Go Faster!

The TransRelational™ Approach to DBMS Implementation

C. J. Date

bookboon
The eBook company

C. J. Date

Go Faster!

The TransRelational™ Approach to DBMS Implementation

Go Faster!

ISBN 978-87-7681-905-7

Legal Notice

It's the best possible time to be alive,
when almost everything you thought you knew is wrong

— Tom Stoppard

Will you go a little faster?

— Lewis Carroll (more or less)

... trying not to let the joins show

— Rudyard Kipling

Good order is the foundation of all good things

— Edmund Burke

Only connect!

— E. M. Forster

… clever condensation, and accuracy, accuracy, accuracy

— Joseph Pulitzer

—◆◆◆◆◆—

I'd like to dedicate this book to all of the people at Required Technologies, Inc.,
who have been working hard to turn the ideas described herein
into a commercial reality.
I hope this book does justice to their efforts.

Contents

About the Author

C. J. Date is an independent author, lecturer, researcher, and consultant, specializing in relational database technology. He is best known for his book *An Introduction to Database Systems* (8th edition, Addison-Wesley, 2004), which has sold some 850,000 copies at the time of writing and is used by several hundred colleges and universities worldwide. He is also the author of many other books on database management, including most recently:

- From Addison-Wesley: *Databases, Types, and the Relational Model: The Third Manifesto* (3rd edition, coauthored with Hugh Darwen, 2006)
- From Trafford: *Logic and Databases: The Roots of Relational Theory* (2007)
- From Apress: *The Relational Database Dictionary, Extended Edition* (2008)
- From O'Reilly: *SQL and Relational Theory: How to Write Accurate SQL Code* (2009)
- From Trafford: *Database Explorations: Essays on The Third Manifesto and Related Topics* (coauthored with Hugh Darwen, 2010)

Another book, *Normal Forms and All That Jazz*, is due for publication in the near future.

Mr. Date was inducted into the Computing Industry Hall of Fame in 2004. He enjoys a reputation that is second to none for his ability to explain complex technical subjects in a clear and understandable fashion.

Foreword

This book went through several drafts, all of them originally written in the period 2001-2002, and all of them reviewed by persons knowledgeable in the subject area as well as by members of the intended target audience. The final version was completed in April 2002. But publication of that version was delayed for legal reasons, one of which was that it was subject to a Non Disclosure Agreement, an agreement that expired only quite recently (in September 2011, to be precise). So the text that follows was actually written almost ten years ago, and is thus necessarily out of date in certain respects. Despite this fact, I think it's worth publishing, even now, for reasons that will quickly become apparent from the text itself—not to mention the fact that I think it's a historical document, of a kind. Indeed, partly because of this latter fact, I made a deliberate decision not even to attempt to bring the text up to date. (I've made a few small editorial revisions, but they're all purely cosmetic in nature. I've also added an "About the Author" section—see page 10—that's current as of 2011, not 2002.) Please be aware, therefore, that:

- Specific figures relating to CPU speeds, disk performance, etc. must all be understood to be as of 2002. (These remarks apply primarily to Chapter 11.)
- Comments (such as they are—actually there are very few of them) regarding the capabilities of commercial DBMS products must also all be understood as referring to the products in question as they were in 2002.
- For reasons beyond the scope of this book, the company (Required Technologies, Inc.) that owns the technology described in the body of the book is currently inactive, and the website *www.requiredtech.com* mentioned in the preface is currently dormant. *Note:* "Required" is an acronym, standing for Record Extraction, Query, and Update Interface with Re-Sort Everything Design.
- A few of the references mentioned in Appendix B have been superseded by later versions.

But none of these points is of any consequence as far as the technical message of the book is concerned.

Now, my reasons for writing the book in the first place are explained in detail in the preface. What they boil down to, however, is that I was (and still am) extremely enthusiastic about the technology under discussion. And I'd like to take this opportunity to demonstrate that I'm far from being alone in my enthusiasm. Indeed, Ted Codd (inventor of the relational model) was highly enthusiastic, too—so much so that he wrote a letter of endorsement to Steve Tarin, inventor of the technology in question, and I'm pleased to be able to quote that letter here in its entirety (except for some minor items of a personal nature and some very minor editorial changes). It's dated July 10th, 2000.

> I want to congratulate you for your brilliant work on the storage representation and management of data (U.S. Patent No. 6,009,432), titled *Value-Instance-Connectivity Computer-Implemented Database* (December 28th, 1999). The mathematical underpinnings and elegance of your approach coupled with the cleverness of its implementation are dazzling, and I have rarely in my life used terms such as these in a scientific context.

For more than 30 years, I have wished that somebody would tackle this problem and develop a solution. In some ways I was sorry that I did not tackle it myself. Now with your solution, I feel that the major part of my life's work in relational database is complete. How I wish that we had your invention when the first RDBMSs were developed. It would have easily demolished the myth that relational systems could not perform well in high-performance transaction environments. Nevertheless, even today, I believe your work can revolutionize most, if not all, computer environments involving the management, storage, and access of data. It fits neatly into the original three-tier approach that I proposed in my relational writings.

As you know, in my research work developing the relational model, I very deliberately took great care to cleanly separate the user view of the data from the logical representation of the data (the relations themselves) and, similarly, separated this logical representation from the physical representation of the data in storage media. I did this for a variety of reasons. The most important of these was to insulate application programs and queries from changes in:

1. physical representation of the data,

2. successive releases of a given RDBMS,

3. movement from one RDBMS to another (inter or intra vendor).

I never specified how data was to be physically stored on media at all because I felt that additional research into the entire area of physical storage management was required, and since the three-tier scheme insulated applications from the representation of data in storage, this research could be ongoing and the results could be introduced into relational systems without impacting applications. Moreover, I also wanted to leave the door open for ingenuity and innovation among those vendors implementing RDBMSs.

Unfortunately, in the 30+ years between the date of my first publication and the granting of your patent, the challenge of storage structures and management had largely been ignored. All of the existing RDBMSs in the marketplace simply stored the logical relations (tables) and added indexes, hashing, and other rather commonplace techniques. With the advent of your technology, the kind of solution I had hoped for is at hand.

While the impact of your invention in the RDBMS arena (including the implementation of the relational operators) is self-evident, your technology can be extremely effective in the subfields of business intelligence (OLAP, data warehousing, and data mining), query/update systems, very high volume transaction systems, etc.

> [Your] invention is so exceptional and relevant to my own past work that I would be delighted to participate … and assist you in getting your work accepted.
>
> My very best wishes for every success.
>
> */ signed /*
> E. F. Codd

Now read on!

C. J. Date
Healdsburg, California
September 2014

Preface

The TransRelational™ Model—"the TR model" for short—represents a radically new, exciting, and elegant approach to implementing database management systems (DBMSs). In fact, the TR model represents a specific application of a more general technology known as the **Tarin Transform Method,** which is intended as an implementation technology for computerized data storage and retrieval systems of all kinds (not just DBMSs). The Tarin Transform Method is the subject of a United States patent—see reference [63] in the list of references in Appendix B at the back of the book—and is the intellectual property of a company called Required Technologies, Inc. My aim in what follows is to introduce the Tarin Transform Method, to describe the TR model in detail, and to show what these fundamental new ideas are likely to mean for the way we do business in the IT world (IT = information technology).

Required Technologies, Inc. has its headquarters at 130 West 42nd Street, Suite 2100, New York, New York 10036. The company was founded in January 1997. In the interests of full disclosure, I must make it clear right away that this book was written under a contract with Required Technologies; nevertheless, "the views expressed are my own," as they say—indeed, I wouldn't have signed the contract in the first place if I hadn't been so profoundly impressed with the technology.

History does repeat itself, sometimes. As you might know, I wrote a book some years back on the subject of business rules, entitled *WHAT Not HOW: The Business Rules Approach to Application Development* (reference [34] in the present book). In the preface to that earlier book, I said this:

> I must make it very clear right away that the book is not impartial. I'm very enthusiastic about business rules!—and I hope you will be too, when you've finished reading. In other words, this is definitely a book with an attitude: It explicitly champions the business rules idea, and it describes and explains what in my opinion are the merits and benefits of that idea. Why am I so enthusiastic? For two reasons: because of what the technology can do, and because it is so squarely in the spirit of "the original relational vision."

Replace the references to business rules in this extract by references to TransRelational technology instead, and the resulting text applies 100 percent to the book you're looking at right now. *Note:* I'm using the term *TransRelational technology* here ("TR technology" for short) as a synonym for the TR model, and I'll follow this practice throughout the book.

