of *data integrity* was introduced. Data integrity became an issue in the relational model only when the model was expanded to include *updating* operations. As long as it permitted only queries, there was no need for integrity checks, because, within the scope of the model, the data never changed. Since the operations that add, delete, or modify data records were expected to be similar to the traditional ones, and to be performed outside the model, the validity checks accompanying these operations were also expected to be performed outside the model. Once the relational model was adopted for serious database work, however, the updating operations, along with the problem of data integrity, could no longer be ignored.

The normalization principles too, we saw earlier, were needed only when the relational model was expanded to include updating operations. Tables did not have to be normalized in the original model, because no data inconsistencies can arise when we restrict ourselves to queries and to the high-level relational operations. We also saw how both the attempt to formalize the process of normalization, and the idea of strict normalization, failed. In the end, the only way to design a correct database is informally, by studying the application's requirements. All that the theory of normalization accomplished was to add to the traditional design problems the complicated concepts of normal forms

and field dependencies. The critical part – the need to determine whether two given fields must be in the same file or in separate files – remained unchanged. With the traditional design method or with the relational one, we can decide in which file to place a new field only by discovering the low-level links between the database entities and the other entities in the application.

The formality and the neat classification of normal forms can be seen, therefore, as a failed attempt to raise the level of abstraction in database design: instead of having to study and understand the application's requirements, it was believed that we could attain the same goal by knowing only how to convert files from one normal form into another – an easier, largely mechanical, task.

But regardless of its failure, the normalization theory was silly because it addressed only a small number of data inconsistencies; specifically, only those that can be prevented by placing fields in separate files (see pp. 755–757). Since most data inconsistencies cannot be eliminated simply by separating fields, we must deal with them through the application's logic: to ensure that an updating operation does not cause inconsistencies, we add various checks, restrictions, or further updating operations. An example of a situation where the updating problems cannot be solved through normalization, we saw, is the requirement for the balance field in the customer record to match at all times the amounts present in that customer's transaction records. Although technically redundant, the balance field is useful because it obviates the need to recalculate the balance by reading the transaction records every time. Thus, instead of avoiding the redundancy, we ensure that the field remains correct by adding to the application's logic some operations to update it whenever a transaction is added, deleted, or modified.

The requirement to match the balance field and the transaction records is, in effect, a database integrity rule. So the notion of integrity was the answer to the updating problems that could not be solved through normalization; that is, to practically *all* the updating problems that can arise in an application. A whole new class of relational features had to be invented – features totally unrelated to the original model – in order to move the data validity operations from the application, where they are normally performed, into the database system. The sole purpose of these features is to permit us to do through a new language, in the database system, what we had been doing all along through a traditional language in the application. Thus, the operations that update the customer balance field, previously mentioned, would no longer be part of the main program; they would be written instead in a special language, and made part of the database environment.

The problem of data validity is well known. Whenever a database field is modified, the application must verify that the new value is correct within the current context. Similarly, when a record is added, the value of each field in the record must be correct. But there is more to the validity problem than verifying the value of individual fields. For example, the application must verify that a record may be modified at all, or added or deleted, in a given situation. Also, adding, deleting, or modifying a given record often affects other records and other files, so the application must perform additional operations if the database as a whole is to remain correct. Generally, all the specifications and restrictions known as business rules – which are reflected in the various processes implemented in the application – can be described, if we want, as integrity rules.

Data validity, thus, is closely related to the application's requirements: what is correct in one situation may be incorrect in another. Just like the "anomalies" they tried to eliminate through normalization, the problems that the relational theorists are discussing under "integrity" are problems we always faced. And we never thought of them as *database* problems, but as a natural part of application development. The so-called integrity problems are merely one aspect of the challenge of programming: if we fail to take into account certain requirements, some data may become incorrect – inconsistent, redundant, invalid – when the application is used. The problems that cause incorrect data are similar to those that cause incorrect operations. In both cases the application will malfunction, and in both cases the reason is that it does not reflect the requirements accurately.

