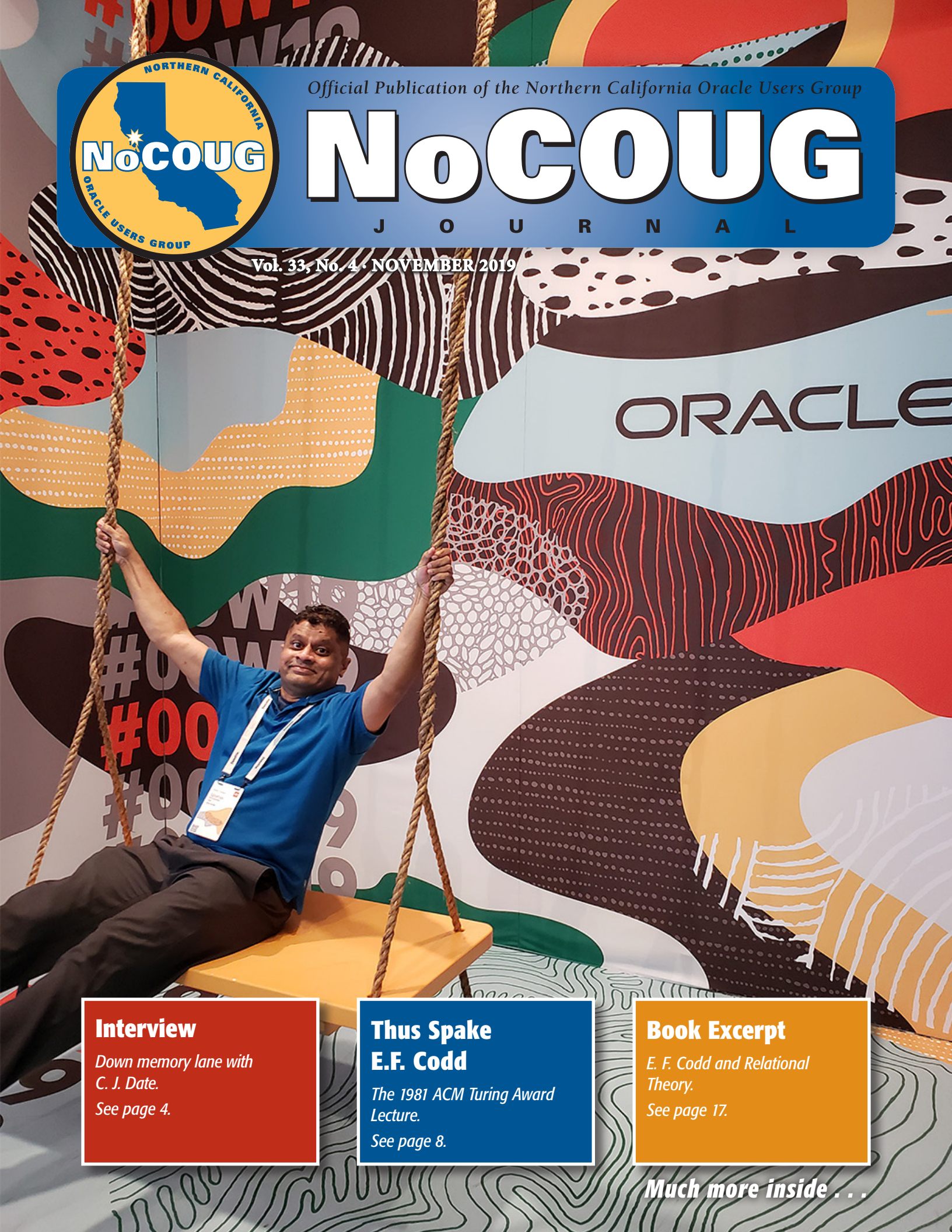
Official Publication of the Northern California Oracle Users Group

NoCOUG JOURNAL

Vol. 33, No. 4 · NOVEMBER 2019



Interview

Down memory lane with C. J. Date.

See page 4.

Thus Spake E.F. Codd

The 1981 ACM Turing Award Lecture.

See page 8.

Book Excerpt

E. F. Codd and Relational Theory.

See page 17.

Much more inside . . .

Amazon Aurora

Performance and availability of commercial-grade databases at 1/10th the cost.

Learn more: aws.amazon.com/aurora

aws

Professionals at Work

First there are the IT professionals who write for the *Journal*. A very special mention goes to Brian Hitchcock, who has written dozens of book reviews over a 12-year period.

Next, the *Journal* is professionally copyedited and proofread by veteran copy-editor Karen Mead of Creative Solutions. Karen polishes phrasing and calls out misused words (such as "reminiscences" instead of "reminisces"). She dots every i, crosses every t, checks every quote, and verifies every URL.

Then, the *Journal* is expertly designed by graphics duo Kenneth Lockerbie and Richard Repas of San Francisco-based Giraffex.

And, finally, the *Journal* is printed and shipped to us. This is the 132nd issue of the *NoCOUG Journal*. Enjoy! ▲

Table of Contents

ADVERTISERS

Publication Notices and Submission Format

The *NoCOUG Journal* is published four times a year by the Northern California Oracle Users Group (NoCOUG) approximately two weeks prior to the quarterly educational conferences.

Please send your questions, feedback, and submissions to the *NoCOUG Journal* editor at **journal@nocoug.org**.

The submission deadline for each issue is eight weeks prior to the quarterly conference. Article submissions should be made in Microsoft Word format via email.

Down Memory Lane

with C.J. Date

"Precious memories, how they linger
How they ever flood my soul"

C.J. Date

Your new book, *E. F. Codd and Relational Theory*, has just been published. Tell us about it: What does it cover? Why did you write it, and what exactly are you trying to achieve with it? Who's your target audience?

Thanks for giving me the chance to answer these questions! What does the book cover? Well, first, notice my title. E. F. ("Ted") Codd was the original inventor of the relational model, of course, and it was that invention that turned databases into an academically respectable field and at the same time made them a truly practical and commercially viable proposition. All of us who work on databases today owe our professional existence, and indeed our very livelihoods, to the work that Ted did in the late 60s and early 70s. Yet in my experience very few database professionals nowadays are aware of what he actually did—in fact, I've met many who've never even heard of Ted Codd! So one thing I wanted to do with the book was give readers some sense of the historical foundations of their field. To do this, in the book I examine in considerable detail what seem to me (for one reason or another) to be the most significant of Ted's writings, explaining in each case what the contribution was, and in particular highlighting not only the many things he got right but also some of the things he got wrong. (No one is perfect, and I have to say Ted did make a few mistakes! But we can forgive him those mistakes in view of the one thing he did get superlatively right—his gift to the world—namely, his wonderful relational model.)

By the way, I hope you don't mind me constantly referring to him as "Ted." Of course, I don't mean to be either presumptuous or disrespectful here, but the fact is that he and I were friends and professional colleagues for many years, and "Ted" is how I always think of him.

Why did I write the book? Well, I've effectively answered this already, in part. Generally speaking, I've been troubled for years by the widespread lack of understanding of relational matters on the part of people who really ought—and need!—to know better. (As an aside, I note that the book includes a short list of relational questions that I used to ask attendees at the beginning of my live seminars. What I found was that most of those attendees—all of whom were database professionals, by the way, typically DBAs or database application developers—were unable to answer *any* of those questions completely correctly.)

More specifically, I've been troubled by how few people seem to have actually read Ted's writings or have any kind of proper understanding or appreciation of his work. So I wanted to produce something that people could read and refer to as a useful summary and analysis of those writings and that work. At the same time I wanted to set the historical record straight here and there … I mean, I have a huge admiration for what Ted did—as we all should have!—but it's important not to be blinded by such feelings into uncritical acceptance of everything he said or wrote. Nor do I feel it appropriate to accept him as the sole authority on relational matters. The fact is, he did get some things wrong, and I feel those things need to be documented too, as well as all the things he got right.

As for my target audience: Of course I'd like *everyone* to read it! More seriously, I believe the material the book covers is something that all database professionals should have tucked away in their brain, as it were, as part of their general background knowledge. So my audience is anyone with a professional interest in relational databases—including, of course, anyone with a professional interest in SQL.

There are a few more things I'd like to say in connection with your original question here. In fact I'd like to quote some text from the book itself. (Very conceited to quote from oneself, I realize!—please forgive me.)

> [Ted's] papers were staggering in their originality. Among other things, they changed, and changed permanently, the way database management was perceived in the IT world; more specifically, they transformed what had previously been nothing but a ragbag of tricks and ad hoc techniques into a solid scientific endeavor. They also, not incidentally, laid the foundation for an entire multibillion dollar industry. Together, they provided the basis for a technology that has had, and continues to have, a major impact on the very fabric of our society. Thus, it's no exaggeration to say that Codd is the intellectual father of the modern database field.
>
> To the foregoing, I'd like to add this: Codd's relational model has been with us now for over 50 years. And I for one think it very telling that, in all that time, no one has managed to invent any kind of new theory, one that might reasonably supplant or seriously be considered superior to the relational model in any way. In my opinion, in fact, no one has even come close to inventing such a theory (though there have been many attempts, as I'm sure you're aware, but attempts that in my opinion have universally failed).

And yet . . . And yet, here we are—as I've said, over 50 years later—and what do we find? Well:

- First, the teaching of relational theory, in universities in particular, seems everywhere to be in decline. What's more, what teaching there is seems not to have caught up—at least, not fully, and certainly not properly—with the numerous developments in relational theory that have occurred since publication of those early papers of Codd's.
- Second, no truly relational DBMS has ever been widely available in the commercial marketplace.

So here are a couple more reasons why I wrote the book … First, I wanted to provide something that might help spur educators, in universities and elsewhere, to get back to teaching relational theory properly (paying attention in particular to some of the work done by others since Ted's early papers). Second, I wanted to suggest some answers to the question implicit in the second of the foregoing bullet points. But I see you're going to ask essentially that same question explicitly later, so let me defer further discussion to my response to that later question.

Can you say something about the book's structure?

The book's subtitle is "A Detailed Review and Analysis of Codd's Major Database Writings." Obviously, the question arises: Which of Ted's writings qualify as major? In my opinion, the following ones do:

- His 1970 paper "A Relational Model of Data for Large Shared Data Banks" and its 1969 predecessor "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks"
- His 1971 papers "A Data Base Sublanguage Founded on the Relational Calculus" and "Further Normalization of the Data Base Relational Model"
- His papers "Relational Completeness of Data Base Sublanguages" (1972) and "Interactive Support for Nonprogrammers: The Relational and Network Approaches" (1974)
- His 1985 two-part Computerworld article, "Is Your DBMS Really Relational?" and "Does Your DBMS Run by the Rules?"
- His 1990 book *The Relational Model for Database Management Version 2*

My book takes seven long chapters to consider the foregoing publications in detail. It also has a "Setting the Scene" chapter that explains exactly what a relational DBMS is (or ought to be, at any rate). Incidentally, that chapter includes a detailed list of SQL departures from the relational model. And the book also has three appendixes: one containing the text of the piece I wrote in connection with Codd's 1981 Turing Award for the pertinent section of the ACM website (**amturing.acm.org**); one containing a set of formal definitions; and the last consisting of a consolidated list of references for the entire book.

What's the competition for your book?

I don't mean to sound arrogant, but I don't think there are any books out there that cover exactly the same material in exactly the same way. In fact I don't really think there could be, given that Ted and I were friends and colleagues for so many years, first in IBM and then in a variety of companies with names like Codd & Date International, Codd & Date Limited, and so on; I mean, I believe I can claim certain unique insights in this area that other writers simply can't.

That said, the closest competitor is probably a book of my own anyway: viz., *The Database Relational Model: A Retrospective Review and Analysis* (Addison-Wesley, 2001). Let me explain. Some 20 years ago or so I wrote a series of twelve articles for the

"I don't really blame practitioners (as opposed to researchers) if they haven't read Ted's writings. Ted was a mathematician originally, and his first few papers tended to reflect that fact—the writing was terse and a little dry, the style theoretical and academic, the notation and examples mostly rather mathematical in tone, and it wasn't easy to relate what they had to say to practical day-to-day concerns. In a word, they weren't easy to read."