Let me elaborate briefly on that matter of the original relational vision, though. In my considered opinion, the TR model is one of the most significant advances—quite possibly *the* most significant advance—in the data management field since Codd first invented the relational model, back in 1969. In particular, as I've more or less said already, TR technology provides us with a highly effective way to implement that model—a way that is dramatically different from ways that have been tried in the past and found wanting, including *all* of the ways encountered in today's mainstream SQL products. And when I say "highly effective" here, I mean, among many other things, both of the following:

- Such an implementation would be orders of magnitude faster than today's SQL products.
- Such an implementation would deliver a far greater degree of data independence than today's SQL products.

In fact, I believe TR technology will allow us to build DBMSs that will, at last, truly deliver on the full promise of the relational model. This is a very strong claim—stronger than you might realize, perhaps—but I stand by it.

> *Aside*: As a matter of fact, Required Technologies is due to release a commercial product based on TR technology round about the time this book appears in print. Now, it's not the intent of the book to describe that product as such; however, you should be able to obtain information about that product—performance information in particular—by visiting the website *www.requiredtech.com*. *End of aside*.

Who should read this book: The book is meant for anyone interested in the field of data management who wants to learn about a truly dramatic development in that field. Thus, the target readership includes, but is not limited to, the following:

- Data and database management product implementers and other vendor personnel
- Data, database, and system administrators
- Data warehouse personnel
- Data management and DBMS professionals of all kinds
- Benchmark and performance specialists
- Computer science professors specializing in data and database management issues
- Database students, both graduate and undergraduate
- People responsible for DBMS product evaluation and acquisition
- Technically aware end-users

The only prerequisites are an elementary appreciation of data management concepts, techniques, and issues, including some knowledge of the relational model in particular (though in fact most of the pertinent relational ideas are reviewed briefly at appropriate points in the book—see in particular Chapter 2, Section 2.1, and Chapter 10, Sections 10.2-10.7). However, if you happen to be familiar with the book mentioned above on business rules [34], it's only fair to warn you that the present book is pitched at a rather more detailed technical level than that earlier book was.

How to read this book: Please note that the book is definitely meant to be read in sequence, not skipped or skimmed—except that you can omit Chapters 1 and 2, or portions thereof, if you're already familiar with the material they cover (those chapters are mostly concerned with scene setting and other background matters). Also, I think I should say that, while the book overall is meant as a tutorial, some of the details in later chapters are a little tricky (that's one reason why the material needs to be read in sequence). There are a few exercises embedded here and there in certain of those later chapters (and collected together for convenience in Appendix A), and I strongly recommend that you make the effort to do those exercises; I know from my own experience that they can help you understand the ideas very much better. There's no substitute for working through detailed examples for yourself.

That said, I should also say that the book isn't meant to cover every last detail of the TR model. It's a tutorial, not a work of reference, and the treatment is deliberately not exhaustive—in fact, it's in the nature of the TR model that it's open ended and extensible. All I've tried to do is include enough material for you to get a good feel for how TR technology is seriously different from what's been tried before, and why it offers so many significant benefits. In particular, I've tried to highlight some of the differences between this new approach and what the lawyers call "prior art"—regarding how joins are implemented, for example. What's more, the fact that I haven't tried to be exhaustive has allowed me to introduce certain conceptual simplifications (not lies!) into the presentation; very importantly, it has also allowed me to focus on the insights and intuition underlying all of the technical detail, instead of just describing that technical detail per se. The overall structure of the book is explained in Section 1.5, at the end of Chapter 1.

Acknowledgments: First of all, I'd like to thank the many people at Required Technologies, too numerous to mention individually, who supported me in various ways in the writing of this book. Second, the following people all reviewed earlier drafts of the text and made numerous helpful comments: Nagraj Alur, Lorrey James Bianchi, Charley Bontempo, Sharon Codd, Scott Cohen, Hugh Darwen, David McGoveran, Gary Morgenthaler, Fabian Pascal, Roberta Rousseau, Tom Sawyer, Steve Tarin, and Shelly Weinberg. A special vote of thanks goes to my old friend Sharon Codd for introducing me to Required Technologies in the first place, also to Gary Morgenthaler for drawing my attention to the hyperplane characterization discussed in Section 15.3, and to Roberta Rousseau for suggesting the idea of Appendix A. Thanks also to Addison-Wesley Publishing Company for permission to reuse certain earlier writings of mine as the basis for short portions of Chapters 1 and 2.

C. J. Date
Healdsburg, California
July 2014

Part I: Preliminaries

1 "Go Faster!"

1.1 Introduction

There's an old joke, well known in database circles, to the effect that what users really want (and always have wanted, ever since database systems were first invented) is for somebody to implement the **go faster!** command. Well, I'm glad to be able to tell you that, as of now, somebody finally has ... This book is all about a radically new database implementation technology, a technology that lets us build database management systems (DBMSs) that are "blindingly fast"—certainly orders of magnitude faster than any previous system. As explained in the preface, that technology is known as **The TransRelational™ Model,** or **the TR model** for short (the terms **TR technology** and, frequently, just **TR** are also used). As also explained in the preface, the technology is the subject of a United States patent (U.S. Patent No. 6,009,432, dated December 28th, 1999), listed as reference [63] in Appendix B at the back of this book; however, that reference is usually known more specifically as the *Initial* Patent, because several follow-on patent applications have been applied for at the time of writing. This book covers material from the Initial Patent and from certain of those follow-on patents as well.

The TR model really is a breakthrough. To say it again, it allows us to build DBMSs that are orders of magnitude faster than any previous system. And when I say "any previous system," I don't just mean previous relational systems. It's an unfortunate fact that many people still believe that the fastest relational system will never perform as well as the fastest nonrelational system. Indeed, it's exactly that belief that accounts in large part for the continued existence and use of older, nonrelational systems such as IMS [25,57] and IDMS [14,25], despite the fact that—as is well known—relational systems are far superior from the point of view of usability, productivity, and the like. However, a relational system implemented using TR technology should dramatically outperform even the fastest of those older nonrelational systems, finally giving the lie to those old performance arguments and making them obsolete (not before time, either).

I must also make it clear that I don't just mean that queries should be faster under TR (despite the traditional emphasis in relational systems on queries in particular)—updates should be faster as well. Nor do I mean that TR is suitable only for decision support systems—it's eminently suitable for transaction processing systems, too (though it's probably fair to say that TR is particularly suitable for systems in which read-only operations predominate, such as data warehouse and data mining systems).

And one last preliminary remark: You're probably thinking that the performance advantages I'm claiming must surely come at a cost: perhaps poor usability, or less functionality, or something (there's no free lunch, right?). Well, I'm pleased to be able to tell you that such is not the case. The fact is, TR actually provides numerous additional benefits, over and above the performance benefit—for example, in the areas of database and system administration. Thus, I certainly don't want you to think that performance is the only argument in favor of TR. We'll take a look at some of those additional benefits in Chapters 2 and 15, and elsewhere in passing. (In fact, a detailed summary of all of the TR benefits appears in Chapter 15, in Section 15.4. You might like to take a quick look at that section right now, just to get an idea of how much of a breakthrough the TR model truly is.)

1.2 TR Technology and the Relational Model

As I said in the preface, I believe TR technology is one of the most significant advances—quite possibly *the* most significant advance—in the data management field since E. F. Codd first invented the relational model (which is to say, since the late 1960s and early 1970s; see references [5-7], also reference [35]). As I also said in the preface, TR represents among other things a highly effective way to implement the relational model, as I hope to show in this book. In fact, the TR model—or, rather, the more general technology of which the TR model is just one specific but important manifestation—represents an effective way to implement data management systems of many different kinds, including but not limited to the following:

- SQL DBMSs
- Information access tools
- Object/relational DBMSs
- Main-memory DBMSs
- Business rule systems
- XML document storage and retrieval systems
- Data warehouse systems
- Data mining tools
- Web search engines
- Temporal DBMSs
- Repository managers
- Enterprise resource planning tools

as well as relational DBMSs in particular. Informally, we could say we're talking about a backend technology that's suitable for use with many different frontends. In planning this book, however, I quickly decided that my principal focus should be on the application of the technology to implementing the relational model specifically. Here are some of my reasons for that decision:

- Concentrating on one particular application should make the discussions and examples more concrete and therefore, I hope, easier to follow and understand.