We saw earlier that files cannot be said to be normalized in an absolute sense, but only relative to the application's requirements. For example, if the product description does not change from one order to the next, the product and orders files are normalized when the description field is in the product file; but when the description may change, they are normalized when the description field is in the orders file. Similarly, the validity criteria cannot be defined in an absolute sense, but only relative to the application's requirements. Some examples: A certain date may be deemed too old in one part of the application, but not in another. Deleting a transaction record may be permitted if certain conditions hold, but deemed invalid otherwise; elsewhere in the application, though, we may have to prevent the deletion under all conditions. Creating a new transaction record may generally entail adding a record to the history file too, and failing to do so would result in an incorrect history file; sometimes, though, when this is not a requirement, it is *adding* the history record that would result in an incorrect file.

Clearly, validity issues like these are part of the application's logic. It is absurd to treat them as a special class of operations just because they are

concerned with the correctness of database entities. We also modify *memory* variables in the application, and they too must remain correct; yet no one has suggested that – in order to safeguard the correctness of memory-based entities – we extract these operations from the application, restrict them to high levels, and design special systems and languages to perform them. If we were to do this for every type of entities and operations, we would no longer need applications and general-purpose languages. Performing and combining various types of operations, including those concerned with data validity, is precisely what applications are for, and what programming languages are designed to do. In any case, the operations that validate the database, as much as those that modify it, must necessarily access *low-level* entities. So the idea of separating them from the application, incorporating them into a database system, and restricting them to high levels is senseless, and bound to fail.

In conclusion, the integrity features added to the relational database systems were totally unnecessary. Their real purpose was to rescue the relational model from refutation. Here is how: The promise had been a model that satisfies all our database needs through high-level operations. The existing data validity functions, however, required *low-level* operations. Moreover, they required *programming*, so they could not be implemented at all in a relational system. Asking us to depend on traditional programming for a critical aspect of database management was, thus, a falsification of the relational model. To save the model, the theorists were compelled to move these functions *from* the application, where they belong naturally and logically, *into* the database system. The integrity features are a fraud because this move is said to complement the high-level database operations, when in reality the new functions require low levels, and programming.

The integrity features, then, were the expedient through which low-level capabilities could be added to a relational system. Instead of recognizing the need to deal with low-level entities as a falsification, the theorists solved the problem by annulling the restriction to high-level operations. Using the issue of data integrity as pretext, they turned a blatant falsification into an important new feature. This feature is so important, in fact, that no serious database requirements can be implemented without it. And all this time, they kept praising the power of the relational model: annulling the restriction to high levels, they say, is an *enhancement* of the model.

The first integrity functions were limited to simple validity checks. Here are some examples: The *attribute integrity* functions check that the value placed in a field is correct with respect to the definition of the field (valid numbers in a

numeric field, valid dates in a date field, etc.). The *domain integrity* functions check that the value placed in a field is correct when the field is restricted to a range of values (a number must not be larger than 1,000, for instance, a date must not be older than 30 days ago, etc.). The *referential integrity* functions check that the relationship between two files remains correct when the files are modified; typically, they are used to prevent the deletion of a record in one file while there exist records in the other file related to it through their key.

To use an integrity function, the programmer specifies the event that is to invoke the function at run time (this event is known as *trigger*), the conditions and values that make up the constraint, and the action to take in case of error (display a message, prevent a change or deletion, etc.). Triggers may be included in the application when a certain field is modified, after a record is added to a certain file, before a record is deleted, and so on.

Validity checks like these can be easily implemented in the application, of course, using the conditional constructs or exception-handling features available in most languages. So it is not at all clear why a database system must provide these checks in the form of built-in functions. Still, if we agree that higher levels of abstraction are sometimes beneficial, these functions do provide a good alternative for specifying and enforcing certain validity criteria.