Miller Freeman monthly magazine *Intelligent Enterprise*, beginning with the October 1998 issue. The overall title for the series was "30 Years of Relational," because they were written in part to celebrate the relational model's forthcoming 30th birthday (August 19th, 1999). As I wrote at the time:

> [These articles are intended] to serve as a historical account and . . . analysis of E. F. Codd's (huge!) contribution to the field of database technology. Codd's relational model, [described in] a startlingly novel series of research papers, was a revolution at the time, albeit one that was desperately needed. Now, however, it seems that—despite the fact that the entire multibillion dollar database industry is founded on Codd's original ideas—those ideas are in danger of being ignored or forgotten (or, at best, being paid mere lip service to). Certainly we can observe many examples today of those ideas being flouted in (among other things) database products, database designs, and database applications. It thus seems appropriate to take another look at Codd's original papers, with a view to assessing their true significance and restating, and reinforcing, their message for a new generation of database professionals . . . So I thought it would be interesting, and (I also thought) useful, to devote a short series of articles to a careful, unbiased, retrospective review and assessment of [those] papers.

And I subsequently collected those articles into the "Retrospective" book mentioned above. However, while I still believe that book provides a useful overview of Ted's achievement overall (and the new book certainly isn't meant to replace it or displace it), I must make it clear that the new book goes into a much

greater level of detail than the earlier book did, and it's really aimed at a different—perhaps a slightly more sophisticated—audience.

I'm intrigued by your reference above to a list of questions that you said attendees on your seminars couldn't answer. Can you tell us more about them?

I'd be happy to—but I must stress that the book isn't aimed primarily at answering them, though in fact it does answer most of them in passing. Let me say first that the seminar on which I asked these questions was called "SQL and Relational Theory," and attendees were certainly supposed to be pretty familiar with SQL ahead of time. Anyway, here are the questions:

1. What exactly is first normal form?
2. What's the connection between relations and predicates?
3. What's semantic optimization?
4. What's an image relation?
5. Why is semidifference important?
6. Why doesn't deferred integrity checking make sense?
7. What's a relation variable?
8. What's prenex normal form?
9. Can a relation have an attribute whose values are relations?
10. Is SQL relationally complete?
11. Why is *The Information Principle* important?
12. How does XML fit with the relational model?

Twelve questions, of course (as you know, everything in the relational world comes in twelves).

By the way, I complained earlier that university education these days doesn't seem to be all that it might be as far as relational matters are concerned. In particular, I frankly doubt whether the average university class deals with the foregoing issues properly (perhaps not at all, in some cases). What's more, much the same can be said of the majority of the numerous database textbooks currently available—at least the ones I'm aware of.

Personally I found portions of your book simply delightful. But I have to ask: Why bother analyzing papers and publications that today's practitioners and even today's researchers have not read and probably never will read?

Thank you for your kind words . . . Over the years I've had many adjectives used to describe my various writings as you can surely imagine, but I don't think "delightful" was ever one of them before. Anyway, you ask (paraphrasing): Why analyze writings that today's practitioners and researchers have never read and probably never will read? Well, I've tried to answer this question in part already, but let me say a bit more about it.

First, researchers. Personally, I'd be ashamed to call myself a researcher of any kind if I wasn't thoroughly familiar with the foundational writings in my field. For example, can you imagine someone calling himself or herself a cosmologist and not being familiar with Einstein's original papers? In other words, database researchers simply owe it to themselves to read Ted's original writings. They have no excuse not to! And then I'd like to think that my book could serve those researchers as a useful explanation of, and commentary on, and even a guide to, those writings—rather like the numerous books extant today that serve as an explanation of and commentary on and guide to Einstein's original writings.

Turning to practitioners: Actually, I don't really blame practitioners (as opposed to researchers) if they haven't read Ted's writings. Ted was a mathematician originally, and his first few papers tended to reflect that fact—the writing was terse and a little dry, the style theoretical and academic, the notation and examples mostly rather mathematical in tone, and it wasn't easy to relate what they had to say to practical day-to-day concerns. In a word, they weren't easy to read. So here I see my book as serving another, slightly different purpose: I see it as explaining what those papers of Ted's were all about, but in a way that's less formal than the originals wherever possible, with simple and practical examples. But I'd like to stress that the book isn't meant to be a substitute for Ted's originals. Rather, I'd like readers to read Ted's originals alongside my chapters, or after them, if they can. (Most of those writings of Ted's can now be found online, by the way.)

Put another way, Codd's vision was never fully implemented, correct? Why not?

I think this is a sad story. It's true that no fully relational commercial DBMSs exist today, so far as I'm aware. (Of course, people can and will argue about what "fully relational" means, but I don't want to get into that debate here.) Sadly, it seems to me that a small part of the blame for this depressing state of affairs has to be laid at Ted's own door. The fact is, those early writings of his, brilliantly innovative though they undoubtedly were, did suffer from a number of defects, as I've already indicated. And it's at least plausible to suggest that some of the sins to be observed in the present database landscape—sins of both omission and commission—can be regarded, with hindsight (always perfect, of course), as deriving from the defects in question.

A much bigger part of the blame, though, has to be laid at SQL's door. SQL was, of course, designed in IBM Research, and it's quite clear that the people responsible for its design never understood the relational model as well as they should have done. (I have written evidence of this claim which this margin is unfortunately too small to contain.) As a consequence, SQL was a long, long way from being what it ought to have been, viz., a true concrete realization of the abstract ideas of the relational model. Nevertheless, when IBM management finally decided (after delaying for much too long!) that IBM should build a relational product, they grabbed on to SQL as the user language for that product, because SQL did at least already exist, albeit only in prototype form. Never mind that there were people in IBM at the time pointing out all the problems with SQL, and pointing out too that it would be better and easier, and probably even cheaper, to use a truly relational language! All that those IBM managers could see was that SQL was light years better than DL/I, the language used with its hierarchical product IMS; thus, arguments that there could be something light years better than SQL simply cut no ice.

So IBM ran with SQL; and given IBM's prominence in the computing world in those days, everyone else decided they needed to have SQL too in order to compete. Also a standard was developed that (at least in its initial manifestations) was essen-

tially IBM SQL, warts and all. And so here we are now, stuck with it. As I say, I find this a very sad story; we had a once-in-a-lifetime opportunity to do it right, and we blew it.

You're quite dismissive of SQL too. But you claim that SQL can be used safely, is that correct?

I'm not "dismissive" of SQL. It would be absurd to dismiss something so widespread and so (within its own lights) successful! Rather, I regard it as a clear and present evil, something that exists and so has to be dealt with appropriately. And yes, I do believe it can be used more or less "safely," as you put it. In a nutshell, I believe it's possible to limit your use—more or less—to what might be called "the relational kernel" of SQL. If you do that, you can behave as if SQL truly were relational—more or less—and you can enjoy the benefits of working with what is in effect a truly relational system.

Of course, if you're going to follow such a discipline, you need to know what that relational kernel is, which means in effect that (a) you need to know the relational model and (b) you need to know exactly how every aspect of that model materializes in SQL. In fact I wrote an entire book based on these ideas—*SQL and Relational Theory: How to Write Accurate SQL Code* (3rd edition, O'Reilly, 2015). But now I'm beginning to stray somewhat from the book I'm supposed to be talking about . . . Let me just say in conclusion that the new book will certainly give you a good idea of what the relational model is all about, but it doesn't have much to say about SQL as such. After all, Ted Codd never had very much to do with SQL as such—though (like me) he did subsequently became very critical of it.

Thank you for giving us so much of your time. One final question. This issue of the* NoCOUG Journal *includes a reprint of the ACM Turing Award Lecture delivered by Codd at ACM '81. Do you have any memories or stories about that lecture to share with us?

As a matter of fact I do. As I mentioned earlier, Ted did indeed receive the 1981 ACM Turing Award for his invention of the relational model, and nobody was more pleased about that than me. (Well, perhaps Ted was.) Actually I was the one who nominated him for that award; and there's a story there . . . I'm sure you understand that putting a proper Turing Award nomination together is a lot of work. You need to provide detailed statements of who the nominee is, and why you think he or she deserves the award; what the person has achieved (i.e., what the contribution is); why the work is significant and novel, and in fact outstanding; what its relationship is to previous work in the field; and what the implications of the work are for computing as such, and for industry, and possibly even for society at large. You also need to provide a list of publications and citations, and letters from recognized authorities in the field supporting the nominee's candidacy, and a whole host of other things besides. And ideally you need to put all this material together without giving the nominee any hint of what's going on (for fear of raising false hopes, perhaps).

Well, I wanted Ted to get the award. I certainly felt he deserved it, and I was prepared to do the work of putting the nomination package together, and so I went to see my IBM manager Jim to get permission to spend maybe a week of my time doing so. (Ted and I were both still working in IBM in those days.) Jim was in charge of the DB2 Technical Planning Department—think about that as you hear what happened next!—and my conversation with him went like this.

> **Me:** "Jim, I'd like to take some of my time to work on nominating Ted Codd for the Turing Award. I think it'll take about a week of my time, though it'll probably continue at noise level over the next few weeks as well."
>
> **Jim:** "What's the Turing Award?"
>
> **Me *(slightly taken aback)*:** "*Jim!* It's the top award in computing! It's not quite the Nobel prize, but it's kind of like a mini Nobel prize. It's awarded every year by the ACM."
>
> **Jim:** "What's the ACM?"
>
> **Me (*even more taken aback, jaw dropping*):** "?!?!?"

At least he didn't ask who Ted Codd was.

Well, anyway, I did do the nomination, and of course Ted did get the award, and that was great. *But* . . . Ted was certainly a genius, but he was also a great procrastinator. The conference was getting nearer and nearer, and Ted was going to have to give his Turing lecture and write a paper to go with it, and repeated efforts by myself (to some extent) and Sharon Weinberg (to a much greater extent) to get him to prepare something seemed to run up against a brick wall every time. Ted didn't even have a theme in mind! Eventually Sharon suggested the theme of productivity, and Ted agreed to it. But he still seemed not to understand that the presentation was needed *right now* (the paper could wait a while, of course). In fact, I'm pretty sure I'm right in saying that most of the slides Ted used in his presentation were actually prepared by Sharon, and prepared quite literally just the night before.

I was there, of course—in fact I was the one who introduced Ted, and it was very tempting to tell the audience the two stories above, but I very nobly didn't.

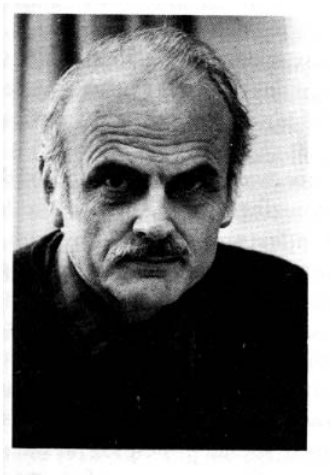
One last point: You'll notice that I don't include Ted's Turing Award paper in my list of what I called his major writings. That's because that paper wasn't truly theoretical or innovative like the ones my book deals with—it was more a kind of summary of where relational systems had come from and where they were going, with the emphasis on why they could indeed serve as the advertised "practical foundation for productivity." All of which was perfectly right and proper, of course, given the context. In particular, I was very glad to see Ted attack a certain rather common but head-in-the-sand belief—one that we both ran up against repeatedly in the early days—namely, the one summed up in the ignorant phrase "If it's theoretical, it can't be practical!"

At the same time I do have to say there are aspects of the paper that I don't agree with. To get into details of such aspects would probably need another interview, though, or even another book, so I think I'd better stop while I'm ahead. ▲

C.J. Date has a stature that is unique in the database industry. He is best known for his textbook An Introduction to Database Systems (Addison-Wesley). He enjoys a reputation that is second to none for his ability to explain complex technical issues in a clear and understandable fashion.

The 1981 ACM Turing Award Lecture

Delivered at ACM '81, Los Angeles, California, November 9, 1981



The 1981 ACM Turing Award was presented to Edgar F. Codd, an IBM Fellow of the San Jose Research Laboratory, by President Peter Denning on November 9, 1981 at the ACM Annual Conference in Los Angeles, California. It is the Association's foremost award for technical contributions to the computing community.

Codd was selected by the ACM General Technical Achievement Award Committee for his "fundamental and continuing contributions to the theory and practice of database management systems." The originator of the relational model for databases, Codd has made further important contributions in the development of relational algebra, relational calculus, and normalization of relations.

Edgar F. Codd joined IBM in 1949 to prepare programs for the Selective Sequence Electronic Calculator. Since then, his work in computing has encompassed logical design of computers (IBM 701 and Stretch), managing a computer center in Canada, heading the development of one of the first operating systems with a general multiprogramming capability, contributing to the logic of self-reproducing automata, developing high level techniques for software specification, creating and extending the relational approach to database management, and developing an English analyzing and synthesizing subsystem for casual users of relational databases. He is also the author of *Cellular Automata*, an early volume in the ACM Monograph Series.

Codd received his B.A. and M.A. in Mathematics from Oxford University in England, and his M.Sc. and Ph.D. in Computer and Communication Sciences from the University of Michigan. He is a Member of the National Academy of Engineering (USA) and a Fellow of the British Computer Society.