- More significantly, the relational model is of **fundamental importance;** it's rock solid, and it will endure. After all, it really is the best contender, so far as we know, for the role of "proper theoretical foundation" for the entire data management field. One hundred years from now, I fully expect database systems still to be firmly based on Codd's relational model—even if they're advertised as "object/relational," or "temporal," or "spatial," or whatever. See Chapter 15 for further discussion of such matters.

- If your work involves data management in any of its aspects, then you should already have at least a nodding acquaintance with the basic ideas of the relational model. Though I feel bound to add that if that "nodding acquaintance" is based on a familiarity with SQL specifically, then you might not know as much as you should about the model as such, and you might know "some things that ain't so." I'll come back to this point in a few moments.

- The relational model is an especially good fit with TR ideas; I mean, it's a very obvious candidate for implementation using those ideas. Why? Because the relational model is at a uniform, and high, **level of abstraction;** it's concerned purely with what a database system is supposed to look like to the user, and has absolutely nothing to say about what the system might look like internally. As many people would put it, the relational model is logical, not physical.

  Let me elaborate on this point for a moment. Rather than saying it's logical, not physical, my own preference—since the terms "logical" and "physical" aren't very precisely defined—would just be to say that the relational model is indeed a model (a data model, that is) and is thus, by definition, not concerned with implementation issues. (I'll have more to say on the difference between model and implementation in the next section.) Anyway, however you might like to express the fact, it's certainly the case that the relational model emphasizes, far more than other data models do, the crucial distinction between different levels of the system—in particular, the distinction between the model or external (user) level and the implementation or internal (system) level. That's why it's a good fit with TR technology. Other data models—for example, the "object model" [3,4] or the "hierarchic model" [25,57] or the CODASYL "network model" [14,25]—muddy the distinction between those levels considerably. As a consequence, those other models give implementers far less freedom (far less than the relational model does, I mean) to adopt inventive or creative approaches to questions of implementation.

  Note: I put the terms "object model," "hierarchic model," and "network model" in quotation marks in the foregoing paragraph because there's considerable doubt as to whether those "models" are truly models at all, at least in the sense that the relational model is a model (see, for example, references [28] and [29] for further discussion of this point). Certainly most of those other "models" are quite ad hoc, instead of being firmly founded, as the relational model is, in set theory and formal logic. As I've already suggested, those other "models" also fail, much of the time, to make a clear separation between issues that truly are model issues and ones that are better regarded as implementation matters. Again, I'll have more to say on this topic in the next section.

And one further point: Although TR is an implementation technology, and thus definitely at a lower level of abstraction than the relational model, it's important to understand that it can still, like the relational model, be regarded as abstract to a degree (as indeed the very term "TR model" implies). In particular, it resembles the relational model in that it can be physically implemented in a variety of different ways. See Chapter 3 and several subsequent chapters for further discussion of this possibility.

- In my very firm opinion, the relational model is the right and proper foundation on which to build sound solutions to a variety of newer data management problems. Examples of such newer problems include user-defined data type support [40], subtyping and type inheritance support [41], and temporal data support [42]. Thus, if TR is a good basis for implementing the relational model, it follows that it should be a good basis for implementing solutions to those newer problems, too.

Actually, there's quite a lot more to be said in connection with this business of using the relational model as a vehicle for explaining TR ideas. First of all, please note that I do mean the relational model, not SQL. SQL and the relational model aren't the same thing! Indeed, considered as a concrete realization of the abstract relational model, SQL is very seriously flawed. This isn't the place to go into details on this particular issue; suffice it to say that the SQL language suffers from far too many sins, of both omission and commission, for it ever to be honestly labeled "truly relational." (For more specifics, see references [15-17], [19], [31], and [39], among others.) As a consequence, SQL is not at all suitable as a foundation for explaining TR ideas (or numerous other ideas, come to that), which is why I don't want to use it for that purpose in this book.

Another problem with SQL, possibly less serious but still significant, has to do with terminology. SQL terms are often quite actively misleading—a fact that again makes SQL unsuitable as a basis for explaining TR and other ideas. However, I will at least try to relate TR concepts and facilities to SQL constructs and terms, and I'll show examples in SQL, whenever it seems to me to make sense to do so.

In connection with the foregoing, I should add that I'll be basing all of my SQL examples on the official SQL standard [53]. A detailed tutorial on that standard (1992 version) is given in reference [39], while a brief overview of the extensions that were added to form the current (1999) version can be found in reference [47]. As you might know, however, no DBMS on the market fully supports even the 1992 version of the official standard—in fact, no DBMS could fully support it, owing to the many contradictions and inconsistencies it contains (see Appendix D of reference [39])—and so the examples might not always work exactly as advertised on your own favorite SQL product. *Caveat lector*.

While I'm on the subject of the SQL standard, by the way, let me add that the official standard pronunciation of the name "SQL" is "ess-cue-ell," though you'll often hear it pronounced "sequel." In this book, I'll favor the official pronunciation, thereby talking in terms of, for example, *an* SQL example instead of *a* SQL example.

Back to TR. Yet another important reason for explaining TR in terms of its usefulness for implementing the relational model specifically is that TR offers the possibility of building a DBMS product that truly is relational—something that, precisely because of the SQL shortcomings mentioned above (and contrary to popular belief, perhaps), has never yet been done. In other words, the potential benefits of the relational model, though well known and paid much lip service to, have never been fully realized (despite the dominance of so-called "relational" DBMSs in the marketplace), because the relational model has never been properly implemented. Now, however, we have the chance to do it right—and I very much hope that someone will be bold enough to take up this particular challenge as soon as possible.

Following on from the previous point, let me focus for a moment on one very significant "potential benefit of the relational model": **data availability and accessibility**. It was always a dream of relational advocates that end-users should be able to query and even update the database directly, without having to go through the potential bottleneck of the IT department (IT = information technology). After all, the data in the database really does belong to those end-users, not to the IT department. But this goal was never properly achieved, because of performance concerns: Database administrators were worried—with good reason—that it would be all too easy for an end-user to issue a request that would bring the system to its knees ("the query that dims the lights"). Thus, all kinds of barriers had to be put in place to prevent the real users from getting direct access to their own data: security controls, time-of-day lockouts, performance monitors, query governors, and other mechanisms. (And all of those mechanisms in turn required further administration of their own, of course, making the database administrator's job still harder.) But if performance isn't a problem—that is, if the claims regarding TR performance are indeed valid—then those mechanisms shouldn't be necessary, and we should be able, at last, to achieve the data availability and accessibility goal.

And one last point: Despite the foregoing criticisms of today's SQL products, another potential application for TR technology arises precisely in connection with those products. To be more specific, it should be possible, at least in principle, to replace the backend code in such a product by code that uses TR technology instead. The user interface—namely, SQL—to the system would remain unchanged; the only change the user would see would be that the system would now run much faster than before. (The database administrator would see a change too, in that the administration job would now be much easier.)

1.3 Model vs. Implementation

Note: This section is based on material that originally appeared in reference [34], pages 33-35, copyright (c) 2000 Addison Wesley Longman Inc. The material is reused here by permission of Pearson Education Inc.

Before I go any further, I need to say a little more about the notion of models—more precisely, data models—in general. I also need to say more about the difference between such models and their implementation (what reference [40] calls one of the great *logical differences*[1]) in particular. And I need to head off at the pass a certain confusion that might otherwise get in the way of understanding. The fact is, the term **data model** is, very unfortunately, used in the database community with two quite different meanings, and we need to be clear as to which of those two meanings is intended in any particular context.

The first meaning is the one we have in mind when we talk about, for example, the relational model in particular. It can be defined as follows:

> **Data model** *(first sense):* An abstract, self-contained, logical definition of the objects, operators, and so forth, that together make up the abstract machine with which users interact. The objects allow us to model the structure of data. The operators allow us to model its behavior.

Please note, incidentally, that I'm using the term *objects* here in its generic sense, not in the special rather loaded sense in which it's used in the world of "object orientation" and "the object model" [3,4].

And then—very important!—we can usefully go on to distinguish the notion of a data model as just defined from the associated notion of an **implementation,** which can be defined as follows:

> **Implementation:** The physical realization on a real machine of the components of the abstract machine that together constitute the data model in question.

For example, consider the relational model. The concept *relation* itself is, naturally, part of that model: Users have to know what relations are, they have to know they're made up of tuples and attributes,[2] they have to know what they mean (that is, how to interpret them), and so on. All that is part of the model. But they don't have to know how relations are physically stored inside the system, they don't have to know how individual data values are physically encoded, they don't have to know what indexes or other physical access paths exist, and so on; all that is part of the implementation, not part of the model.