Only simple checks, however, can be implemented through standard, built-in functions. This is true because all we can do in a standard function is specify a few conditions and values and the action to take, while most integrity checks entail *combinations* of conditions, values, and actions. Thus, the checks we need in a typical application may affect several fields and files, may require a unique piece of logic, and may need some data that resides in the application, not in the database.

So, like all high-level operations, the concept of standard integrity functions can be useful if provided as an *option*, to be employed only when better than *programming* the same checks. The relational theorists, though, hoped to turn *all* integrity checks into standard functions. Their naivety is betrayed by their attempt to *classify* the integrity functions – referential integrity, domain integrity, and so forth. They actually believed that they could discover a set of standard functions that would encompass all conceivable data validation requirements (or, at least, the most common ones). Note also the pretentious names they invented to describe what are in reality simple operations. Clearly, they believed that the concept of built-in integrity functions represents an important contribution to database science. But preventing the deletion of a particular record, or ensuring that a field's value lies within a certain range, are operations we routinely perform in every application, using ordinary programming languages; and we don't need scientific-sounding terms like "referential integrity" and "domain integrity" to describe them.

The concept of built-in integrity functions failed, of course. After devising a few standard functions, the theorists had no choice but to give us the means to create freely our own functions, which is the only way to satisfy real-world data validation requirements. And, since it is only through programming that one can create such functions, new programming languages were invented – languages whose only purpose was to allow programmers to implement these functions as part of the database, rather than part of the application. Then the languages started to grow, as programmers demanded greater functionality. Means were introduced to perform calculations, to create flow-control constructs, to call subroutines, to access memory variables, to use general-purpose function libraries, and so forth. These languages provided, in other words, more and more of the very same features that were already available in the *traditional* languages.

No one noticed the absurdity of this situation. *Programming* our own functions is an alternative we always had. The promise had been, not a new language, but a higher level of abstraction. And if this turned out to be impossible, the theorists should have admitted that the only way to implement versatile data validation is through a programming language – the way we always did – and encouraged us to return to the traditional methods. What is the point in inventing specialized languages, indicating by means of "triggers" where in the application we need the integrity functions, and placing the functions in the database, when we could simply keep them as part of the application? After all, despite the multiplying features, the new languages remain inferior to the traditional ones, even in the narrow domain of database work that is their specialty. So programmers must now assimilate and depend on some new languages without deriving any real benefits. In the end, not only do the relational systems fail to provide the promised higher level of abstraction, but they make the task of data validation more complicated than before.[26]

The theorists, of course, could not admit that the concept of high-level integrity functions had failed, and that we must return to the traditional methods, because this would have been tantamount to admitting that the relational model had been falsified. So, inventing new languages was the way to cover up this falsification. Imagine an application written in COBOL, and a database system that asks us to write the data validation functions also in COBOL, but to store them in the database. Since we know that we can

[26] We hear sometimes the argument that storing the integrity functions outside the application facilitates the implementation of corporate standards, as all validity criteria are specified in one place. But this argument is tenuous. First, we can accomplish the same thing with ordinary subroutines. Second, even with the functions outside the application, why do we need new languages?

accomplish the same thing by making those functions an integral part of the application, we would reject the database alternative as absurd. If, however, it is not in COBOL but in a new language that we must write these functions – and if the language is accompanied by some new and impressive terminology, and if it is provided through a large and intricate development environment – the absurd alternative can be made to look like an important programming concept. And if we add to this the enthusiasm of the experts and the media, and the urgent needs of the companies that already depend on relational systems, everyone would perceive this concept as progress. Thus, what is in reality a *falsification* of the relational model is made to appear as an *enhancement* of the model.

Reinstating the programming capabilities, then, is what made the relational systems practical. *All* relational principles had to be annulled, as we saw earlier; but the other annulments would have amounted to nothing had the restriction to high-level database operations been maintained. The idea of programmable integrity functions was so well received because it provides the means to bypass the restriction to relational operations. Although not as useful as the traditional file operations, the operations available through the new languages do have similar capabilities. So they allow us to implement many database requirements that would be impractical through relational operations alone.