The ACM Turing Award is presented each year in commemoration of A. M. Turing, the English mathematician who made major contributions to the computing sciences.

Relational Database: A Practical Foundation for Productivity

E. F. Codd
IBM San Jose Research Laboratory

It is well known that the growth in demands from end users for new applications is outstripping the capability of data processing departments to implement the corresponding application programs. There are two complementary approaches to attacking this problem (and both approaches are needed): one is to put end users into direct touch with the information stored in computers; the other is to increase the productivity of data processing professionals in the development of application programs. It is less well known that a single technology, relational database management, provides a practical foundation for both approaches. It is explained why this is so.

While developing this productivity theme, it is noted that the time has come to draw a very sharp line between relational and non-relational database systems, so that the label "relational" will not be used in misleading ways. The key to drawing this line is something called a "relational processing capability."

CR Categories and Subject Descriptors: H.2.0 [**Database Management**]: General; H.2.1 [**Database Management**]: Logical Design–*data models*; H.2.4 [**Database Management**]: Systems

General Terms: Human Factors, Languages

Additional Key Words and Phrases: database, relational database, relational model, data structure, data manipulation, data integrity, productivity

Author's Present Address: E. F. Codd, IBM Research Laboratory, 5600 Cottle Road, San Jose, CA 95193.

Communications of the ACM
February 1982 Volume 25 Number 2

1. Introduction

It is generally admitted that there is a productivity crisis in the development of "running code" for commercial and industrial applications. The growth in end user demands for new applications is outstripping the capability of data processing departments to implement the corresponding application programs. In the late sixties and early seventies many people in the computing field hoped that the introduction of database management systems (commonly abbreviated DBMS) would markedly increase the productivity of application programmers by removing many of their problems in handling input and output files. DBMS (along with data dictionaries) appear to have been highly successful as instruments of data control, and they did remove many of the file handling details from the concern of application programmers. Why then have they failed as productivity boosters?

There are three principal reasons:

(1) These systems burdened application programmers with numerous concepts that were irrelevant to their data retrieval and manipulation tasks, forcing them to think and code at a needlessly low level of structural detail (the "owner–member set" of CODASYL DBTG is an outstanding example[1]);

(2) No commands were provided for processing multiple records at a time—in other words, DBMS did not support *set processing* and, as a result, programmers were forced to think and code in terms of iterative loops that were often unnecessary (here we use the word "set" in its traditional mathematical sense, not the linked structure sense of CODASYL DBTG);

(3) The needs of end users for direct interaction with databases, particularly interaction of an unanticipated nature, were inadequately recognized—a query capability was assumed to be something one could add on to a DBMS at some later time.

Looking back at the database management systems of the late sixties, we may readily observe that there was no sharp distinction between the programmer's (logical) view of the data and the (physical) representation of data in storage. Even though what was called the logical level usually provided protection from placement expressed in terms of storage addresses and byte offsets, many storage-oriented concepts were an integral part of this level. The adverse impact on development productivity of requiring programmers to navigate along access paths to reach the target data (in some cases having to deal directly with the layout of data in storage and in others having to follow pointer chains) was enormous. In addition, it was not possible to make slight changes in the layout in storage without simultaneously having to revise all programs that relied on the previous structure. The introduction of an index might have a similar effect. As a result, far too much manpower was being invested in continual (and avoidable) maintenance of application programs.

Another consequence was that installation of these systems was often agonizingly slow, due to the large amount of time spent in learning about the systems and in planning the organization of the data at both logical and physical levels, prior to database activation. The aim of this preplanning was to "get it right once and for all" so as to avoid the need for subsequent changes in the data description that, in turn, would force coding changes in application programs. Such an objective was, of course, a mirage, even if sound principles for database design had been known at the time (and, of course, they were not).

To show how relational database management systems avoid the three pitfalls cited above, we shall first review the motivation of the relational model and discuss some of its features. We shall then classify systems that are based upon that model. As we proceed, we shall stress application programmer productivity, even though the benefits for end users are just as great, because much has already been said and demonstrated regarding the value of relational database to end users (see [23] and the papers cited therein).

2. Motivation

The most important motivation for the research work that resulted in the relational model was the objective of providing a sharp and clear boundary between the logical and physical aspects of database management (including database design, data retrieval, and data manipulation). We call this the *data independence objective.*

A second objective was to make the model structurally simple, so that all kinds of users and programmers could have a common understanding of the data, and could therefore communicate with one another about the database. We call this the *communicability objective.*

A third objective was to introduce high level language concepts (but not specific syntax) to enable users to express operations upon large chunks of information at a time. This entailed providing a foundation for set-oriented processing (i.e., the ability to express in a single statement the processing of multiple sets of records at a time). We call this the *set-processing objective.*

There were other objectives, such as providing a sound theoretical foundation for database organization and management, but these objectives are less relevant to our present productivity theme.

[1] The crux of the problem with the the CODASYL DBTG owner–member set is that it combines into one construct three orthogonal concepts: one-to-many relationship, existence dependency, and a user-visible linked structure to be traversed by application programs. It is the last of these three concepts that places a heavy and unnecessary navigation burden on application programmers. It also presents an insurmountable obstacle for end users.

Communications of the ACM
February 1982 Volume 25 Number 2

3. The Relational Model

To satisfy these three objectives, it was necessary to discard all those data structuring concepts (e.g., repeating groups, linked structures) that were not familiar to end users and to take a fresh look at the addressing of data.

Positional concepts have always played a significant role in computer addressing, beginning with plugboard addressing, then absolute numeric addressing, relative numeric addressing, and symbolic addressing with arithmetic properties (e.g., the symbolic address $A + 3$ in assembler language; the address $X(I + 1, J - 2)$ of an element in a Fortran, Algol, or PL/I array named X). In the relational model we replace positional addressing by totally associative addressing. Every datum in a relational database can be uniquely addressed by means of the relation name, primary key value, and attribute name. Associative addressing of this form enables users (yes, and even programmers also!) to leave it to the system to (1) determine the details of placement of a new piece of information that is being inserted into a database and (2) select appropriate access paths when retrieving data.

All information in a relational database is represented by values in tables (even table names appear as character strings in at least one table). Addressing data by value, rather than by position, boosts the productivity of programmers as well as end users (positions of items in sequences are usually subject to change and are not easy for a person to keep track of, especially if the sequences contain many items). Moreover, the fact that programmers and end users all address data in the same way goes a long way to meeting the communicability objective.

The n-ary relation was chosen as the single aggregate structure for the relational model, because with appropriate operators and an appropriate conceptual representation (the table) it satisfies all three of the cited objectives. Note that an n-ary relation is a mathematical set, in which the ordering of rows is immaterial.

Sometimes the following questions arise: Why call it the relational model? Why not call it the tabular model? There are two reasons: (1) At the time the relational model was introduced, many people in data processing felt that a relation (or relationship) among two or more objects must be represented by a linked data structure (so the name was selected to counter this misconception); (2) Tables are at a lower level of abstraction than relations, since they give the impression that positional (array-type) addressing is applicable (which is not true of n-ary relations), and they fail to show that the information content of a table is independent of row order. Nevertheless, even with these minor flaws, tables are the most important conceptual representation of relations, because they are universally understood.

Incidentally, if a data model is to be considered as a serious alternative for the relational model, it too should have a clearly defined conceptual representation for database instances. Such a representation facilitates thinking about the effects of whatever operations are under consideration. It is a requirement for programmer and end-user productivity. Such a representation is rarely, if ever, discussed in data models that use concepts such as entities and relationships, or in functional data models. Such models frequently do not have any operators either! Nevertheless, they may be useful for certain kinds of data type analysis encountered in the process of establishing a new database, especially in the very early stages of determining a preliminary informal organization. This leads to the question: What is a data model?

A data model is, of course, not just a data structure, as many people seem to think. It is natural that the principal data models are named after their principal structures, but that is not the whole story.

A data model [9] is a combination of at least three components:

(1) A collection of data structure types (the database building blocks);

(2) A collection of operators or rules of inference, which can be applied to any valid instances of the data types listed in (1), to retrieve, derive, or modify data from any parts of those structures in any combinations desired;

(3) A collection of general integrity rules, which implicitly or explicitly define the set of consistent database states or changes of state or both—these rules are general in the sense that they apply to any database using this model (incidentally, they may sometimes be expressed as insert–update–delete rules).

The relational model is a data model in this sense, and was the first such to be defined. We do not propose to give a detailed definition of the relational model here—the original definition appeared in [7], and an improved one in Secs. 2 and 3 of [8]. Its *structural part* consists of domains, relations of assorted degrees (with tables as their principal conceptual representation), attributes, tuples, candidate keys, and primary keys. Under the principal representation, attributes become columns of tables and tuples become rows, but there is no notion of one column succeeding another or of one row succeeding another as far as the database tables are concerned. In other words, the left to right order of columns and the top to bottom order of rows in those tables are arbitrary and irrelevant.

The *manipulative part* of the relational model consists of the algebraic operators (select, project, join, etc.) which transform relations into relations (and hence tables into tables).

The *integrity part* consists of two integrity rules: entity integrity and referential integrity (see [8, 11] for recent developments in this latter area). In any particular application of a data model it may be necessary to impose further (database-specific) integrity constraints, and thereby define a smaller set of consistent database states or changes of state.

In the development of the relational model, there has always been a strong coupling between the structural,

manipulative, and integrity aspects. If the structures are defined alone and separately, their behavioral properties are not pinned down, infinitely many possibilities present themselves, and endless speculation results. It is therefore no surprise that attempts such as those of CODASYL and ANSI to develop data structure definition language (DDL) and data manipulation language (DML) in separate committees have yielded many misunderstandings and incompatibilities.

4. The Relational Processing Capability

The relational model calls not only for relational structures (which can be thought of as tables), but also for a particular kind of set processing called *relational processing*. Relational processing entails treating whole relations as operands. Its primary purpose is loop-avoidance, an absolute requirement for end users to be productive at all, and a clear productivity booster for application programmers.

The SELECT operator (also called RESTRICT) of the relational algebra takes *one* relation (table) as operand and produces a new relation (table) consisting of selected tuples (rows) of the first. The PROJECT operator also transforms *one* relation (table) into a new one, this time however consisting of selected attributes (columns) of the first. The EQUI–JOIN operator takes *two* relations (tables) as operands and produces a third consisting of rows of the first concatenated with rows of the second, but only where specified columns in the first and specified columns in the second have matching values. If redundancy in columns is removed, the operator is called NATURAL JOIN. In what follows, we use the term "join" to refer to either the equi–join or the natural join.

The relational algebra, which includes these and other operators, is intended as a yardstick of power. It is *not* intended to be a standard language, to which all relational systems should adhere. The set-processing objective of the relational model is intended to be met by means of a data sublanguage[2] having at least the power of the relational algebra *without making use of iteration or recursion statements.*

Much of the derivability power of the relational algebra is obtained from the SELECT, PROJECT, and JOIN operators alone, provided the JOIN is not subject to any implementation restrictions having to do with predefinition of supporting physical access paths. A system has an *unrestricted join capability* if it allows joins to be taken wherein *any* pair of attributes may be matched, providing only that they are defined on the same domain or data type (for our present purpose, it does not matter whether the domain is syntactic or semantic and it does not matter whether the data type is weak or strong, but see [10] for circumstances in which it does matter).

Occasionally, one finds systems in which join is supported only if the attributes to be matched have the same name or are supported by a certain type of predeclared access path. Such restrictions significantly impair the power of the system to derive relations from the base relations. These restrictions consequently reduce the system's capability to handle unanticipated queries by end users and reduce the chances for application programmers to avoid coding iterative loops.

Thus, we say that a data sublanguage L has a *relational processing capability* if the transformations specified by the SELECT, PROJECT, and unrestricted JOIN operators of the relational algebra can be specified in L without resorting to commands for iteration or recursion. For a database management system to be called *relational* it must support:

(1) Tables without user-visible navigation links between them;

(2) A data sublanguage with at least this (minimal) relational processing capability.