Or consider the concept *join*. The join operator is part of the relational model: Users have to know what a join is, they have to know how to invoke a join, they have to know what the input and output relations look like, and so on. Again, all that is part of the model. But users don't have to know how joins are physically implemented—they don't have to know what expression transformations take place under the covers, they don't have to know what indexes or other physical access paths are used, they don't have to know what physical I/O operations are executed,[3] and so on; all that is part of the implementation, not part of the model.

In a nutshell, therefore: The model, in the first sense of the term, is **what the user has to know;** the implementation is what the user **doesn't** have to know.

(Just to elaborate for a moment: Of course, I don't mean that users aren't *allowed* to know about the implementation. They might indeed know something about it; they might possibly even use the model better if they do; but, to repeat, they don't *have* to know about it.)

Now let's turn to the second meaning of the term *data model,* which can be defined as follows:

> **Data model** *(second sense):* A model of the persistent data of some particular enterprise.

Examples might include a model of the persistent data for some bank, or some hospital, or some government department.

By the way, there's a nice analogy here that I think can help clarify the relationship between the two meanings of the term:

- A data model in the first sense is like a programming language, whose constructs can be used to solve many specific problems, but in and of themselves have no direct connection with any such specific problem.

- A data model in the second sense is like a specific program written in that language—it uses the facilities provided by the model, in the first sense of that term, to solve some specific problem.

Having now, I hope, made clear the distinction between the two meanings, I can now be explicit and say that throughout the rest of this book, I'll be using the term data model in its first ("abstract machine") sense. What's more, I'll usually abbreviate the term data model to just model, unqualified; that is, I'll take the term model, unqualified, to mean a data model specifically (barring explicit statements to the contrary, of course).

1.4 So How is it Done?

Back now to TR specifically. What then is the crucial difference between the TR approach and previous approaches to implementing the relational model? In a nutshell, it's this:

- Previous approaches have typically failed to recognize (or at least to act on) the clean separation between model and implementation that the relational model makes possible. In those systems, what the user sees and what's stored internally are, typically, very similar to one another; typically, there's a simple one-to-one correspondence between *base relations* as seen by the user and *files* as stored internally,[4] and a simple one-to-one correspondence between the tuples and attributes in such relations and the records and fields in such stored files as well (see Fig. 1.1). In other words, what's physically stored is effectively just a **direct image** of what the user logically sees.

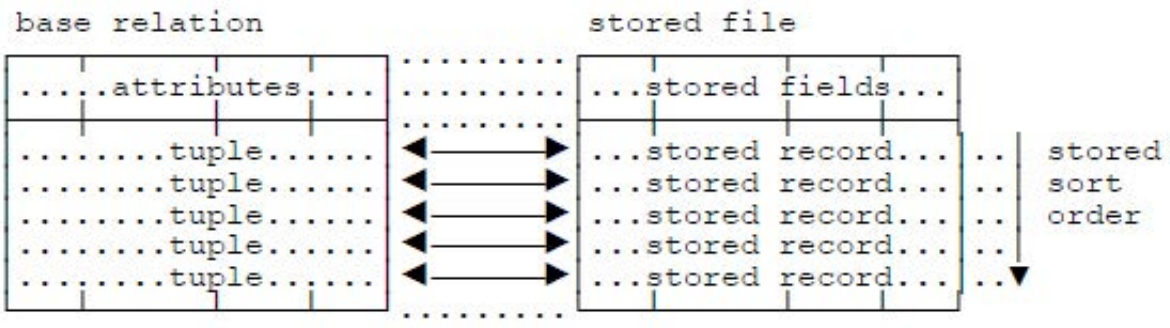


Fig. 1.1: Direct-image implementation

But that direct-image style of implementation has many undesirable consequences. One of the most important is that the tuples of the relation in question are effectively kept in just one physical sequence (that is, one "stored sort order"—see Fig. 1.1 again), and certain auxiliary structures, typically indexes, therefore have to be built and maintained in order to provide access to those tuples in any other sequence. Those auxiliary structures in turn lead to numerous further problems, including among other things stored data redundancy, additional storage space requirements, DBMS implementation complexity, physical and logical database design complications, and query and update inefficiencies and overheads. I'll elaborate on these matters in Chapter 2.

- In the TR approach, by contrast, what's physically stored is very far from being a direct image of what the user logically sees. Instead, the relations and tuples seen by the user are *transformed* into internal structures that **eliminate virtually all stored data redundancy** and **provide many stored sort orders simultaneously**. Furthermore, the transformation is done without incurring large overheads in either space or time: The transform process is rapid in both directions, and the internal structures occupy a fraction of the storage space—a figure of 20 percent is quite typical—that would be needed for the data if it were kept in raw direct-image form. (Observe, therefore, that TR is an improvement over previous approaches in terms of both space and time: Faster execution times aren't achieved at the cost of additional storage space—quite the opposite, in fact.)

And now, perhaps a little belatedly, I can explain what the term "transrelational" means. The usual meaning of "trans" is *across, beyond,* or *through*. But the "trans" in "transrelational" doesn't stand for any of these; rather, it stands for *transform* or *transformed,* and it refers to the fact that, in TR, data as seen by the user—in other words, relational data—is transformed into very different internal representations, representations that are much more suitable for internal processing purposes. Thus, TR certainly doesn't go "beyond" the relational model in the sense that it adds new logical data structures and operators to that model; rather, it goes "beyond" that model in that it introduces constructs that are explicitly oriented toward efficient implementation: constructs, in other words, that are beyond the purview of the relational model by definition.

Precisely because TR does transform the data as seen by the user instead of storing it in direct-image form, from time to time I'll talk in what follows in terms of "transform" technology explicitly, thereby highlighting the fundamental distinction between TR and the traditional direct-image approach. In Parts II and III of this book, I'll explain the TR transform process in detail; then, in Part IV, I'll step back from that level of detail and consider the fundamental significance of the transform idea.

Now, it's obviously impossible to be very specific with respect to the advantages of transform technology at such an early stage in the book. However, let me just say that I see a fruitful analogy with logarithms.[5] As we all know, logarithms allow what would otherwise be complicated, tedious, and time-consuming numeric problems to be solved by transforming them into vastly simpler but (in a sense) equivalent problems and solving those simpler problems instead. Well, it's my claim that—as I hope to show in the body of the book—*TR technology does the same kind of thing for data management problems*.

Reference [63] summarizes the distinction between TR and previous approaches (or in other words the transform vs. direct-image distinction) as follows:

> Rather than [achieving] orderedness through increasing redundancy (that is, superimposing an ordered data representation on top of the original unordered representation of the same data), the present invention achieves orderedness through eliminating redundancy on a fundamental level.
>
> *—from the Initial Patent*

In what follows, we'll see in detail exactly how these ideas are realized in practice.

1.5 Structure of the Book

The book overall is divided into four parts, plus two appendixes. A sketch of the contents follows.

Part I

Part I consists of three chapters. Following this initial chapter, Chapter 2 takes a look at the historical context; in particular, it explains the concept of direct-image implementation in more detail, and it discusses some of the problems that arise with such implementations. Chapter 3 then describes a conceptual framework, based on *three levels of abstraction,* that serves as a basis for explaining TR ideas in detail. That framework is assumed throughout the rest of the book.

Part II

Part II (seven chapters) describes the TR model. Chapters 4 and 5 in a sense form the heart of the book; they explain the two fundamental constructs of the TR model, the **Field Values Table** and the **Record Reconstruction Table,** very carefully and in considerable detail. Everything that follows builds on the ideas of these two chapters, and I recommend that you read them both as carefully as you can. In particular, they both include a number of embedded exercises, and I suggest very strongly that you attempt all of them. Working through those exercises will give you a good feel for how the fundamental TR algorithms really work—a much better feel than you can possibly get from simply reading the text.

Next, Chapter 6 addresses the issue of updates,[6] a topic that Chapters 4 and 5 scarcely consider at all (deliberately, of course). Chapters 7-9 then go on to discuss some major refinements to the basic model as described in Chapters 4, 5, and 6. Strictly speaking, the refinements in question are indeed just that, refinements, and therefore optional, but it seems to me that most if not all of them would surely be included in any commercial implementation of the TR model. What's more, several of the more significant and interesting benefits of the TR model are direct consequences of those refinements. These chapters also all include embedded exercises, and again I recommend that you take those exercises seriously.