Thus, in the guise of integrity functions, programmers could now add to their applications a great variety of *low-level* file operations. Whenever a database requirement was too complicated or too inefficient to express through the relational operations, they could program it in the form of an integrity function and define a "trigger" in the application to invoke it. After all, with a little imagination any database requirement can be associated with some integrity checks or rules. For example, if we have to modify a record in such a way that a field's value is the result of calculations and conditions involving some other files and some memory variables – a task impossible or impractical through relational operations – we can program all this in a database language and call it an integrity function.

Understandably, this stratagem was very popular. Programmers praised the virtues of the relational model, but resorted to "integrity" functions and "triggers" whenever the requirements called for low-level file operations, or low-level links between database entities and other types of entities; in other words, whenever they needed the capabilities of the *traditional* file operations. They appeared to like the relational model, but what they liked in reality was the new, low-level capabilities – which contradict the relational principles.

In the end, all pretences of integrity and triggers were discarded, and these functions were expanded into the broader concept known as *stored procedures*. These procedures are general-purpose pieces of software that can be employed freely in the application. They are stored in the database, but are used like ordinary subroutines: they can be invoked from the application or from other stored procedures, can have parameters, and can return values. And, since there is no limit to the size or number of stored procedures, larger and larger portions of the application were being developed in this new fashion, in order to take advantage of the low-level capabilities of the database languages. Thus, while programmers were convinced that they were using the relational model, their applications resembled more and more those developed with the traditional languages and file operations.

So, by allowing programmers to bypass completely the relational principles, the concept of stored procedures was the final answer to the need for low-level file operations and low-level links to the other aspects of the application.

## 5

Although there are many relational database languages, it is SQL that became the official one. And it is through SQL, more than through the others, that the fraud of reinstating the traditional concepts was committed. Today's relational systems would be unusable without SQL. From its modest beginning as a query language, SQL has achieved its current status, and has grown to its enormous size, as a result of the enhancements introduced in order to provide programming and low-level capabilities – precisely those capabilities that the relational model had claimed to be unnecessary. Thus, today's official relational language is in reality the official means, not of *implementing*, but of *overriding*, the relational principles. Let us study this evolution.

The original relational model, we recall, was meant only for queries. And SQL (which stands for Structured *Query* Language) was the language through which programmers and users alike were expected to access the database. The original SQL, thus, allowed us to select and combine subsets of tables by specifying various criteria in the form of relational operations.

The SQL statement for queries is SELECT. This one statement, however, contains many clauses, which permit us to specify various details: the files involved in the query, the operations required to relate these files, the sorting sequence, the record selection criteria, which fields to display, and how to group the selected records for showing subtotals and the like. Thus, while neat and straightforward for trivial queries, a SELECT statement can become very long and complicated for intricate queries or queries involving several files.

The reason is that, no matter how complex, a query must be expressed in its entirety in *one* statement. Specifications that in a traditional language would be implemented naturally by combining some simple constructs must be expressed now by means of clauses and further SELECTs awkwardly nested within the various parts of the main SELECT. Moreover, in order to support real-world queries, some contrived features had to be added to SELECT. The features are, in reality, substitutes for ordinary programming concepts. But, while the traditional languages provide these concepts naturally, as diverse statements, in SQL they must all be crammed, artificially, into the SELECT statement. SQL, thus, while perceived as a modern, high-level database language, is in fact a primitive, ugly language.

Another way to include traditional operations in the SELECT statement was by making them look similar to the relational operations. For example, an operation that results in one value for a group of selected records (the sum of the values present in certain fields, or their average, or minimum) can be included through an option that creates a temporary file of one record where the fields contain the result; and an operation performed on a certain field in every record in the group (calculating the square root, multiplying by a constant, etc.) can be included through an option that creates a temporary file with the same number of records as the original group, but where the fields contain the new value. Many operations easily performed in traditional languages (mathematical and statistical functions, character string manipulation, date and time calculations, etc.) were artificially added to the SELECT statement in this fashion.