One consequence of this is that a DBMS that does *not* support relational processing should be considered *non-relational.* Such a system might be more appropriately called *tabular*, providing that it supports tables without user-visible navigation links between tables. This term should replace the term "semi-relational" used in [8], because there is a large difference in implementation complexity between tabular systems, in which the programmer does his own navigation, and relational systems, in which the system does the navigation for him, i.e., the system provides *automatic navigation.*

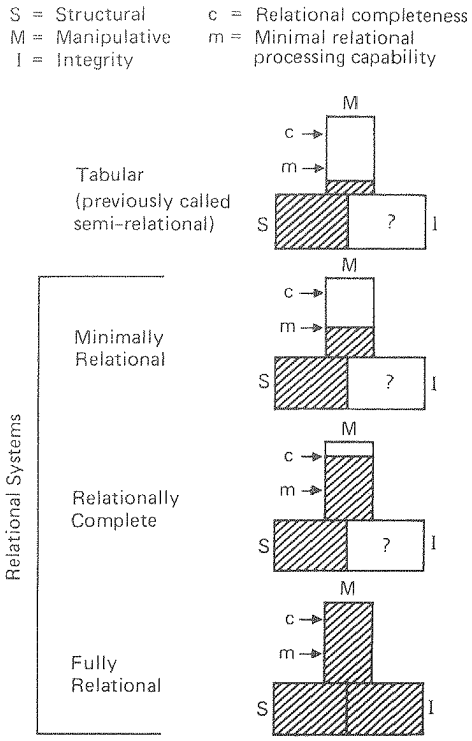
The definition of relational DBMS given above intentionally permits a lot of latitude in the services provided. For example, it is not required that the full relational algebra be supported, and there is no requirement in regard to support of the two integrity rules of the relational model (entity integrity and referential integrity). Full support by a relational system of these latter two parts of the model justifies calling that system *fully relational* [8]. Although we know of no systems that qualify as fully relational today, some are quite close to qualifying, and no doubt will soon do so.

In Fig. 1 we illustrate the distinction between the various kinds of relational and tabular systems. For each class the extent of shading in the **S** box is intended to show the degree of fidelity of members of that class to the structural requirements of the relational model. A similar remark applies to the **M** box with respect to the manipulative requirements, and to the **I** box with respect to the integrity requirements.

m denotes the minimal relational processing capability. **c** denotes relational completeness (a capability corresponding to a two-valued first order predicate logic without nulls). When the manipulation box **M** is fully shaded, this denotes a capability corresponding to the

[2] A data sublanguage is a specialized language for database management, supporting at least data definition, data retrieval, insertion, update, and deletion. It need not be computationally complete, and usually is not. In the context of application programming, it is intended to be used in conjunction with one or more programming languages.

Fig. 1. Classification of DBMS.



full relational algebra defined in [8] (a three-valued predicate logic with a single kind of null). The question mark in the integrity box for each class except the fully relational is an indication of the present inadequate support for integrity in relational systems. Stronger support for domains and primary keys is needed [10], as well as the kind of facility discussed in [14].

Note that a relational DBMS may package its relational processing capability in any convenient way. For example, in the INGRES system of Relational Technology, Inc., the RETRIEVE statement of QUEL [29] embodies all three operators (select, project, join) in one statement, in such a way that one can obtain the same effect as any one of the operators or any combination of them.

In the definition of the relational model there are several prohibitions. To cite two examples: user-visible navigation links between tables are ruled out, and database information must not be represented (or hidden) in the ordering of tuples within base relations. Our experience is that DBMS designers who have implemented non-relational systems do not readily understand and accept these prohibitions. By contrast, users enthusiastically understand and accept the enhanced ease of learning and ease of use resulting from these prohibitions.

Incidentally, the Relational Task Group of the American National Standards Institute has recently issued a report [4] on the feasibility of developing a standard for relational database systems. This report contains an enlightening analysis of the features of a dozen relational systems, and its authors clearly understand the relational model.

5. The Uniform Relational Property

In order to have wide applicability most relational DBMS have a data sublanguage which can be interfaced with one or more of the commonly used programming languages (e.g., Cobol, Fortran, PL/I, APL). We shall refer to these latter languages as *host languages.* A relational DBMS usually supports at least one end-user oriented data sublanguage—sometimes several, because the needs of these users may vary. Some prefer string languages such as QUEL or SQL [5], while others prefer the screen-oriented two-dimensional data sublanguage of Query-by-Example [33].

Now, some relational systems (e.g., System R [6], INGRES [29]) support a data sublanguage that is usable in two modes: (1) interactively at a terminal and (2) embedded in an application program written in a host language. There are strong arguments for such a *double-mode* data sublanguage:

(1) With such a language application programmers can separately debug at a terminal the database statements they wish to incorporate in their application programs—people who have used SQL to develop application programs claim that the double-mode feature significantly enhances their productivity;

(2) Such a language significantly enhances communication among programmers, analysts, end users, database administration staff, etc.;

(3) Frivolous distinctions between the languages used in these two modes place an unnecessary learning and memory burden on those users who have to work in both modes.

The importance of this feature in productivity suggests that relational DBMS be classified according to whether they possess this feature or not. Accordingly, we call those relational DBMS that support a double-mode sublanguage *uniform relational.* Thus, a uniform relational DBMS supports relational processing at both an end-user interface and at an application programming interface *using a data sublanguage common to both interfaces.*

The natural term for all other relational DBMS is *non-uniform relational.* An example of a non-uniform relational DBMS is the TANDEM ENCOMPASS [19]. With this system, when retrieving data interactively at a terminal, one uses the relational data sublanguage ENFORM (a language with relational processing capability). When writing a program to retrieve or manipulate data, one uses an extended version of Cobol (a language that does not possess the relational processing capability). Common to both levels of use are the structures: tables without user-visible navigation links between them.

A question that immediately arises is this: how can a data sublanguage with relational processing capability be interfaced with a language such as Cobol or PL/I that can handle data one record at a time only (i.e., that is incapable of treating a set of records as a single operand)? To solve this problem we must separate the following

two actions from one another: (1) definition of the relation to be derived; (2) presentation of the derived relation to the host language program.

One solution (adopted in the Peterlee Relational Test Vehicle [31]) is to cast a derived relation in the form of a file that can be read record-by-record by means of host language statements. In this case delivery of records is delegated to the file system used by the pertinent host language.

Another solution (adopted by System R) is to keep the delivery of records under the control of data sublanguage statements and, hence, under the control of the relational DBMS optimizer. A query statement Q of SQL (the data sublanguage of System R) may be embedded in a host language program, using the following kind of phrase (for expository reasons, the syntax is not exactly that of SQL)

DECLARE C CURSOR FOR Q

where C stands for any name chosen by the programmer. Such a statement associates a *cursor* named C with the defining expression Q. Tuples from the derived relation defined by Q are presented to the program one at a time by means of the named cursor. Each time a FETCH per this cursor is executed, the system delivers another tuple from the derived relation. The order of delivery is system-determined unless the SQL statement Q defining the derived relation contains an ORDER BY clause.

It is important to note that in advancing a cursor over a derived relation the programmer is *not* engaging in navigation to some target data. The derived relation is itself the target data! It is the DBMS that determines whether the derived relation should be materialized *en bloc* prior to the cursor-controlled scan or materialized piecemeal during the scan. In either case, it is the system (not the programmer) that selects the access paths by which the derived data is to be generated. This takes a significant burden off the programmer's shoulders, thereby increasing his productivity.

6. Skepticism About Relational Systems

There has been no shortage of skepticism concerning the practicality of the relational approach to database management. Much of this skepticism stems from a lack of understanding, some from a fear of the numerous theoretical investigations that are based on the relational model [1, 2, 15, 16, 24]. Instead of welcoming a theoretical foundation as providing soundness, the attitude seems to be: if it's theoretical, it cannot be practical. The absence of a theoretical foundation for almost all non-relational DBMS is the prime cause of their *ungepotchket* quality. (This is a Yiddish word, one of whose meanings is patched up.)

On the other hand, it seems reasonable to pose the following two questions:

(1) Can a relational system provide the range of services that we have grown to expect from other DBMS?

(2) If (1) is answered affirmatively, can such a system perform as well as non-relational DBMS?[3]

We look at each of these in turn.

6.1 Range of Services

A full-scale DBMS provides the following capabilities:

- data storage, retrieval, and update;
- a user-accessible catalog for data description;
- transaction support to ensure that all or none of a sequence of database changes are reflected in the pertinent databases (see [17] for an up-to-date summary of transaction technology);
- recovery services in case of failure (system, media, or program);
- concurrency control services to ensure that concurrent transactions behave the same way as if run in some sequential order;
- authorization services to ensure that all access to and manipulation of data be in accordance with specified constraints on users and programs [18];
- integration with support for data communication;
- integrity services to ensure that database states and changes of state conform to specified rules.

Certain relational prototypes developed in the early seventies fell far short of providing all these services (possibly for good reasons). Now, however, several relational systems are available as software products and provide all these services with the exception of the last. Present versions of these products are admittedly weak in the provision of integrity services, but this is rapidly being remedied [10].

Some relational DBMS actually provide more complete data services than the non-relational systems. Three examples follow.

As a first example, relational DBMS support the extraction of all meaningful relations from a database, whereas non-relational systems support extraction only where there exist statically predefined access paths.

As a second example of the additional services provided by some relational systems, consider views. A *view* is a virtual relation (table) defined by means of an expression or sequence of commands. Although not directly supported by actual data, a view appears to a user as if it were an additional base table kept up-to-date and in a state of integrity with the other base tables. Views are useful for permitting application programs and users at terminals to interact with constant view structures, even when the base tables themselves are undergoing structural changes at the *logical* level (providing that the pertinent views are still definable from the new base tables). They are also useful in restricting the scope of

[3] One should bear in mind that the non-relational ones always employ comparatively low level data sublanguages for application programming.

access of programs and users. Non-relational systems either do not support views at all or else support much more primitive counterparts, such as the CODASYL subschema.

As a third example, some systems (e.g., SQL/DS [28] and its prototype predecessor System R) permit a variety of changes to be made to the logical and physical organization of the data dynamically—while transactions are in progress. These changes rarely require application programs to be recoded. Thus, there is less of a program maintenance burden, leaving programmers to be more productive doing development rather than maintenance. This capability is made possible in SQL/DS by the fact that the system has complete control over access path selection.

In non-relational systems such changes would normally require all other database activities including transactions in progress to be brought to a halt. The database then remains out of action until the organizational changes are completed and any necessary recompiling done.

6.2 Performance

Naturally, people would hesitate to use relational systems if these systems were sluggish in performance. All too often, erroneous conclusions are drawn about the performance of relational systems by comparing the time it might take for one of these systems to execute a complex transaction with the time a non-relational system might take to execute an extremely simple transaction. To arrive at a fair performance comparison, one must compare these systems on the same tasks or applications. We shall present arguments to show why relational systems should be able to compete successfully with non-relational systems.

Good performance is determined by two factors: (1) the system must support performance-oriented physical data structures; (2) high-level language requests for data must be compiled into lower-level code sequences at least as good as the average application programmer can produce by hand.

The first step in the argument is that a program written in a Cobol-level language can be made to perform efficiently on large databases containing production data structured in tabular form with no user-visible navigation links between them. This step in the argument is supported by the following information [19]: as of August 1981, Tandem Computer Corp. had manufactured and installed 760 systems; of these, over 700 were making use of the Tandem ENCOMPASS relational database management system to support databases containing production data. Tandem has committed its own manufacturing database to the care of ENCOMPASS. ENCOMPASS does not support links between the database tables, either user-visible (navigation) links or user-invisible (access method) links.

In the second step of the argument, suppose we take the application programs in the above-cited installations and replace the database retrieval and manipulation statements by statements in a database sublanguage with a relational processing capability (e.g., SQL). Clearly, to obtain good performance with such a high level language, it is essential that it be compiled into object code (instead of being interpreted), and it is essential that that object code be efficient.

Compilation is used in System R and its product version SQL/DS. In 1976 Raymond Lorie developed an ingenious pre- and post-compiling scheme for coping with dynamic changes in access paths [21]. It also copes with early (and hence efficient) authorization and integrity checking (the latter, however, is not yet implemented). This scheme calls for compiling in a rather special way the SQL statements embedded in a host language program. This compilation step transforms the SQL statements into appropriate CALLs within the source program together with access modules containing object code. These modules are then stored in the database for later use at runtime. The code in these access modules is generated by the system so as to optimize the sequencing of the major operations and the selection of access paths to provide runtime efficiency. After this precompilation step, the application program is compiled by a regular compiler for the pertinent host language. If at any subsequent time one or more of the access paths is removed and an attempt is made to run the program, enough source information has been retained in the access module to enable the system to re-compile a new access module that exploits the now existing access paths *without requiring a re-compilation of the application program.*

Incidentally, the same data sublanguage compiler is used on ad hoc queries submitted interactively from a terminal and also on queries that are dynamically generated during the execution of a program (e.g., from parameters submitted interactively). Immediately after compilation, such queries are executed and, with the exception of the simplest of queries, the performance is better than that of an interpreter.