The last chapter in Part II, Chapter 10, discusses the use of the TR model in implementing the operators of the relational model (restrict, project, join, and so forth), showing how radically different those implementations are from what we're used to seeing in traditional direct-image systems.

Part III

Divide-and-conquer is always a good pedagogical approach, and this book makes heavy use of it. In particular, Part II assumes (for the most part, at any rate) that the database is in main memory, and it ignores the complications that are introduced by the fact that real databases are usually too big to fit into memory.[7] Part III then goes on to consider what happens when we drop this assumption. Chapter 11 describes the problem in general terms; Chapters 12-14 then go on to discuss three highly TR-specific solutions to that general problem.

By the way, the point is worth making that, the foregoing paragraph notwithstanding, main-memory databases are becoming increasingly important in practice, and commercial products are becoming available that are optimized for such databases. The TR model is an excellent basis on which to build such products, as you'd probably expect.

Part IV

Part IV consists of a single wrap-up chapter (Chapter 15); it provides a summary and analysis of what's been covered in earlier chapters, including in particular a summary of the benefits the TR model provides, and it offers a brief look at what the future might hold.

Appendixes

Finally, there are two appendixes: one collecting together all of the exercises from Part II (Appendix A), and one giving a consolidated set of references for the entire book (Appendix B). Appendix A in particular is provided as a convenient place where you might actually want to work the exercises; not only does it contain the exercise statements as such, it also repeats some of the necessary background material, and it should thus save you from having to do a lot of tedious page flipping and cross-referencing while you're trying to work out your answers.

Endnotes

1. This useful term comes from Wittgenstein's dictum that *All logical differences are big differences*. For further discussion, see reference [40].
2. In case you're not familiar with these terms (or the term *relation* itself, come to that, or other related terms), they'll all be explained in Chapter 2. Here just let me note that *tuple* is usually pronounced to rhyme with "couple."
3. I/O = input/output. I'm assuming here that the data is physically stored on secondary storage media (magnetic disks, etc.).
4. I'll explain the difference between base relations and other kinds in Chapter 2. In the interests of accuracy, I should also mention that the correspondence between base relations and stored files isn't always one-to-one as I'm claiming here—some products allow several base relations to share the same stored file, and some allow a single base relation to span several stored files. However, these facts don't significantly affect the bigger picture, and ignoring them (as I plan to do from this point forward) doesn't materially affect any of the arguments I'm going to be making.
5. Thanks to Steve Tarin for suggesting this analogy.
6. Here and throughout this book, I follow convention in using the term update to refer to the INSERT, DELETE, and UPDATE operators considered generically. If I need to refer to the UPDATE operator specifically, I'll set it in all caps, as here.
7. Here and throughout this book, I follow convention in using the unqualified term *memory* to mean main memory specifically.

2 The Historical Context

2.1 Introduction

The main purpose of this chapter is to explain in more detail some of the problems that arise in connection with what the lawyers call "prior art"—meaning, in the case at hand, systems that use the traditional direct-image approach to implementation. Of course, you can skip this material if you're already familiar with conventional implementation technology. However, this first section does also introduce a few simple relational ideas, and you might at least want to make sure you're familiar with those and fully understand them.

Consider Fig. 2.1, which depicts a relation called S ("suppliers"). Observe that each supplier has a supplier number (S#), unique to that supplier;[1] a supplier name (SNAME), not necessarily unique (though in fact the sample names shown in the figure do happen to be unique); a rating or status value (STATUS); and a location (CITY). I'll use this example to remind you of a few of the most fundamental relational terms and concepts.

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Fig. 2.1: The suppliers relation S

- First of all, a **relation** can, obviously enough, be pictured as a *table*. However, a relation **is not** a table.[2] A picture of a thing isn't the same as the thing! In fact, the difference between a thing and a picture of that thing is another of the great logical differences (see the remarks on this latter notion in Chapter 1, near the beginning of Section 1.3). One problem with thinking of a relation as a table is that it suggests that certain properties of tables—for example, the property that the rows are in a certain top-to-bottom order—apply to relations too, when in fact they don't (see below).

- Each of the five suppliers is represented by a **tuple** (pronounced as noted in Chapter 1 to rhyme with "couple"). Tuples are depicted as rows in figures like Fig. 2.1, but tuples aren't rows.

- Each supplier tuple contains four values, called **attribute** values; that is, the suppliers relation involves four attributes, called S#, SNAME, STATUS, and CITY. Attributes are depicted as columns in figures like Fig. 2.1, but attributes aren't columns.

- Attributes are defined over **data types** (*types* for short, also known as *domains*), meaning that every value of the attribute in question is required to be a value of the type in question. Types can be either system-defined (built in) or user-defined. For example, attribute STATUS might be defined over the system-defined type INTEGER (STATUS values are integers), while attribute SNAME might be defined over the user-defined type NAME (SNAME values are names). *Note:* For definiteness, I'll assume these specific types throughout what follows, where it makes any difference. I'll also assume that attribute S# is defined over a user-defined type with the same name (that is, S#), and attribute CITY is defined over the system-defined type CHAR (meaning character strings of arbitrary length).

- The tuples of a relation are *all distinct*. In fact, relations never contain duplicate tuples—the tuples of a relation form a mathematical set, and sets in mathematics don't contain duplicate elements. *Note*: People often complain about this aspect of the relational model, but in fact there are good practical reasons for not permitting duplicate tuples. A detailed discussion of the point is beyond the scope of this book; see any of references [13], [20], or [33] if you want to pursue the matter.

- There's **no top-to-bottom ordering** to the tuples of a relation. Although figures like Fig. 2.1 clearly suggest there *is* such an ordering, there really isn't—to say it again, the tuples of a relation form a mathematical set, and sets in mathematics have no ordering to their elements. *Note:* It follows from this point that we could draw several different pictures that would all represent the same relation. An analogous remark applies to the point immediately following.

- There's **no left-to-right ordering** to the attributes of a relation. Again, figures like Fig. 2.1 clearly suggest there *is* such an ordering, but there really isn't; like the tuples, the attributes of a relation form a set, and thus have no ordering. (By the same token, there's no left-to-right ordering to the components of a tuple, either.) No relation can have two or more attributes with the same name.

- The suppliers relation is in fact a **base** relation specifically. In general, we distinguish between base and **derived** relations; a derived relation is one that is *derived from,* or *defined in terms of,* other relations, and a base relation is one that isn't derived in this sense. Loosely speaking, in other words, the base relations are the "given" ones—they're the ones that make up the actual database—while the derived ones are *views, snapshots, query results,* and the like [33]. For example, given the base relation of Fig. 2.1, the result of the query "Get suppliers in London" is a derived relation that looks like this:

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London

  Another way to think about the distinction is that base relations exist in their own right, while derived ones don't—they're *existence-dependent* on the base relations.

- Every relation has at least one **candidate key** (or just *key* for short), which serves as a unique identifier for the tuples of that relation. In the case of the suppliers relation (and the derived relation just shown as well), there's just one key, namely {S#}, but relations can have any number of keys, in general. *Note:* It's important to understand that keys are always *sets* of attributes (though the set in question might well contain just a single attribute). For this reason, in this book I'll always show key attributes enclosed in braces, as in the case at hand—braces being used by convention to bracket the elements that make up a set.

- As you probably know, it's customary (though not obligatory) to choose, for any given relation, one of that relation's candidate keys—possibly its *sole* candidate key—as **primary;** thus, for example, we might say in the case of the suppliers relation that {S#} is not just *a* key but *the* "primary" key. In figures like Fig. 2.1, I'll follow the convention of identifying primary key attributes by double underlining.

- Finally, relations can be operated on by a variety of **relational operators**. In general, a relational operator is an operator that takes zero or more relations as input and produces a relation as output. Examples include the well-known operators *restrict, project, join,* and so on. By the way, the—very important!—fact that the output from any given relational operation is another relation is referred to as the **closure** property (the *relational* closure property, to be more precise, because other kinds of closure are also discussed in the literature). It's the relational closure property that, among other things, allows us to write *nested relational expressions*.