Clearly, if we have to develop real-world applications while being unable to create our own file scanning loops, and if SELECT is the only statement available, every operation that we will ever need must be included somehow in this one statement. Thus, the reason for the growing complexity of SELECT is the desire to keep SQL "non-procedural"; specifically, the attempt to provide programming capabilities while restricting these capabilities to a higher level of abstraction than a traditional language. This is an absurd quest, since, if we want the ability to implement any conceivable queries, the language must provide low-level file operations. (We examined in chapter 6 the fallacy of non-procedural languages; see pp. 442–443.) So, in the end, the entities and operations that became part of the SELECT statement had to be of the same level of abstraction as those used in traditional languages: fields, records, keys, comparisons, calculations, and so on.

What the relational theorists are trying to avoid at any cost, even if the cost is increased complexity, is code like that shown in figures 7-13 to 7-16 (pp. 680, 683–685); that is, traditional programming, where the file operations are managed through explicit flow-control constructs. An SQL SELECT statement

may well be a little shorter than the equivalent COBOL code, but it does not provide a higher level of abstraction.[27] What is different between SQL and COBOL – *implicit* loops and conditions as opposed to *explicit* ones – is the easy, mechanical part of programming. The difficult part – the overall logic, the file relations, the concept of nested loops and conditions, the links between database entities and the other entities in the application – is necessarily the same in both. With SQL or with COBOL, since the computer cannot know what we want, the only way to implement a given query is by specifying all the details. It is futile to seek a higher level of abstraction.

Thus, even when restricted to queries (and hence still within the relational model), we already note the need to enhance SQL in order to extend its usefulness beyond trivial requirements, as well as the effort to cover up the fact that this is achieved by introducing *programming* capabilities. A complex SQL query is in reality a little program, and what we are doing when creating a complex SELECT statement is programming. We would be better off, therefore, to implement that requirement as several simpler statements, linked through a flow-control structure that follows naturally and intuitively the query's logic, as we do in most languages. But then we could no longer delude ourselves that SQL is non-procedural, or that we are using only high-level relational operations. In the end, as in all mechanistic software delusions, not only did SQL fail to eliminate the need for programming, but in attempting to do this it made programming more difficult.

❖

When the relational systems were expanded to include updating operations, SQL was enhanced with the capability to add, modify, and delete records. The respective statements are INSERT, UPDATE, and DELETE. And these statements are very similar to SELECT, in that they create an implicit file scanning loop and include clauses for various details (record selection criteria, for instance). Updating operations, we recall, are not part of the formal relational model. Thus, regardless of how we feel about SQL as a *query* relational language, the new statements cannot be judged at all by relational principles. So the fact that they are in the same contrived style as SELECT, or the fact that INSERT also permits us to bypass the relational principles altogether and process individual records, can easily be overlooked.

Recall the traditional file operations (pp. 676–679): WRITE, REWRITE, DELETE,

[27] SQL code corresponding to the COBOL code of figure 7-13 might be: SELECT P-NUM FROM PART WHERE P-NUM>=P1 AND P-NUM<=P2 AND P-QTY>=Q1 ORDER BY P-NUM. For the operations in figures 7-14 to 7-16, however, the SQL code would be far more involved, especially if we have to access individual fields from two or three files at the same time.

READ, START, and READ NEXT. We concluded that this is the minimal practical set of file operations – the operations that are both necessary and sufficient for using indexed data files in serious applications. In conjunction with the flow-control constructs provided by the traditional languages, these operations permit us to implement any conceivable database requirement. Putting it in reverse, to permit us to implement any database requirement, a database system *must* provide these operations, or their equivalent.

After the various enhancements, SQL provided four of these operations: INSERT, UPDATE, DELETE, and SELECT correspond, respectively, to the traditional WRITE, REWRITE, DELETE, and READ. Only START and READ NEXT had no SQL counterpart. READ NEXT instructs the file system to retrieve the current record in the indexing sequence and advance the pointer to the next record. It is normally used, therefore, in a file scanning loop (and START is used once, before the loop, to indicate the first record). READ NEXT was thought to be unnecessary in SQL because the four other statements create their own, implicit file scanning loops.