The generation of access modules (whether at the initial compiling or re-compiling stage) entails a quite sophisticated optimization scheme [27], which makes use of system-maintained statistics that would not normally be within the programmer's knowledge. Thus, only on the simplest of all transactions would it be possible for an average application programmer to compete with this optimizer in generation of efficient code. Any attempts to compete are bound to reduce the programmer's productivity. Thus, the price paid for extra compile-time overhead would seem to be well worth paying.

Assuming non-linked tabular structures in both cases, we can expect SQL/DS to generate code comparable with average hand-written code in many simple cases, and superior in many complex cases. Many commercial transactions are extremely simple. For example, one may need to look up a record for a particular railroad wagon to find out where it is or find the balance in someone's

savings account. If suitably fast access paths are supported (e.g., hashing), there is no reason why a high-level language such as SQL, QUEL, or QBE should result in less efficient runtime code for these simple transactions than a lower level language, even though such transactions make little use of the optimizing capability of the high-level data sublanguage compiler.

7. Future Directions

If we are to use relational database as a foundation for productivity, we need to know what sort of developments may lie ahead for relational systems.

Let us deal with near-term developments first. In some relational systems stronger support is needed for domains and primary keys per suggestions in [10]. As already noted, all relational systems need upgrading with regard to automatic adherence to integrity constraints. Existing constraints on updating join-type views need to be relaxed (where theoretically possible), and progress is being made on this problem [20]. Support for outer joins is needed.

Marked improvements are being made in optimizing technology, so we may reasonably expect further improvements in performance. In certain products, such as the ICL CAFS [22] and the Britton–Lee IDM500 [13], special hardware support has been implemented. Special hardware may help performance in certain types of applications. However, in the majority of applications dealing with formatted databases, software-implemented relational systems can compete in performance with software-implemented non-relational systems.

At present, most relational systems do not provide any special support for engineering and scientific databases. Such support, including interfacing with Fortran, is clearly needed and can be expected.

Catalogs in relational systems already consist of additional relations that can be interrogated just like the rest of the database using the same query language. A natural development that can and should be swiftly put in place is the expansion of these catalogs into full-fledged active dictionaries to provide additional on-line data control.

Finally, in the near term, we may expect database design aids suited for use with relational systems both at the logical and physical levels.

In the longer term we may expect support for relational databases distributed over a communications network [25, 30, 32] and managed in such a way that application programs and interactive users can manipulate the data (1) as if all of it were stored at the local node—*location transparency*—and (2) as if no data were replicated anywhere—*replication transparency*. All three of the projects cited above are based on the relational model. One important reason for this is that relational databases offer great decomposition flexibility when planning how a database is to be distributed over a network of computer systems, and great recomposition power for dynamic combination of decentralized information. By contrast, CODASYL DBTG databases are very difficult to decompose and recompose due to the entanglement of the owner–member navigation links. This property makes the CODASYL approach extremely difficult to adapt to a distributed database environment and may well prove to be its downfall. A second reason for use of the relational model is that it offers concise high level data sublanguages for transmitting requests for data from node to node.

The ongoing work in extending the relational model to capture in a formal way more meaning of the data can be expected to lead to the incorporation of this meaning in the database catalog in order to factor it out of application programs and make these programs even more concise and simple. Here, we are, of course, talking about meaning that is represented in such a way that the system can understand it and act upon it.

Improved theories are being developed for handling missing data and inapplicable data (see for example [3]). This work should yield improved treatment of null values.

As it stands today, relational database is best suited to data with a rather regular or homogeneous structure. Can we retain the advantages of the relational approach while handling heterogeneous data also? Such data may include images, text, and miscellaneous facts. An affirmative answer is expected, and some research is in progress on this subject, but more is needed.

Considerable research is needed to achieve a rapprochement between database languages and programming languages. Pascal/R [26] is a good example of work in this direction. Ongoing investigations focus on the incorporation of abstract data types into database languages on the one hand [12] and relational processing into programming languages on the other.

8. Conclusions

We have presented a series of arguments to support the claim that relational database technology offers dramatic improvements in productivity both for end users and for application programmers. The arguments center on the data independence, structural simplicity, and relational processing defined in the relational model and implemented in relational database management systems. All three of these features simplify the task of developing application programs and the formulation of queries and updates to be submitted from a terminal. In addition, the first feature tends to keep programs viable in the face of organizational and descriptive changes in the database and therefore reduces the effort that is normally diverted into the maintenance of programs.

Why, then, does the title of this paper suggest that relational database provides only a foundation for improved productivity and not the total solution? The

reason is simple: relational database deals only with the shared data component of application programs and end-user interactions. There are numerous complementary technologies that may help with other components or aspects, for example, programming languages that support relational processing and improved checking of data types, improved editors that understand more of the language being used, etc. We use the term "foundation," because interaction with shared data (whether by program or via terminal) represents the core of so much data processing activity.

The practicality of the relational approach has been proven by the test and production installations that are already in operation. Accordingly, with relational systems we can now look forward to the productivity boost that we all hoped DBMS would provide in the first place.

Acknowledgments. I would like to express my indebtedness to the System R development team at IBM Research, San Jose for developing a full-scale, uniform relational prototype that entailed numerous language and system innovations; to the development team at the IBM Laboratory, Endicott, N.Y. for the professional way in which they converted System R into product form; to the various teams at universities, hardware manufacturers, software firms, and user intallations, who designed and implemented working relational systems; to the QBE team at IBM Yorktown Heights, N.Y.; to the PRTV team at the IBM Scientific Centre in England; and to the numerous contributors to database theory who have used the relational model as a cornerstone. A special acknowledgement is due to the very few colleagues who saw something worth supporting in the early stages, particularly, Chris Date and Sharon Weinberg. Finally, it was Sharon Weinberg who suggested the theme of this paper.

Received 10/81; revised and accepted 12/81

References

1. Beeri, C., Bernstein, P., Goodman, N. A sophisticate's introduction to database normalization theory. *Proc. Very Large Data Bases*, West Berlin, Germany, Sept. 1978.
2. Bernstein, P.A., Goodman, N., Lai, M-Y. Laying phantoms to rest. Report TR-03-81, Center for Research in Computing Technology, Harvard University, Cambridge, Mass., 1981.
3. Biskup, J.A. A formal approach to null values in database relations. *Proc. Workshop on Formal Bases for Data Bases*, Toulouse, France, Dec 1979; published in [16] (see below) pp 299–342.
4. Brodie, M. and Schmidt, J. (Eds), Report of the ANSI Relational Task Group., (to be published ACM SIGMOD Record).
5. Chamberlin, D.D., et al. SEQUEL2: A unified approach to data definition, manipulation, and control. *IBM J. Res. & Dev., 20,* 6, (Nov. 1976) 560–565.
6. Chamberlin, D.D., et al. A history and evaluation of system R. *Comm. ACM, 24,* 10, (Oct. 1981) 632–646.
7. Codd, E.F. A relational model of data for large shared data banks. *Comm. ACM, 13,* 6, (June 1970) 377–387.
8. Codd, E.F. Extending the database relational model to capture more meaning. *ACM TODS, 4,* 4, (Dec. 1979) 397–434.
9. Codd, E.F. Data models in database management. *ACM SIGMOD Record, 11,* 2, (Feb. 1981) 112–114.
10. Codd, E.F. The capabilities of relational database management systems. *Proc. Convencio Informatica Llatina*, Barcelona, Spain, June 9–12, 1981, pp 13–26; also available as Report 3132, IBM Research Lab., San Jose, Calif.
11. Date, C.J. Referential integrity. *Proc. Very Large Data Bases*, Cannes, France, September 9–11, 1981, pp 2–12.
12. Ehrig, H., and Weber, H. Algebraic specification schemes for data base systems. *Proc. Very Large Data Bases*, West Berlin, Germany, Sept 13–15, 1978, 427–440.
13. Epstein, R., and Hawthorne, P. Design decisions for the intelligent database machine. *Proc. NCC 1980, AFIPS, Vol. 49., May 1980,* pp 237–241.
14. Eswaran, K.P., and Chamberlin, D.D. Functional specifications of a subsystem for database integrity. *Proc. Very Large Data Bases*, Framingham, Mass., Sept. 1975, pp 48–68.
15. Fagin, R. Horn clauses and database dependencies. *Proc. 1980 ACM SIGACT Symp. on Theory of Computing*, Los Angeles, CA, pp 123–134.
16. Gallaire, H., Minker, J., and Nicolas, J.M. *Advances in Data Base Theory.* Vol 1, Plenum Press, New York, 1981.
17. Gray, J. The transaction concept: virtues and limitations. *Proc. Very Large Data Bases*, Cannes, France, September 9–11, 1981, pp 144–154.
18. Griffiths, P.G., and Wade, B.W. An authorization mechanism for a relational database system. *ACM TODS, 1,* 3, (Sept 1976) 242–255.
19. Held, G. ENCOMPASS: A relational data manager. Data Base/81, Western Institute of Computer Science, Univ. of Santa Clara, Santa Clara, Calif., August 24–28, 1981.
20. Keller, A.M. Updates to relational databases through views involving joins. Report RJ3282, IBM Research Laboratory, San Jose, Calif., October 27, 1981.
21. Lorie, R.A., and Nilsson, J.F. An access specification language for a relational data base system. *IBM J. Res. & Dev., 23,* 3, (May 1979) 286–298.
22. Maller, V.A.J. The content addressable file store—CAFS. *ICL Technical J., 1,* 3, (Nov. 1979) 265–279.
23. Reisner, P. Human factors studies of database query languages: A survey and assessment. *ACM Computing Surveys, 13,* 1, (March 1981) 13–31.
24. Rissanen, J. Theory of relations for databases—A tutorial survey. *Proc. Symp. on Mathematical Foundations of Computer Science*, Zakopane, Poland, September 1978, Lecture Notes in Computer Science, No. 64, Springer Verlag, New York, 1978.
25. Rothnie, J.B., Jr. et al. Introduction to a system for distributed databases (SDD-1). *ACM TODS, 5,* 1, (March 1980) 1–17.
26. Schmidt, J.W. Some high level language constructs for data of type relation. *ACM TODS, 2,* 3, (Sept 1977) 247–261.
27. Selinger, P.G., et al. Access path selection in a relational database system. *Proc. 1979 ACM SIGMOD International Conference on Management of Data*, Boston, MA, May 1979, pp 23–34.
28. ———, SQL/Data system for VSE: A relational data system for application development. IBM Corp. Data Processing Division, White Plains, N.Y., G320-6590, Feb 1981.
29. Stonebraker, M.R., et al. The design and implementation of INGRES, *ACM TODS, 1,* 3, (Sept. 1976) 189–222.
30. Stonebraker, M.R., and Neuhold, E.J. A distributed data base version of INGRES. *Proc. Second Berkeley Workshop on Distributed Data Management and Computer Networks*, Lawrence-Berkeley Lab., Berkeley, Calif., May 1977, pp 19–36.
31. Todd., S.J.P. The Peterlee relational test vehicle—A system overview. *IBM Systems J., 15,* 4, 1976, 285–308.
32. Williams, R. et al. R*: An overview of the architecture. Report RJ3325, IBM Research Laboratory, San Jose, Calif., October 27, 1981.
33. Zloof, M.M. Query by example. *Proc. NCC, AFIPS Vol 44,* May 1975, pp 431–438.

E.F. Codd and Relational Theory

A Detailed Review and Analysis of Codd's Major Database Writings

By C.J. Date

These are extracts from the book E. F. Codd and Relational Theory: A Detailed Review and Analysis of Codd's Major Database Writings *by C. J. Date, Lulu Publishing Services, July 18, 2019, ISBN 978-1684705276. Reprinted with permission.*

Extract I: Introduction to Chapter 3 ("The Completeness Paper")

The paper I'll be referring to in this chapter as "the completeness paper" for short is just one in that series of papers by E. F. Codd that I characterized in the introduction to this part of the book as "staggering in their originality." Here's a full citation:

- "Relational Completeness of Data Base Sublanguages," IBM Research Report RJ987 (March 6th, 1972); republished in Randall J. Rustin (ed.), *Data Base Systems: Courant Computer Science Symposia Series 6* (Prentice-Hall, 1972)

Of all of Codd's early papers this one is surely the most formal, and for that reason it's probably also the one that's the most difficult to understand. Let me therefore state for the record just what the paper did:[1]

1. It defined a *relational algebra*—i.e., a set of operators, such as join and projection, for deriving a relation from some given set of relations.
2. It defined a *relational calculus*—i.e., a notation, based on predicate calculus, for defining a relation in terms of some given set of relations.
3. It defined a notion of *relational completeness* ("a basic yardstick of selective power"). Basically, a language *L* is said to be relationally complete if and only if it's as powerful as the calculus (see point 2)—i.e., if and only if for every possible expression of that calculus, there's a logically equivalent expression in that language *L*.
4. It presented an algorithm ("Codd's reduction algorithm") for reducing an arbitrary expression of the calculus as defined under point 2 to a logically equivalent expression of the algebra as defined under point 1, thereby:
   a. Showing that the algebra was relationally complete, and
   b. Providing a possible basis for implementing the calculus.
5. Finally, it offered some opinions in favor of the calculus as opposed to the algebra as a basis on which to define a general purpose relational language—what the paper referred to as "a query language (or other data sublanguage)."