Perhaps I should say a little more about the relational operators. I assume you're already somewhat familiar with the restrict, project, and join operators in particular; but if you aren't, then at least I'll be showing examples of their use at various points in subsequent chapters, and those examples should be sufficient to illustrate the general idea in each case. Nonetheless, a few words of explanation might be helpful here. In outline:

- The **restrict** operator takes a single relation as input and returns all tuples of that relation that satisfy a specified condition. Here's an SQL example:

```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY
FROM S
WHERE S.CITY = 'London' ;
```

  The result is the derived relation shown a few paragraphs back[3] (the SELECT statement in this example is, of course, an SQL formulation of the query "Get suppliers in London" discussed earlier, when derived relations were first mentioned).

- The **project** operator takes a single relation as input and returns all (sub)tuples that remain after specified attributes have been removed. Here's an SQL example:

```
SELECT S.S#, S.CITY
FROM S ;
```

  Here's the result:

S#	CITY
S1	London
S2	Paris
S3	Paris
S4	London
S5	Athens

- The **join** operator takes two relations as input (in which common attributes—that is, attributes with the same name—must be of the same type) and returns all "combined" tuples such that:

a) Each such combined tuple involves all of the attributes of the two given relations (and no other attributes); and

b) The combined tuple in question includes a tuple from each of the two relations as a subtuple.

Note that the two subtuples necessarily have common values for common attributes, by definition. Here's an SQL example:

```
SELECT FIRST.S# AS X#, SECOND.S# AS Y#
FROM S AS FIRST, S AS SECOND
WHERE FIRST.CITY = SECOND.CITY ;
```

In this particular example, the two relations being joined are in fact one and the same. What's more, the example doesn't *just* involve a join—it does involve a join, of course, but then it takes a projection of the result of that join over two attributes, and then it renames those two attributes. The final result looks like this:

X#	Y#
S1	S1
S1	S4
S2	S2
S2	S3
S3	S2
S3	S3
S4	S1
S4	S4
S5	S5

Note: We could tidy up this result by extending the WHERE clause in the original SQL formulation to include the specification "AND FIRST.S# < SECOND.S#" (immediately before the semicolon). The result would then look like this:

X#	Y#
S1	S4
S2	S3

If you want to learn more about the relational model, and more about the relational operators in particular, please refer to the tutorial treatment in reference [33]. A more formal treatment can be found in reference [40]. Reference [35] might also be helpful.

Incidentally, you're probably well aware that, in SQL, relations, tuples, and attributes are called *tables, rows,* and *columns,* respectively. You might also find these terms less intimidating (more user-friendly) than *relations, tuples,* and *attributes*. As I've tried to show, however, there are good reasons ("a relation is not a table," etc.) for using the more formal terms. In fact, I want to come back later and say more about this question of terminology—in particular, I want to explain in more detail just why I want to use the more formal terms rather than the SQL ones—and I'll do that in Chapter 3.

So much for relational terminology and concepts (for now, at any rate). In later sections of this chapter, I'll use the suppliers relation of Fig. 2.1 as a basis for showing how relations, and operations on relations, are typically implemented in today's SQL products. First, however, I need to say a little more about the concept of *ordering*.

2.2 Ordering

As explained in the previous section, the relational model has no concept of ordering—neither left-to-right attribute ordering, nor top-to-bottom tuple ordering. Of course, these omissions are deliberate; there are many reasons why it would be a bad idea to include any concept of ordering at the relational level (see, e.g., reference [33] if you want further explanation). However, ordering is of considerable importance, for a variety of pragmatic reasons, at other (nonrelational) levels of the system. To be specific, it's important at the *presentation, internal,* and *hardware* levels. To elaborate:

- At the **presentation** level, users usually like to see certain top-to-bottom and left-to-right orderings when results are extracted from the system and displayed or printed. In fact, of course, it's virtually impossible to display or print results *without* such orderings, owing to the inherent nature of display and print media; by way of an example, consider Fig. 2.1! But, of course, human users usually want the orderings in question to be meaningful ones, not just arbitrary—again, see Fig. 2.1 for an example—because such orderings can help the user to grasp more readily the real import of what's being displayed or printed.[4] That's why relational systems typically provide an ORDER BY operator, whose purpose is to arrange the tuples of some specified relation into some specified top-to-bottom sequence. For example, Fig. 2.1 might represent the result—the ordered result, that is, with top-to-bottom tuple ordering—from the following SQL query:

```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY
FROM S
ORDER BY S# ;
```

  Points arising:

  - Note very carefully that although the input to ORDER BY is a relation, *the output isn't;* by definition, the output is ordered, and relations aren't. (The output can be thought of instead as a *list*—or a *sequence,* or a *vector,* or a *one-dimensional array*—of tuples.) Thus, when I say that Fig. 2.1 might represent the result of the foregoing query, what I mean is that it might be *interpreted* as representing that result, instead of (as previously) being interpreted as representing the suppliers relation as such. In other words, while ORDER BY is certainly a respectable and useful operator, it isn't a relational operator as such, and it isn't part of the relational model.

  - In SQL in particular, the ORDER BY clause orders the tuples top to bottom, while the SELECT clause orders the attributes left to right. Thus, we might further interpret Fig. 2.1 as representing the result of the foregoing SQL query with its specified top-to-bottom tuple ordering *and* with its specified left-to-right attribute ordering.

- At the internal level, many implementation algorithms rely on the ability to access the stored data in some specific sequence. For example, it's well known that—at least in a conventional system—sort/merge is one of the most efficient ways of implementing the relational join operation [59]. That is, a good way to evaluate the relational expression *A* JOIN *B*, where the join is to be done on the basis of values of some common attribute *C*, is as follows:

  1. Sort the stored version of the tuples of relation *A* into sequence according to values of the common attribute *C*. Call the result *List 1*.

  2. Do the same for relation *B,* calling the result *List 2*.

3. Combine ("merge") entries from *List 1* and *List 2* that have the same value for the common attribute *C*. This process can be done in a single pass over each of the two lists (that's why it's efficient).

Clearly, this algorithm can be made considerably more efficient if the stored versions of *A* and *B* are in the desired sequence already, because then the two sorts won't be necessary.

- At the **hardware** level, it's in the nature of essentially all physical storage media—from main memory to disks and tapes and all the way up and down the storage hierarchy—that there's an inherent ordering to the way the medium is physically accessed. Clearly, therefore, it's desirable that the ordering in question be put to some good use; that is, we'd like to store the data in such a way that the sequence in which it has to be physically accessed corresponds to the sequence in which the implementation most often needs it.

From all of the above, it follows that there are a couple of important implementation challenges to be faced. First, at the hardware level, there's clearly only one physical sequence available, so we definitely want to make the best use of it we can; what's the best way to exploit the single physical ordering that's available to us? Second, both users and the implementation often need to see the data in an ordering other than that single physical one; how can we impose additional orderings on the stored data, over and above that physical one?

2.3 Indexing

Note: This section and the next three are based on material that originally appeared in reference [26], pages 724-743, copyright (c) 1995 Addison Wesley Longman Inc. The material is reused here by permission of Pearson Education Inc.

Now let's get back to the suppliers relation of Fig. 2.1. For the rest of the chapter, I'm going to assume—as SQL systems typically do assume—that (a) each tuple of that relation maps to its own stored record at the internal level, and (b) that stored record looks very much like the corresponding tuple; in other words, I'm assuming what in Chapter 1 I called a **direct-image** style of implementation. I'm also going to assume that

a) Those stored records together constitute a stored file,[5] and

b) That stored file is physically stored in supplier number sequence, as suggested by Fig. 2.1 (first supplier S1, then supplier S2, and so on).

Thus, we can now reinterpret Fig. 2.1, no longer as a picture of a relation as such, but rather as a picture of a possible *stored representation* of that relation. Fig. 2.2 is a revised version of Fig. 2.1 that highlights significant aspects of that reinterpretation; note in particular that it emphasizes the top-to-bottom sequence of records (in the next chapter, we'll be concerned with the left-to-right sequence of fields, too). Note also that there's no double underlining in Fig. 2.2; that's because double underlining is used to indicate primary keys, and primary keys are a relational concept, not a stored-file one. Of course, we could define an analogous concept for stored files too, but keeping it as a purely relational notion helps to distinguish pictures of relations like Fig. 2.1 from pictures of stored files like Fig. 2.2.