So, with the traditional operations we use READ to access *individual* records, and READ NEXT to access in a loop a series of *consecutive* records; and to modify or delete records we use REWRITE or DELETE, either for individual records or in a READ NEXT loop. With the SQL statements, on the other hand, we access records *only* in a loop – the *implicit* loop generated by each one of the four statements. (Consequently, if we need to access a single record in SQL, we must specify selection criteria that will result in a trivial loop of one iteration.)

The most striking difference between SQL statements and the traditional operations, then, is the implicit file scanning loop as opposed to the loop that we create ourselves. So SQL statements are a little simpler, but to benefit from this simplicity we must give up all flexibility. When we create our own scanning loop, in a traditional language, we can include in the loop additional operations (to perform various tasks related to the file operation). In SQL, the only operations we can have in the loop are those provided by the statement itself, through its clauses. For example, in SQL we can specify with UPDATE the record selection criteria and how to modify the fields in these records. But with a traditional language, a loop based on READ NEXT and REWRITE can also include display operations, subroutine calls, and calculations involving both database fields and memory variables. Thus, when we create our own file scanning loop we can easily link the file entities to the other entities in the application. This is the seamless integration of the database and the application that we discussed earlier (see "The Lost Integration").

We saw how the relational theorists crammed into SELECT various features in an attempt to restore some of the flexibility that was lost in the implicit SQL loops. But there is a limit to the number of operations that can be specified in this fashion, and in the end they had to admit that the capability to create *explicit* file scanning loops, and to control the operations in the loop, is an indispensable database feature. So they added this feature to SQL too, by way of a new enhancement: the FETCH statement.

FETCH is the true counterpart of the traditional READ NEXT: it lets us create explicit loops, and retrieve one record at a time, just as we do in a traditional language. (There is no equivalent of the traditional START: in SQL we always start from the beginning of the file, and the system will deliver only those records that passed the selection criteria previously specified with a SELECT.) FETCH, of course, is not independent. To use it we also need the capability to create explicit loops, and this capability was added to SQL by means of further enhancements: actual loop-control constructs, and a way to perform SQL statements from within a traditional language. (We will examine these enhancements in a moment.)

The mechanism through which we read one record at a time in a loop is known in SQL as *cursor*, and is identical to the mechanism known as *pointer* in the traditional file operations (see pp. 677, 678). The cursor is the indicator that keeps track of records in the current indexing sequence: each time we perform a FETCH, the system retrieves the record identified by the cursor and advances the cursor to the next record – just as it does in the case of the traditional READ NEXT. And if we do this at the beginning of each iteration, all the operations in the loop will be able to access the fields in that record. Thus, in SQL too we can now include in a file scanning loop any operations we want, and thereby link the database fields to other types of entities (memory variables, display fields, etc.). Also, when used in conjunction with FETCH, UPDATE and DELETE can now modify or delete individual records in a scanning loop – just as REWRITE and DELETE can in conjunction with READ NEXT in a traditional loop.

❖

To appreciate the importance of the cursor, we must recall the original relational principles. For, without the means to create explicit file scanning loops, SQL would have been almost useless. The relational model specifically restricts us to high-level operations: all we can do is extract and combine logical portions of tables. The permitted operations are PROJECTION, SELECTION, UNION, JOIN, and the like (see p. 702). The original SQL SELECT statement, with its implicit file scanning loop, follows this principle: we specify the operations

through the various clauses of a SELECT, and combine them by nesting SELECTs within one another. At every step we manipulate only tables – tables that contain, usually, just some of the records and fields of an actual data file.