Overall, the paper was dramatically different in just about every possible way from other database publications at the time. In particular, as I've indicated, it had a heavily formal (i.e., logical and/or mathematical) flavor. Now, I'm all for formalism in its place; in particular, I'm all for the precision that formalism can lead to. As I've said, however, the formalism in the case at hand had the unfortunate effect of making the paper quite difficult to understand.[2] Moreover, as a consequence of this latter fact, I believe it also had the effect of obscuring certain errors in the presentation. At least it's true that the errors in question weren't noticed, so far as I'm aware, for quite a long time. The purpose of the present chapter, then, is to clarify the contributions of the completeness paper, and also to raise a few pertinent questions.

Extract II: Section "Calculus vs. Algebra" from Chapter 3 ("The Completeness Paper")

Despite the various flaws identified in the present chapter, the algebra and the calculus and the reduction algorithm can certainly all be "cleaned up" in such a way as to meet all of Codd's original objectives in this connection—in particular, the objective that the algebra and the calculus have exactly equivalent functionality.[3] Given that equivalence, then, what are the relative merits of the two formalisms? Section 5 of the completeness paper offers some opinions in this connection. Before discussing those opinions, however, I'd like to make a few observations of a more general nature:

- *Implementation:* As noted earlier, the algebra can serve as a vehicle for implementing the calculus. That is, given a calculus based language such as QUEL or Query-By-Example, one approach to implementing that language

[1] Or what it tried to do, rather, since I'll be arguing later that at least in some respects the paper didn't quite achieve what it attempted. *Note:* Codd's 1969 and 1970 papers also touched on most of the points identified in the list of achievements I cite here. To be specific, they defined a set of algebraic operators (point 1); they suggested that predicate calculus would be a possible basis on which to define a "data sublanguage" (point 2); they suggested that such a language "would provide a yardstick of linguistic power for all other proposed data languages" (point 3); and they suggested rather strongly that predicate calculus would be, not just a possible, but in fact the *best*, basis on which to define a data sublanguage (point 5).

[2] I first read the paper myself when it was published in 1972 and I was working for IBM, and I certainly had difficulty with it—though not perhaps as much as my manager, who came into my office one day with a huge grin on his face, brandishing the paper and saying "I've found it! I've found it! I've found the piece of Codd that passeth all understanding."

[3] See, e.g., the paper "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions," by Anthony Klug (*Journal of the ACM 29*, No. 3, July 1982).

would be to take the user's original request (which is basically just a calculus expression) and apply the reduction algorithm to it, thereby obtaining an equivalent algebraic expression. That algebraic expression will consist of a set of algebraic operations, which are by definition inherently implementable.[4]

- *Purpose:* There seems to a widespread misconception concerning the purpose of the algebra (and the calculus too, come to that, but for simplicity I'll limit my attention here to the algebra specifically). To spell the point out, many people seem to think the algebra is meant purely for formulating queries[5]—but it's not; rather, it's meant for writing *relational expressions*. Those expressions in turn serve many purposes, including query but certainly not limited to query alone. Here are some other important ones:
  a. Defining views and snapshots
  b. Defining a set of tuples to be inserted, deleted, or updated (or, more generally, defining a set of tuples to serve as the source for some relational assignment)
  c. Defining constraints (though here the relational expression or expressions in question will be just subexpressions of some boolean expression)
  d. Serving as a basis for research into other areas, such as optimization and database design[6]
  e. Serving as a yardstick against which the power of database languages in general can be measured (see the next bullet item below)

  And so on (the foregoing list isn't meant to be exhaustive).

"I'm all for formalism in its place; in particular, I'm all for the precision that formalism can lead to. As I've said, however, the formalism in the case at hand had the unfortunate effect of making the paper quite difficult to understand. Moreover, as a consequence of this latter fact, I believe it also had the effect of obscuring certain errors in the presentation."

- *Relational completeness:* Codd suggests in his completeness paper that a general purpose relational language "should be at least relationally complete in the sense defined in this paper," and defines a language to be complete in this sense if and only if it has "at least the selective power of the relational calculus."[7] Given some proposed relational language *L*, then, it becomes desirable to be able to prove that *L* is relationally complete. And if the calculus and the algebra are logically equivalent, then it's sufficient (and usually easier) to show that it's as powerful as the algebra, rather than the calculus. For example, to show that SQL is relationally complete, we would need to demonstrate
  a. That there exist SQL counterparts to each of the primitive operators restriction, projection, product, union, and difference, and then
  b. That the operands to those SQL counterparts can be denoted by arbitrary SQL expressions, meaning that the SQL operators can be nested and combined in arbitrary ways.[8]

By the way, Codd stresses in his completeness paper the fact that relational completeness is only a *basic* measure of "selective power."[9] To quote:

> In a practical environment [that power] would need to be augmented by a counting and summing capability, together with the capability of invoking . . . library functions tailored to that environment.

And elsewhere:

> Relational completeness represents a very basic selective power, which in most practical environments would need to be enhanced.[10]

Despite these caveats, however, Codd notes that relational completeness means, in effect, that queries can be formulated "without resorting to programming loops or any other form of branched execution—an important consideration when interrogating a data base from a terminal."

[4] In fact, a pioneering paper on optimization—"A Data Base Search Problem," by Frank P. Palermo (IBM Research Report RJ1072, July 27th, 1972, republished in Julius T. Tou (ed.), *Information Systems: COINS IV*, Plenum Press, 1974)—is based on exactly this approach; it implements a given calculus expression by executing an appropriate sequence of algebraic operations, applying a variety of optimizations to those operations as it does so.

[5] This particular misconception is supported rather strongly by the very term *query*, of course, also by the associated term *query language*. Indeed, both of these terms occur several times in the completeness paper itself.

[6] Various "nice" properties of the algebra are important in this connection: commutativity, associativity, distributivity, and so on (thanks to Hugh Darwen for this observation).

[7] "Selective power" is Codd's term. Personally I much prefer the term *expressive* power.

[8] As a matter of fact SQL is *not* relationally complete, because it lacks the ability to take projections over the empty set of attributes and hence fails to support the crucially important relations TABLE_DEE and TABLE_DUM. PS: That business of "nesting and combining the operators in arbitrary ways" has to do with the all important property of *closure*, of course. It's interesting to note, therefore, that the completeness paper in fact never mentions that property at all!

[9] Indeed, the point is worth stressing that relational completeness doesn't necessarily imply other kinds of completeness. In particular, it certainly doesn't imply computational completeness.

[10] I can think of several other features that would need to be added too—for example, relation literals.

So which is preferable?—the algebra or the calculus? In one sense, this question is unimportant; given the complete interchangeability of the two formalisms, the difference is essentially one of style, not substance. In my own experience, it seems that people familiar with programming tend to prefer the algebra (because it's basically operators—restriction, etc.—and programmers understand operators), while end users tend to prefer the calculus (because it's a little "closer to natural language"[11]). To repeat, however, I don't really think the difference is all that important.

Note: The difference in question is sometimes characterized as being analogous to that between procedural and nonprocedural languages, in the sense that the algebra is *prescriptive* (i.e., procedural) while the calculus is *descriptive* (i.e., nonprocedural). Indeed, this characterization might even be helpful from an intuitive point of view. However, it's not really valid, of course (nor is it accurate), given the logical equivalence between the two. To repeat, the difference is really just a difference in style.

Anyway, Codd concludes his paper with some brief arguments in support of his own opinion that the calculus is to be preferred. His arguments are as follows (paraphrasing considerably):

- *Extendability:* As noted above, a real language will surely need to be able to invoke various "library functions," and extending the calculus to support such invocations seems to be straightforward:
  a. Such invocations could appear as part of the "prototype tuple," where they would have the effect of transforming the result of evaluating the alpha expression.
  b. They could also appear in place of one of the comparands in a "join term," where they would have the effect of generating the actual comparand.
  c. If they return a truth value, they could be used in place of a "join term."

Codd claims that "such enhancements readily fit into the calculus framework," whereas extending the algebra seems to be much less straightforward and would "give rise to circumlocutions." I'm not at all sure I agree with these claims, however—grafting the operators EXTEND and SUMMARIZE[12] on to the original relational algebra (which is more or less what's needed to do the job) seems fairly straightforward to me.

- *Ease of capturing the user's intent* (important for optimization, authorization, and so forth): Codd claims that because it permits the user "to request data by its properties" instead of by means of "a sequence of algebraic operations," the calculus is a better basis than the algebra for this purpose. Again I'm not at all sure I agree, however, given that any request that can be stated as a single calculus expression can equally well be stated as a single algebraic expression. *Note:* The concept of *lazy evaluation*, pioneered by the PRTV prototype, is relevant here.[13]
- *Closeness to natural language:* Codd makes the point that most users shouldn't have to deal directly with either the algebra or the calculus as such. And he continues:

> ***"Individual members had coded solutions to those problems using a variety of existing database products (IMS, TOTAL, IDS, and others), which they took turns to present to the rest of the group. Needless to say, the solutions were all quite complicated, each of them involving several pages of detailed code. So then I got up and was able to demonstrate that the five problems—which I'd come to cold (I mean, I'd never seen them before that afternoon)—could each be formulated as a single line of code in relational calculus."***

> However, requesting data by its properties is far more natural than [having to devise] a particular algorithm or sequence of operations for its retrieval. Thus, a calculus oriented language provides a good target language for a more user oriented source language.

Once again, however, I don't really agree, for essentially the same reason as under the previous bullet item. But I do want to point out that because they're more systematically defined, the calculus and the algebra are both much more suitable than SQL as such a target! (By the way, it's worth noting in passing that for implementation purposes, many SQL systems effectively convert SQL internally into something rather close to the algebra anyway.[14])

Extract III: Section "Historical Background" from Chapter 6 ("The Essentiality Paper")

There were, of course, no mainstream relational products at the time of the debate; however, there was a great deal of interest in the possibility, or likelihood, that such products might materialize in the not too distant future. If I might be permitted a personal anecdote here . . . In mid 1974, during the GUIDE 39 meeting in Anaheim, California,[15] I attended a working meeting of the GUIDE Database Programming Languages Project. The meeting was held just a few weeks after The Great Debate, possibly even in the same month—at this distance I don't recall

[11] If your knowledge of the calculus derives only from the completeness paper and nothing else, you might find this claim a little surprising! In fact, however, the calculus can easily be wrapped up in "syntactic sugar" and made much more palatable to the average user than the completeness paper might lead one to expect.

[12] Actually support for EXTEND would suffice, especially if image relations are supported as well (which I think they should be).

[13] See S. J. P. Todd: "The Peterlee Relational Test Vehicle—A System Overview," *IBM Systems Journal 15*, No. 4 (1976), also the discussion of WITH clauses in Chapter 4 of the present book.

[14] I say "rather close to" because the algebra in question can't be a true *relational* algebra, on account of the fact that SQL tables aren't true relations.

precisely. My attendance was at the invitation of the IBM representative to the project, and I was there in order to answer questions about this new thing called relational database. Participants in the project had defined a set of five sample problems, and individual members had coded solutions to those problems using a variety of existing database products (IMS, TOTAL, IDS, and others), which they took turns to present to the rest of the group. Needless to say, the solutions were all quite complicated, each of them involving several pages of detailed code. So then I got up and was able to demonstrate that the five problems—which I'd come to cold (I mean, I'd never seen them before that afternoon)—could each be formulated as *a single line of code* in relational calculus. Well, I can assure you that everyone in the group was impressed, and indeed pretty excited at the possibilities.

As I say, then, there was a lot of interest—but there was competition, too. The CODASYL Data Base Task Group (DBTG) had published its *Report* in April 1971, proposing what became known as the network approach,[16] and of course that approach had its advocates too. Not only that, but the network approach had the advantage that commercial products did already exist—indeed, had existed for some time. So the likelihood of there being, sooner or later, some kind of confrontation between advocates of the two approaches was probably fairly high. When it came to the point, though, I was the one who caused it to occur . . . Here's how it happened.