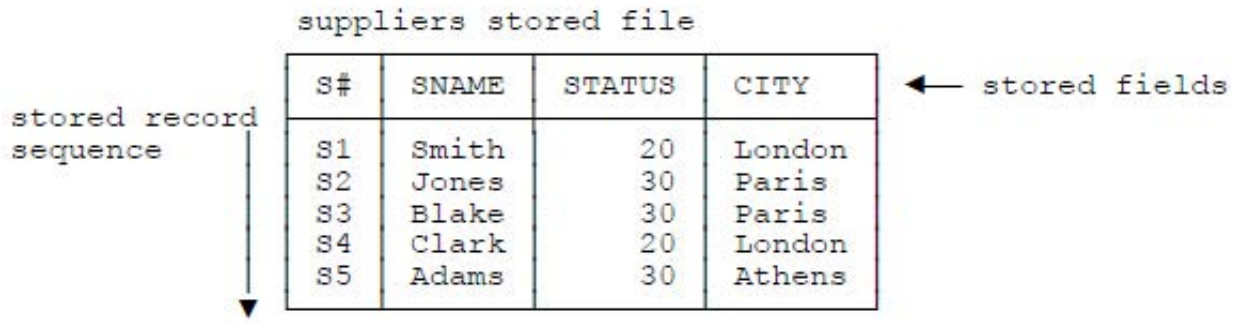


Fig. 2.2: Direct-image representation of suppliers

The question we now need to consider, then, is this: How can we impose additional orderings on the stored file illustrated in Fig. 2.2—for example, an ordering that lets us access suppliers in city name sequence (Athens, then London, then Paris)?

The historical answer to this question has always been to introduce certain **auxiliary storage structures,** whose purpose is precisely to represent such additional orderings and to provide what are called *alternative access paths* to the data. **Indexes** are probably the best-known example of such a structure. In the case at hand, for example, we might create a *city index* as shown in Fig. 2.3.

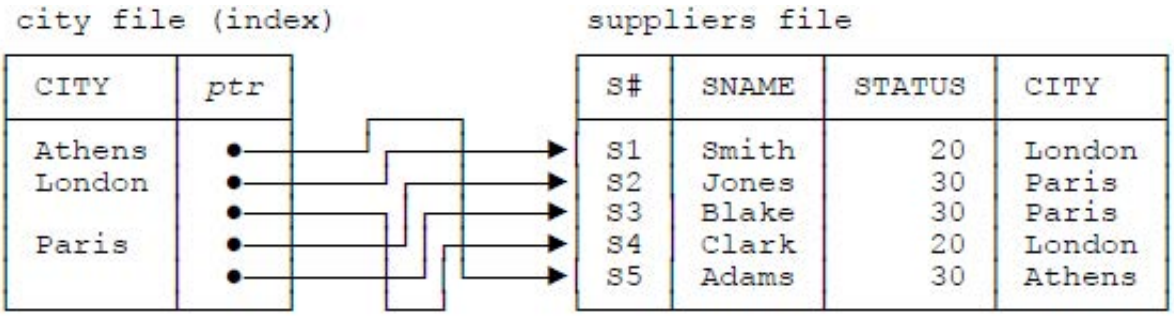


Fig. 2.3: Suppliers with a city index

Observe now that, as the figure suggests, the city index itself constitutes another stored file, and its records—namely, the three index entries—are physically stored in (let's assume) city name sequence. Each index entry contains a city name, together with **pointers** to all of the stored records for suppliers located in the city in question. Thus, if the user issues the following SQL query—

```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY
FROM S
ORDER BY CITY ;
```

—then the system can obtain the desired result in the desired sequence by carrying out a sequential search on the city index and following each pointer in turn to the corresponding record in the suppliers file.

Perhaps I should digress for a moment to explain what I mean by the term *pointer*. Basically, a pointer *variable* is a variable whose permitted values are pointer values, and a pointer *value* is, conceptually, the address of some location in storage—though it doesn't have to be a physical address, of course, and in fact it usually isn't, especially if the storage we're talking about is secondary storage specifically (see the remarks below regarding logical disk addresses). *Note:* From this point forward, I'll use the unqualified term *pointer,* sloppily but conveniently, to mean either a pointer variable or a pointer value, depending on context.

Now, it should be clear that tangles of arrows like those in Fig. 2.3 can make such figures rather messy and difficult to follow. For that reason, from this point forward I'll favor a different style: I usually won't try to show pointers as arrows—instead, I'll show the corresponding pointer values. Fig. 2.4 is a revised version of Fig. 2.3 that takes this latter approach. Observe that the supplier records have been given sequential *record numbers,* and those record numbers have then been used as *references* to those records—that is, as appropriate pointer values—within the city index. (By the way, the index entries are stored records too, as we already know, and they therefore have record numbers of their own. However, I haven't tried to show those in Fig. 2.4.)

city file (index)

CITY	ptr
Athens	5
London	1
	4
Paris	2
	3

suppliers file

	S#	SNAME	STATUS	CITY
1	S1	Smith	20	London
2	S2	Jones	30	Paris
3	S3	Blake	30	Paris
4	S4	Clark	20	London
5	S5	Adams	30	Athens

Fig. 2.4: Fig. 2.3 revised to show pointer values (record numbers)

Of course, what I'm here calling "record numbers" won't really be simple sequential numbers, as such, in any real implementation; instead, they'll be some kind of logical disk address (assuming the records are kept in secondary storage)—for example, a page and offset address [26], or some other kind of address that doesn't change if the records are physically moved around in storage. (Sequential numbers would clearly be a problem if, for example, a new record were inserted somewhere in the middle.) Despite this fact, however, I'll continue to show sequential numbers in figures like Fig. 2.4, for obvious reasons of simplicity.

I should say too that those record numbers are indeed, as suggested, best thought of as addresses, not as part of the records they refer to. Certainly they aren't physically stored as part of the records they refer to, which is why Fig. 2.4 shows them separately, alongside but separate from the suppliers file itself.

Let's get back to the question of indexing. In general, of course, a given stored file can have any number of associated indexes; in the case of suppliers, for example, we might have a status index as well as the city index just discussed. We might even have an index on everything—a supplier number index, a supplier name index, a status index, and a city index—in which case the suppliers file would be said to be "fully inverted" [25,26]. (In fact, we might go further still, using a technique called *combined indexes* [56,61,64], but to discuss that idea in detail would take us too far beyond the scope of the present discussion.)

Observe next that, as you probably know, indexes can be used to provide direct as well as sequential access to the indexed file. For example, consider the query "Get suppliers in London" (which involves a relational restrict operation, recall). If there's a city index, then the system now has two strategies available to it for implementing that query:

- *Sequential search:* Do a sequential search on the suppliers file, looking for all records with city name equal to London.

- *Direct lookup via the index:* Do a sequential search on the city index, looking for the London entry, and then follow the pointers to the corresponding records in the suppliers file.

But this fact—the fact, that is, that there are now two strategies—makes life harder for the implementation. The two strategies will clearly have different performance characteristics. For example, if the ratio of London suppliers to others is small, then direct lookup via the index will probably be more efficient; by contrast, if most suppliers are in fact in London, then sequential search will probably be more efficient. So the implementation now needs to include a new component, the **optimizer,** whose job among other things is to select the appropriate access path to use in implementing any given query.

Now, it's easy to see that access path selection has the potential to be a seriously complex business. For example, in the case at hand, does the optimizer know—and if so, *how* does it know—what the ratio of London suppliers to others is? If it does know, then how does it pick a threshold ratio for deciding between the two strategies? If the user asks to see suppliers in status sequence instead of city name sequence and there's no status index, is it better to sort the data dynamically or to build a new status index on the fly? If the user asks to see suppliers in London with status 20, and if city and status indexes both exist, which one should be used? Perhaps both? Perhaps neither? And so on and so forth.

Perhaps you can begin to understand why I said in Chapter 1 that indexes and other auxiliary structures can lead to DBMS implementation complexity. In fact, I said such structures can lead to a variety of other problems as well as complexity. Let's take a quick look at some of the others.

- *Stored data redundancy:* This one's obvious. For example, there's clearly more repetition of city names in Fig. 2.4 than there was in Fig. 2.2. Think what has to be done if a city changes its name—New Amsterdam becomes New York, Peking becomes Beijing, Bombay becomes Mumbai, St. Petersburg becomes Leningrad and vice versa, and so on and so forth. Of course, changing a city name is a problem in Fig. 2.2 too, but the problem is considerably worse in Fig. 2.4 (remember that the city index has to be physically kept in city name sequence).

- *Additional storage space requirements:* This one's obvious, too; again, compare Figs. 2.4 and 2.2. A heavily indexed file can easily occupy several times as much storage as the raw data by itself.