So the original SELECT statement (plus INSERT, UPDATE, and DELETE if we allow updating operations) is *all we need* in order to implement the relational model in SQL. This is true because in high-level queries, as the model was originally intended, we only need the means to specify which fields to list, and such details as their order and format. But if we want to employ the model for *any* conceivable query, in *any* application, we need the means to perform additional operations with these fields, not just list them. Also, we need the means to use the fields together with other data types – display and data entry fields, and memory variables. The theorists hoped at first to satisfy these two demands by adding more and more options to the SELECT statement; that is, by inventing a high-level feature for every conceivable situation. This is an absurd idea, however, and they realized in the end that the only practical solution is to permit *low-level* operations.

Thus, only trivial requirements can be implemented if restricted to the implicit file scanning loops of SELECT. It was by adding to SQL the concept of a cursor, and the means to create *explicit* file scanning loops, that we gained the two critical qualities: the capability to perform additional operations in the loop, and the capability to link low-level database entities (individual fields and records) to other entities in the application.

With the concept of a cursor, then, all the capabilities of the traditional file operations were finally available in relational systems. But this was accomplished by abolishing the relational principles: the way we use data files in SQL is now practically identical to the way we use them in a traditional language.

As SQL was used for more and more demanding tasks, it had to be enhanced with the kind of features found in general-purpose programming languages. And software vendors increasingly used *these* features – which have nothing to do with the relational principles, or with database operations – as a way to promote their database systems and attract buyers. For example, some vendors enhanced their version of SQL with the means to create conditional, iterative, and other flow-control constructs (officially abandoning, therefore, the idea of a non-procedural language). And, in addition to those functions similar to the traditional operations and subroutine libraries, already mentioned, countless expedients were provided to assist programmers in developing applications: functions for creating reports, for data entry and display, for system management, and so forth.

What is the point of these enhancements? We already had these features, in a hundred languages. The relational promise had been mathematical logic and a higher level of abstraction, not a new programming language. And if this idea turned out to be impractical, it should have been abandoned. Instead, like all pseudoscientists, the relational experts rescued their theory by reinstating precisely those concepts that the theory had attempted to replace. As a result, software vendors are competing today, not by stressing the *relational* capabilities of their systems, but by adding more and more low-level, programming features; that is, features meant to help us bypass the rigours imposed by the relational model. In other words, the value of a relational system is measured today by how good it is at overriding the relational principles.

But despite the enhancements, SQL remained inferior to the traditional languages. It was still too awkward and too inefficient for serious applications, so one more feature had to be invented: the capability to use SQL from within a traditional language. This feature, called *embedded SQL*, is the ultimate relational degradation: the most effective way for a programmer to enjoy the benefits of the traditional database concepts while pretending to use the relational model.

With embedded SQL, we implement in a traditional language the entire application, including all database requirements; then, we invoke isolated SQL statements here and there in the form of subroutines. The relational system, thus, is relegated to the role of subroutine library, and works similarly to a traditional file system. A typical use of this concept is with the FETCH statement, as explained earlier: we create a file scanning loop in COBOL or any other language, and use FETCH within the loop to read one record at a time – exactly the way we use the traditional READ NEXT. Every other operation in the loop is implemented in the traditional language. The resulting code is identical, for all practical purposes, although we employ in one case a relational system and in the other a traditional file system. We have come a long way from the idea of "tables and nothing but tables," accessed through high-level operations.

An important promise of the relational model had been that the result of a query is mathematically guaranteed to be correct: if we restrict ourselves to the relational operations – to extracting and combining portions of tables – the data in the final table will always reflect accurately the data in the tables we started with. So, if we bypass the restriction to relational operations, this promise no longer holds. Whether the new operations are added in the form of SELECT options or in the form of explicit file scanning loops, the benefits promised by the relational model are now lost. Without the restriction to

relational operations, what we have is no longer a relational model, so the resulting tables may or may not reflect accurately the starting ones.

The SQL fallacy, thus, is the belief that the relational model can be enhanced with features that contradict its most fundamental principles, and still retain its original qualities. The mathematical benefits were shown to emerge only if we restrict ourselves to the relational operations. The theorists keep adding features designed specifically to bypass this restriction, but they continue to promote the model with the original claims.