In July 1973, when I was still working for IBM in England, I had attended (again by invitation) a conference in Montreal, Canada: viz., the SHARE Working Conference on Data Base Management Systems (July 23rd-27th, 1973). I remember an awful lot of nonsense being talked at that event!—a state of affairs that made me realize that the database field was most certainly still in its infancy at the time. What made the conference interesting, though, was that Charles Bachman had just been named the 1973 ACM Turing Award winner, and he did a dry run for us of his Turing Award lecture "The Programmer as Navigator."[17]

Now, Charlie was the lead designer behind a system called IDS, which was the origin of the DBTG proposals mentioned above, and so Ted Codd and I—Ted was at the conference too, of course—already had a good idea of what he, Charlie, was going to say.[18] We also knew that the "navigational" (i.e., IDS or DBTG) ideas he was going to be describing were *not* the way to go—Ted's relational approach was. So of course Ted wanted to engage Charlie, and the audience, in an argument on the relative merits of the two schemes, right then and there. I wasn't too happy about that idea, though, because I knew that to do the job properly (a) we would need more time to prepare our position, and in any case (b) we would need more time to present that position, too—certainly more time than we could reasonably expect to be given, or would even be appropriate, in that SHARE conference environment. So I suggested to Ted that he not attempt to hold a proper debate with Charlie then and there, but rather that he issue a challenge to hold such a debate at some future event, perhaps at the next ACM SIGFIDET conference, which was due to be held in Ann Arbor, Michigan, the following May.[19] And Ted agreed.

So Charlie gave his talk. I could feel Ted almost quivering with anticipation in his seat next to me, waiting to leap up to the microphone the second Charlie finished. Of course, Ted and I both knew that the whole audience expected him to jump up and say something—probably offering some detailed technical criticism of what Charlie had presented. But what Ted actually did say was as follows, more or less: "First let me congratulate Charlie on his Turing Award. It couldn't happen to a nicer guy" (and so on and so forth) . . . Then he continued: "But none of that alters the fact that on this issue Charlie is, unfortunately, dead wrong." And he went on to say that a Q&A session following a presentation wasn't the right forum for the kind of discussion that was really needed on these matters, and therefore that he and Chris Date would like to issue a challenge (etc., etc.). And so that was the origin of what came to be called The Great Debate, which was indeed held in Ann Arbor, Michigan, the following May, and was regarded by many people as one of the defining moments in the ongoing battle between the old way of doing things and Ted's new relational way.

Now, Ted and I took this debate very seriously, and we prepared a couple of detailed technical papers to support the arguments we planned to make: viz., the two papers mentioned in the introduction to this chapter. The idea was that during the debate Ted would present the first paper and I would present the second. Unfortunately, however, the dates of the conference (May 1st-3rd, 1974) exactly clashed with the date of my move from England to California to start my California assignment for IBM, which was May 2nd. (As I recall, there was some complication or delay in my getting the necessary visas.) As a consequence I was unable to be present for the debate as such, and my presentation was given in my absence by Dionysios (Dennis) Tsichritzis, of the University of Toronto. But I don't think my absence mattered very much—Ted was clearly the man people wanted to hear, and what he had to say was far more important, as well as being much more innovative, than any contribution of mine could possibly have been. ▲

[15]GUIDE was one of the two major IBM user groups in the U.S. at the time (the other was SHARE).

[16]By the way, I think it's fair to say that the widespread perception, which persisted for many years, that there were three broad approaches to "the database problem"—viz., the relational, hierarchic, and network approaches—was due largely to my own book *An Introduction to Database Systems* (1st edition, Addison-Wesley, 1975). In writing that book, I felt that for pedagogic reasons I needed to impose some kind of structure on what might otherwise have seemed to be just one giant disorganized mess. The structure I chose was inspired by a throwaway remark in the abstract to Codd's 1970 paper ("Existing . . . data systems provide users with tree-structured files or slightly more general network models of the data"), and I divided the bulk of my book into three parts accordingly. Other books and publications then followed suit.

[17]His paper of that title was published in *Communications of the ACM 16*, No. 11 (November 1973).

[18]Forgive the familiarity here, but "Charlie" and "Ted" were how I knew and referred to Bachman and Codd, respectively, at the time, and the original notes I'm basing these explanations on weren't written to be all that formal.

[19]SIGFIDET stood for Special Interest Group on File Definition and Translation. By 1974, however, that name was beginning not to make much sense, and so that year the name was changed to SIGMOD (Special Interest Group on Management of Data).

The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win

Brian Hitchcock

Book notes by Brian Hitchcock

Details

Authors: Gene Kim, Kevin Behr, George Spafford

ISBN-13: 978-1-94278-829-4

Publication Date: February 27, 2018

Publisher: IT Revolution Press

Introduction

I was asked to review "something different" and this book is, indeed, different.

Parts Unlimited – We are shown a list of the officers of the company, followed by a list of the members of the board. A research note from an analyst tells us that the CEO is stepping down, the "Phoenix" program will restore profitability, and the drama begins.

Part 1

Chapter 1 – We learn that CIO means "Career Is Over." Our hero, Bill, has been promoted after two executives were fired. Bill doesn't want the promotion. Payroll is failing.

Chapter 2 – Bill jots down some notes; the payroll system is showing zero hours for all hourly employees. It appears that no one has bothered to figure out the sequence of events surrounding this failure.

Chapter 3 – Social Security numbers have been replaced with "squiggly characters." This reminds me of when I worked on a system that was converted to Unicode and all the Japanese Kanji were lost, replaced by all sorts of strange graphic "characters." A security-related code fix was rushed so the developer could go on vacation. It seems that developers cause all the problems and Operations always cleans them up. The security group is dismissed as hysterical. To support a security audit next week, the Social Security numbers were tokenized using some vendor code. The vendor promised its code was harmless. The change wasn't tested because there is no budget for building a test environment. No one is using the change management process.

Chapter 4 – The relationship between Development and Operations is described as tribal warfare. Corporate theater. Project Phoenix is critical, but so is getting payroll done. All of the budgeted time and money are gone, and the promised deployment date looms. The senior operations people are too busy fixing problems caused by Development to attend the meetings to design the next deployment.

Chapter 5 – The company has failed a SOX audit. Developers have administrative access to production databases.

Chapter 6 – Project Phoenix is further behind than our hero thought. There's no time for testing before the production release. It turns out that the resources needed to address the failed SOX audit will consume almost all IT resources for the next year, as currently scheduled. The SAN failed because failed drives are not being replaced due to procurement issues. There are arguments about what a "change" is. Is rebooting a server a change? Is running a database script a change?

Chapter 7 – Our hero enjoys a huge chocolate doughnut with Fruit Loops. A new board member explains that Bill doesn't understand what "work" is. I'm guessing this new board member (Erik) is the Ghost of Christmas Past, introduced to explain all the things Bill is missing. Work is deliverables, outages, and compliance. Our friend from the past regales us with stories of how the plant used to be run, how he fixed it all, and what must be done differently for IT to get fixed. Sounds like the point of our story has been introduced. We are being sold a shiny new solution to all our old problems.

Chapter 8 – Bill's request for more resources is shot down. The attempt to enforce the long-ignored change release process results in so many formal change requests that IT can't even record them all.

Chapter 9 – The credit card processing system fails; all stores are affected. There is one IT resource who has to work on everything. This resource fixes the current problem, alone, and doesn't even tell anyone that changes are being made. Bill blows a gasket, as he suspects this same IT "hero" may have caused the outage that they are now being admired for fixing.

Chapter 10 – Brent is the IT hero. As things continue to implode, Bill wonders whether Brent actually isn't as smart as they thought. Indeed. The IT hero never writes anything down. When asked how he fixed a problem, he honestly answers that he doesn't know. Brent doesn't want to give up being everyone's hero.

Chapter 11 – It becomes clear almost everything goes through the IT hero, Brent. Sixty percent of planned changes were not implemented because they were assuming that Brent

would be available to help them. We are getting close to some sort of cathartic event where the solution to all our problems will be revealed.

Chapter 12 – Project Phoenix runs in development. It won't run in production. Someone thinks to ask, and indeed they discover, that a firewall port needs to be opened. I laugh because this is exactly what I saw on my last job. Seriously, a very well-known issue came up over and over and was never fully captured. Project Phoenix is deployed, and starts "leaking" credit card numbers through the website.

Chapter 13 – The credit card issue goes public. The front-end servers for Phoenix are being rebooted hourly to prevent failures.

Chapter 14 – Senior management does what they always do. They start planning to outsource all of IT. Technology is changing too fast for anyone to keep up. Bill wishes he had stayed in the technology backwaters where he was before this promotion. The development team has a party to celebrate the Phoenix release. Operations is still in crisis mode dealing with all the failures.

Chapter 15 – Bill has the moment of clarity we have been waiting for. He now sees that there are four categories of work, just as the Ghost of Christmas Past had predicted. According to this book, the four are Business Projects, Internal IT Projects, Changes, and Unplanned Work— otherwise referred to as "Firefighting."

Chapter 16 – There is another massive failure. Bill is pleased with how much better his team is at handling this latest fiasco. But senior management is livid. Lots of drama ensues. Bill resigns.

Part 2

Chapter 17 – The company can't live without Bill. The CEO calls and begs him to come back. IT matters. Life will now be unicorns and rainbows.

Chapter 18 – They have an off-site. Touchy-feely crap. They are going to create a team where they can all trust one another.

Chapter 19 – The off-site continues. Each of the main characters relates something significant from their personal history. Our hero Bill cries as he describes his alcoholic father. Everyone is excited to put the plan in place.

Chapter 20 – The magic plan is to freeze all projects except Phoenix. What will happen when the freeze is lifted? This late in the story, the company is still experiencing Sev 1 outages. Bill meets with the Ghost of Christmas Past, Erik, again, and they take another look at the plant floor. Work centers are discussed. People like Brent will always be a limiting factor on how much work can actually get done. They discuss documenting process so other people can do it as well.

Chapter 21 – Another audit meeting. The company is able to push back and deflect some of the deficiencies.

Chapter 22 – The company is going further down the consultant path, talking about Kanban boards. Frequent service requests will be documented so that anyone can handle them. Colored cards will now help prioritize projects. They are still working on how to get Brent out as a constraint on all their work.

Chapter 23 – Handoffs are causing wait times as projects require multiple people to handle different parts of the task. We now need Kanban lanes for each task.

Chapter 24 – Bill has some time off at home. He is called into a meeting between the CFO and the head of the company audit process.

Chapter 25 – We have a list of the CFO's priorities. He must always confirm that the entire organization achieves its goal. We must understand the true business context that IT resides in.

Chapter 26 – Business process owners are interviewed to understand customer needs and wants. Bill concludes that Project Phoenix should never have been approved. It will never return what it cost.

Chapter 27 – Bill now knows the desired business outcomes. It seems that the business wants to increase revenue, market share, average order size, etc. Who knew? The plan is to pair up people in IT with the audit team to increase the security expertise. The plan is to integrate security into their daily work.

Chapter 28 – Bill's laptop, which has been broken for most of the book, is now working well. Sev 1 outages are down, recovery time is down as well. Bill feels he is hot on the trail of understanding how he and IT can help the business win. Projects are flowing much faster. His team now has biweekly outage drills. Life has been made better by following the Three Ways, which are fundamental to DevOps. These have been discussed throughout the book. First is work flowing as fast as possible, from the business to development, etc. Second is increased feedback loops. Third is a culture where experimentation and learning are encouraged. Despite all this sunshine, the head of Retail Operations is running their own shadow IT operation. A critical database migration fails; the company comes to a halt. The issue traces back to one of the changes made by the shadow IT group.

Chapter 29 – Project Phoenix is getting worse. The flow of work has improved but the batch size is too large. Too much work in progress (WIP). Deployments cause unplanned work, i.e., failures.

Part 3

Chapter 30 – Erik explains "inner-Allspaw" and "takt time." I'm not kidding. It's getting weirder and weirder. Dev and Ops must work together. A deployment pipeline is needed. IT Operations must be more like manufacturing.

Chapter 31 – Bill now wants to do ten deployments a day. Marketing must be able to make their own changes to content and business rules. But IT work is much more complex than manufacturing. More things can go wrong. The release instructions are never up to date. The solution? They will write a deployment run book that will capture all the needed information. I'm not clear how the run book will be up to date when nothing else ever has been. The solution is to create a common environment creation process that can build Dev and Production environments at the same time. It all sounds so simple! Bill marvels at how much the organization has changed.

Chapter 32 – Bill interacts with developers. Lots of sandals and skateboards. There is a lot of detail but the message is simple: the company must set up DevOps right now or the company will be split up.

Chapter 33 – Project Phoenix is slow. The answer is to spin up instances in the cloud. Security is a concern, but not to worry: the team agrees to list the top issues and prepare countermeasures. And the auditors have gone away because all the SOX deficiencies have been addressed.

Chapter 34 – Because of the newly installed DevOps, sales

(continued on page 25)

SQL 101: Which Query Is Better?—Part III

by Iggy Fernandez

Originally featured in the February 2011 issue.

In 1988, a SQL researcher named Fabian Pascal wrote an article (**http://www.dbdebunk.com/page/page/ 1317920.htm**) for *Database Programming and Design* in which he quoted Chris Date as follows:

"SQL is an extremely redundant language. By this I mean that all but the most trivial of problems can be expressed in SQL in a variety of different ways. Of course, the differences would not be important if all formulations worked equally well but that is unlikely. As a result, users are forced to spend time and effort trying to find the "best" formulation (that is, the version that performs best)—which is exactly one of the things the relational model was trying to avoid in the first place."

Pascal then went on to test seven equivalent queries with five different database engines. Only one out of the five database engines came anywhere near to acing the test; it appeared to use the same execution plan for six of the queries but did not support the seventh query. The other engines used a range of query plans with different execution times. Pascal then predicted that

"Eventually, all SQL DBMSs, for competitive reasons, will have to equalize the performance of redundant SQL expressions and to document their execution plans. Forcing users to maximize performance through query formulation is not only unproductive, but simply a lost cause, especially if there is no guidance from the system. The more users understand the relational model and its productivity intentions, the more they will demand equalized performance and documented execution plans from vendors, instead of doggedly attempting to undertake unnecessary and futile burdens."

Let's find out if Pascal's 20-year-old prediction of equalized performance came true. First, let's create tables similar to those that Pascal used. We need a table called Personnel containing employee details and a table called Payroll containing salary payments. The Payroll table is linked to the Personnel table by the Employee ID. The tests were conducted using Oracle Database 11*g* Enterprise Edition Release 11.2.0.1.0 on 32-bit Windows XP Professional Version 2002 Service Pack 3.

```
CREATE TABLE personnel
(
  empid CHAR(9) NOT NULL,
  lname CHAR(15) NOT NULL,
  fname CHAR(12) NOT NULL,
  address CHAR(20) NOT NULL,
  city CHAR(20) NOT NULL,
  state CHAR(2) NOT NULL,
  ZIP CHAR(5) NOT NULL
);

CREATE TABLE payroll
(
  empid CHAR(9) NOT NULL,
  bonus INTEGER NOT NULL,
  salary INTEGER NOT NULL
);

INSERT INTO personnel
SELECT
  TO_CHAR(LEVEL, '09999') AS empid,
  DBMS_RANDOM.STRING('U', 15) AS lname,
  DBMS_RANDOM.STRING('U', 12) AS fname,
  '500 ORACLE PARKWAY' AS address,
  'REDWOOD SHORES' AS city,
  'CA' AS state,
  '94065' AS zip
FROM
  dual
CONNECT BY LEVEL <= 9900;

INSERT INTO payroll(
  empid,
  bonus,
  salary
)
SELECT
  empid,
  0 AS bonus,
  99170 + 10000 * CEIL(DBMS_RANDOM.VALUE * 10)
    AS salary
FROM
  personnel;

CREATE UNIQUE INDEX personnel_pk
  ON personnel(empid);

CREATE UNIQUE INDEX payroll_pk
  ON payroll(empid);

ALTER TABLE personnel
  ADD CONSTRAINT personnel_pk
  PRIMARY KEY (empid);

ALTER TABLE payroll
  ADD CONSTRAINT payroll_pk
  PRIMARY KEY (empid);

ALTER TABLE payroll
  ADD CONSTRAINT payroll_fk1
  FOREIGN KEY (empid)
  REFERENCES personnel(empid);

EXEC DBMS_STATS.GATHER_TABLE_STATS(
  ownname=>'HR',
  tabname=>'PERSONNEL'
);

EXEC DBMS_STATS.GATHER_TABLE_STATS(
  ownname=>'HR',
  tabname=>'PAYROLL'
);
```

"SQL is an extremely redundant language. By this I mean that all but the most trivial of problems can be expressed in SQL in a variety of different ways. Of course, the differences would not be important if all formulations worked equally well but that is unlikely. As a result, users are forced to spend time and effort trying to find the 'best' formulation (that is, the version that performs best)—which is exactly one of the things the relational model was trying to avoid in the first place."

Observe that we gathered optimizer statistics for the newly created tables, although they are not strictly necessary for the tests that follow. A database engine can generate query plans even in the absence of statistics and should generate the same plan for all semantically equivalent queries whether or not statistics are available.

The SQL exercise is to list the employees who were paid $199,170. Here are five different ways in which a programmer might formulate the required query, three of which are from Pascal's article. Focus on the plan hash values in the query plans below. We see that Oracle uses four different query plans, meaning that Pascal's prediction of equalized performance for equivalent queries is yet to be fulfilled.

The first formulation uses an unnecessary join when only a semi-join is strictly necessary. However, in this particular case, a semi-join is equivalent to a join because the Payroll table is linked to the Personnel table by the Employee ID. Oracle uses a HASH JOIN in the EXPLAIN PLAN.

```
SELECT lname
FROM personnel, payroll
WHERE personnel.empid = payroll.empid
AND salary = 199170;

Plan hash value: 2476600843

----------------------------------------
| Id  | Operation          | Name      |
----------------------------------------
|   0 | SELECT STATEMENT   |           |
|*  1 |  HASH JOIN         |           |
|*  2 |   TABLE ACCESS FULL| PAYROLL   |
|   3 |   TABLE ACCESS FULL| PERSONNEL |
----------------------------------------

1 - access("PERSONNEL"."EMPID"="PAYROLL"."EMPID")
2 - filter("SALARY"=199170)
```

The second formulation implements a semi-join using an uncorrelated subquery. Oracle neatly converts this into a join and once again uses a HASH JOIN in the EXPLAIN PLAN. The query plan is identical to the previous one.

```
SELECT lname
FROM personnel
WHERE empid IN (
  SELECT empid
  FROM payroll
  WHERE salary = 199170
);

Plan hash value: 2476600843

----------------------------------------
| Id  | Operation          | Name      |
----------------------------------------
|   0 | SELECT STATEMENT   |           |
|*  1 |  HASH JOIN         |           |
|*  2 |   TABLE ACCESS FULL| PAYROLL   |
|   3 |   TABLE ACCESS FULL| PERSONNEL |
----------------------------------------

1 - access("EMPID"="EMPID")
2 - filter("SALARY"=199170)
```

The third formulation implements a semi-join using a correlated subquery. This time Oracle declines to convert it into a regular join.

```
SELECT lname
FROM personnel
WHERE EXISTS (
  SELECT *
  FROM payroll
  WHERE personnel.empid = payroll.empid
  AND salary = 199170
);

Plan hash value: 1844477241

----------------------------------------
| Id  | Operation           | Name      |
----------------------------------------
|   0 | SELECT STATEMENT    |           |
|*  1 |  HASH JOIN RIGHT SEMI|          |
|*  2 |   TABLE ACCESS FULL | PAYROLL   |
|   3 |   TABLE ACCESS FULL | PERSONNEL |
----------------------------------------

1 - access("PERSONNEL"."EMPID"="PAYROLL"."EMPID")
2 - filter("SALARY"=199170)
```

The fourth formulation uses a scalar subquery in the WHERE clause. Oracle uses a different query plan.

```
SELECT lname
FROM personnel
WHERE (
  SELECT salary
  FROM payroll
  WHERE personnel.empid = payroll.empid
) = 199170;

Plan hash value: 1680572657

-------------------------------------------------
| Id  | Operation                   | Name      |
-------------------------------------------------
|   0 | SELECT STATEMENT            |           |
|*  1 |  FILTER                     |           |
|   2 |   TABLE ACCESS FULL         | PERSONNEL |
|   3 |   TABLE ACCESS BY INDEX ROWID| PAYROLL  |
|*  4 |    INDEX UNIQUE SCAN        | PAYROLL_PK|
-------------------------------------------------

1 - filter(=199170)
4 - access("PAYROLL"."EMPID"=:B1)
```

The fifth formulation uses a scalar subquery in the SELECT clause. Oracle uses yet another query plan.

```
SELECT
  (
    SELECT lname
    FROM personnel
    WHERE personnel.empid = payroll.empid
  )
FROM payroll
WHERE salary = 199170;

Plan hash value: 1891291052

-----------------------------------------------------
| Id  | Operation                   | Name         |
-----------------------------------------------------
|   0 | SELECT STATEMENT            |              |
|   1 |  TABLE ACCESS BY INDEX ROWID| PERSONNEL    |
|*  2 |   INDEX UNIQUE SCAN         | PERSONNEL_PK |
|*  3 |  TABLE ACCESS FULL          | PAYROLL      |
-----------------------------------------------------

2 - access("PERSONNEL"."EMPID"=:B1)
3 - filter("SALARY"=199170)
```

Christopher Charles laments the need to tune SQL manually in a well-written piece titled "My Brilliant Career Tuning SQL" (**http://christophercharles.blogspot.com/2003/07/databases-my-brilliant-career-tuning.html**). Why was Ingres able to achieve equalized performance in 1988? Because back in 1988, Ingres internally converted SQL queries into a QUEL representation. As Pascal explains (**http://searchoracle.techtarget.com/tip/When-will-they-ever-learn**), *"QUEL was a different relational data language, one property of which was lack of subqueries, meaning that however you expressed a query in SQL, it would map to only one QUEL execution (Chris Date designed the mapping), hence the ease of optimization and consistency."* ▲

Brian's Notes *(continued from page 22)*

have gone through the roof! The company can't keep up with demand. But the competition has introduced new products. To compete would require making changes to the mainframe MRP application that was outsourced. The outsource vendor wants months to gather requirements. No problem: Bill simply buys out the vendor contract and the team writes an interface to their new DevOps environment. Bill is a hero.

Chapter 35 – Bill now has times when he literally has nothing to do. Yes, this is what the text says. Can you imagine any form of IT where you have nothing to do? I can't. The company is having a record-breaking quarter. Bill's nemesis is being fired. Bill is being promoted to COO.

Afterword – the technology value stream is discussed. The authors relate stories of how well liked their book is.

The DevOps Handbook is included. It describes Agile, Continuous Delivery, and the Three Ways.

Conclusion

A novel is a fine thing, but it is just that. The definition of a novel is fictitious prose with some degree of realism. I would like to read about a real-world complex IT system that can actually be rebuilt from scratch with a click of a mouse. Not just a web server, but an entire business system.

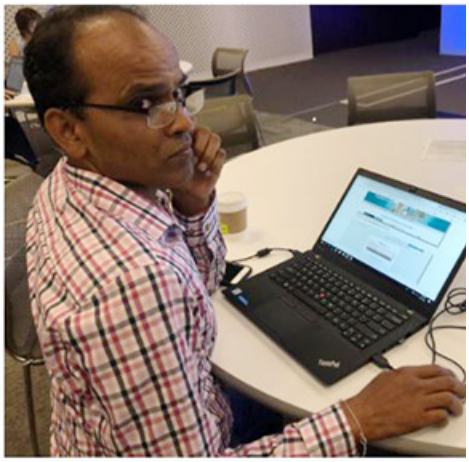
The copyright date is 2018 but all the acknowledgements are from 2012. This appears to be the second revised edition from 2014, which means it reflects the world as it was five years ago—a long time in terms of DevOps.

At the very beginning, we are told that the competition has been taking business away. Once we enter the realm of Magical Thinking, see Chapter 17, this is never brought up again. I don't think the real-world outcome would be like this. Bill's company would have been toast. Throughout the story, all the characters remain in place. I have never worked anywhere where this happened. People at all levels came and went all the time. My group didn't have the resources to even attend meetings about DevOps, let alone implement it. As I've said in the past, when reading other books, the world the authors describe sounds wonderful. I've never seen anything like these worlds, so it is difficult for me to relate. At my last job, we couldn't re-create an environment from backups because they were so complex, each one hand-crafted by people that in many cases were no longer in our group. I do sincerely hope these worlds that seem magical to me are being built. I wish I had experienced them during my career. ▲

Brian Hitchcock previously worked for Oracle Corporation, where he had been supporting Fusion middleware since 2013. Before that, he supported Fusion applications and the Federal OnDemand group. He was with Sun Microsystems for 15 years (before it was acquired by Oracle Corporation), where he supported Oracle databases and Oracle applications. Brian's contact information and all of his book reviews and presentations are available at **www.brianhitchcock.net/oracle-dbafmw/.** *The statements and opinions expressed here are his own.*

NoCOUG Conference #131

Post-Conference Kart Racing Sponsored by Quest



NoCOUG
P.O. Box 3282
Danville, CA 94526

RETURN SERVICE REQUESTED