We already know that the *updating* operations lie outside the scope of the formal model, so the model's mathematical grounding is irrelevant when a relational system is used for general database work. And now we see that the model's mathematical grounding has become irrelevant even for queries. As was the case with the other modifications, the SQL features do not *enhance* the relational model but *annul* it.

## The Verdict

In the end, what has the relational model accomplished? After thirty years of "enhancements," relational database programming is more or less the same as *traditional* database programming: we manipulate fields, keys, records, and files in order to create database structures. The only real difference is that the database operations have been separated from the rest of the application, and are now possible only through complicated, inefficient, and expensive database environments.

If we disregard that extreme degradation, embedded SQL, applications are now divided into two parts: the part written in an ordinary language, where the application's main logic resides, and the part written in SQL (in the form of integrity functions, stored procedures, and the like), where the database-related operations reside. More and more pieces of the application have been moved into the SQL part; this is not because they are easier to implement in SQL, though, but because they are closely related to the database operations, and keeping them together is the only practical way to link them. And this artificial separation obscures the fact that the part dealing with the database-related operations is now very similar to what it was when integrated with the application's main logic.

The relational charlatans, thus, claimed at first that we must separate the database operations from the application, and restrict the links between the two to a high level of abstraction, because this is the only way to benefit from the relational model. Then, when the separation proved to be impractical, they

restored the low-level links. They did it, though, not by moving the database operations back into the application, but by bringing into the SQL procedures further pieces of the application. They restored the links, thus, by reinstating in the SQL procedures the same low-level programming concepts that we had used in the application before the separation. So the benefits believed to emerge from the relational model are now lost even if we forget that they had already been lost, in the other annulments of the relational principles. The separation of the application into two parts is absurd because what we are doing in the SQL procedures is about the same as what we were doing before, in a much simpler way, in the application.

So, after all the "enhancements," there remains very little that is relational in the relational database systems. Programmers use SQL in about the same way that the traditional file operations are used. Only now and then, when not too inconvenient or too inefficient, do they employ the relational operations as they were defined in the original theory. But by calling files "tables," records "rows," and fields "columns," they can delude themselves that they are programming under the relational model.

It must be noted that some features are indeed found only in relational systems. But these features could easily be added also to the traditional file systems, simply because they have nothing to do with the relational model. These are the kind of features made possible by hardware and software advances – larger files, new types of fields, enhanced caching and buffering, better security or backup facilities, and the like. So, if these features are missing in a file system, it is deliberate: in their effort to make everyone dependent on complicated and expensive development environments, the software elites are doing everything in their power to discredit the straightforward, traditional languages and file systems; and refusing to keep them up to date is part of this manipulation.

It is all the more remarkable, thus, that the traditional languages and file systems, while remaining practically unchanged for the last thirty years, have been the chief source of inspiration for the features added to the relational systems. This shows, again, just how little the relational model itself had to offer.

The relational model is still described as an application of mathematical logic. And those monstrous database systems are promoted with the claim that the relational model is the only way to have rigorous databases, even as everyone can see that these systems have little to do with the relational model, and that their only practical features are those taken from the traditional languages and file operations. So, like the theories of structured programming and object-oriented programming, and like all other pseudosciences, the relational theory continues to be promoted on the basis of its original

principles even after these principles were abandoned, and hence their benefits were lost.

If we have to bypass the relational restrictions and revert to operations that are practically identical to the traditional ones, in what sense is the relational model beneficial? The theorists are committing a fraud when promoting the relational systems if, at the same time, they enhance these systems with means to override the relational principles.

The multibillion-dollar relational database industry thrives on the incompetence of the software practitioners, whose skills are limited to knowing how to use programming aids and substitutes. To repeat, the six basic file operations and an ordinary language are all we need in order to implement database requirements. Thus, only a programmer incapable of designing some simple loops and conditions can be impressed by the relational features. Every one of these features has been available – in a much simpler form, through file management systems and languages like COBOL – since about 1970.