- *Physical database design complications:* How do we decide what indexes should exist? *Who* decides? Can the system itself give us any guidance? What about query vs. update tradeoffs (see further discussion below)? And what about tuning the system for performance (for example, adding and dropping indexes) as and when performance requirements change? By the way, note the implied need here for performance monitoring tools and performance tuning "knobs" as well—meaning more implementation complexity and more database administration headaches.

- *Logical database design complications:* In principle, indexes and the like are a physical design consideration only and should have no impact on logical design. Precisely because of the direct-image approach to implementation adopted in most of today's systems, however, physical design considerations have a habit in practice of reflecting back on the logical design (see the discussion of "data independence" at the very end of this chapter).

- *Query inefficiencies and overheads:* The inefficiencies occur if the optimizer does a less than perfect job in the access path selection process (which in practice it probably will). The overheads occur in carrying out that access path selection process in the first place.

- *Update inefficiencies and overheads*: The very same inefficiencies and overheads that occur with queries clearly occur with updates as well. However, there's another, perhaps more important, consideration to be taken into account in connection with updates. To be specific, while an index might perhaps be used to speed up queries, it will at the same time slow down updates. For example, every time a new supplier is inserted (meaning a new stored record has to be added to the suppliers file), a new entry also has to be added to the city index. By way of another example, consider what the system has to do to the city index of Fig. 2.4 if supplier S2 moves from Paris to Rome.

There's one more point—an important one—to be made in connection with the foregoing list of problems. As we've seen, indexes can be used to impose different orderings on a given stored file and thus (in a sense) "level the playing field" with respect to different processing sequences; all of those sequences are equally good from a logical point of view. But they certainly aren't equally good from a performance point of view. For example, even if there's a city index, processing suppliers in city name sequence will involve, in effect, random accesses to storage, precisely because the supplier records aren't *physically* stored in city name sequence but are scattered all over the disk. In fact, at most one index on a given stored file can have a logical sequence that matches the physical sequence in which the records of that file are stored. (In the case of the suppliers file, given our assumption that the physical sequence is by S# value, that index would have to be a supplier number index specifically.) That special index, if it exists, is said to be a **clustering** index; sequential access via a clustering index is reasonably efficient, because the logical sequence defined by the index—which corresponds to the physical sequence of the index itself—reflects the physical sequence of the indexed file as well.

By the way, you might be thinking, in the case of suppliers specifically, that there wouldn't be much point in having a supplier number index. Why not? Because such an index would apparently have the same number of entries as the indexed file has records (since supplier numbers are unique), and searching that index would thus be just as slow as searching the indexed file itself. In practice, however, index entries usually point, not as I've been pretending to individual records in the indexed file, but rather to blocks (or pages) of records in that file. As a consequence, a supplier number index might indeed make sense after all. In fact, today's SQL products almost certainly would have such an index—in part because such an index is typically used as the means of enforcing the required uniqueness constraint.

Let me close this section by noting that, of course, what I've said so far has merely scratched the surface of the topic of indexing in general. The index structures found in real DBMSs are usually much more sophisticated than those I've been describing. For example, some products support join indexes [68], which are indexes that index two stored files at the same time and are specifically intended to improve the performance of certain join operations. Likewise, some products support bitmap indexes, which are indexes that are specifically designed to improve the performance of certain other retrieval operations.[6] In other words, numerous refinements on the basic indexing idea have been developed over the years. However, all of those refinements suffer from the same basic problems, pretty much; thus, the discussions above should be sufficient to illustrate the idea of indexing in general and to show what some of those problems are.

Note: Remarks analogous to those of the foregoing paragraph apply to the topics of the next three sections as well, as you'd probably expect. If you'd like to read more about those topics (or about indexing, of course), or about a variety of related topics that are beyond the scope of this book, then I recommend any of references [50], [55], or [69].

2.4 Pointer Chains

Pointer chains are another auxiliary structure that can be used to impose additional orderings on a given stored file.[7] For example, suppose again, as in the previous section, that we want to be able to access suppliers in city name sequence. Then we might create the pointer chain structure shown in Fig. 2.5.

city file (parent)

	CITY	*ptr*
p1	Athens	*c5*
p2	London	*c1*
p3	Paris	*c2*

suppliers file (child)

	S#	SNAME	STATUS	*ptr*
c1	S1	Smith	20	*c4*
c2	S2	Jones	30	*c3*
c3	S3	Blake	30	*p3*
c4	S4	Clark	20	*p2*
c5	S5	Adams	30	*p1*

Fig. 2.5: Suppliers with a city parent file

As you can see, there are two stored files in the figure, a suppliers file and a city file, much as there were in the indexing example in Figs. 2.3 and 2.4. This time, however, the city file is not an index but what is sometimes called a **parent** file; the suppliers file is accordingly called a **child** file, and the overall structure is an example of **parent/child organization**. The parent file—which is stored (let's assume) in city name sequence—contains one stored record for each distinct city, giving the city name and acting as the head of a chain of pointers connecting together all of the child records for suppliers in that city. For example, the parent record for Paris contains the pointer value *c2;* child record *c2* (the one for supplier S2) contains the pointer value *c3;* and child record *c3* (the one for supplier S3) contains the pointer value *p3,* which takes us back to the parent record for Paris again. Note that the city names have been removed from the suppliers file, thereby eliminating some data redundancy.

If the user now issues the same SQL query as in the previous section—

```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY
FROM S
ORDER BY CITY ;
```

—then the system can obtain the desired result in the desired order by doing a sequential search on the city file and, for each record in that file in turn, following the pointer chain to the corresponding records in the suppliers file.

In general, of course, a given stored file can have any number of associated parent files and pointer chains, just as it can have any number of indexes. For example, we might have a status parent file for suppliers as well as the city parent file just discussed. What's more, pointer chain structures can be used to provide direct access as well as sequential access to the child file, again as with indexes. For example, to find all suppliers in London (a relational restrict operation again), the system can do either of the following:

- *Sequential search:* Do a sequential search on the suppliers file, and, for each supplier in turn, follow the pointer chain to the city file to find out whether the supplier is in London.

- *Direct lookup via the parent file:* Do a sequential search on the city file, looking for the London entry, and then follow the pointer chain to find all corresponding records in the suppliers file.

In general, pointer chain structures have advantages and disadvantages analogous to, though not the same as, those that apply to index structures. It's not worth going into those advantages and disadvantages in detail here; let me just make a few pertinent observations.

- The principal advantage of pointer chains over indexes is that the insert/delete algorithms are somewhat simpler [26]. Also, the parent/child structure in the example is less redundant than, and will probably occupy less storage space than, the corresponding index structure, because each city name appears exactly once instead of several times.

- The principal disadvantages are as follows:

  - For a given city, the only way to access the *N*th supplier in that city is to follow the chain and access the 1st, 2nd, ..., (*N*−1)st supplier too.

  - Processing suppliers in city name sequence will still typically involve random accesses to storage, unless the supplier records happen to be physically stored ("clustered") in city name sequence, which in practice they probably won't be.

  - Although the pointer chain structure might help with the query "Get the suppliers in a given city," it's of no help—in fact, it's a positive hindrance—for the inverse query "Get the city for a given supplier" (contrast the situation with indexes).

  - Imposing a new pointer chain structure on an existing stored file is a highly nontrivial matter (partly because the pointer chains actually run through the stored records); in fact, such an operation will typically require a database reorganization.[8] By contrast, it's a comparatively straightforward matter to create a new index over an existing stored file.

2.5 Hashing

After indexes, the auxiliary structures most widely used in practice are probably **hash** structures. Such structures are very good for direct access but typically don't support sequential access at all; in fact, hash structures, unlike the index and pointer chain structures described in previous sections, aren't usually thought of as a way of imposing ordering as such (unless the hash function—see later—is "order-preserving" [51], which in practice it usually isn't). But hashing is important, and it deserves a brief discussion here.

Hashing is a technique for providing fast access to a specific stored record on the basis of a given value within that record. It works as follows (in outline):

- Each stored record is placed in storage at a location whose address is computed as some function (the **hash function**) of some specific value contained within that record. The computed address is called the **hash address**.

- To store the record initially, the system computes the hash address for the new record and stores the record at that location. To access the record subsequently, the system performs the same computation as before and goes to the record at the computed address.

By way of a simple example, suppose we have supplier records for suppliers S100, S200, S300, S400, S500 (instead of S1, S2, S3, S4, S5, respectively), and consider the following hash function:

```
hash address = MOD ( numeric part of S# value, 13 ) + 1
```

(where MOD stands for "modulus"; the expression MOD(*a,b*) returns the remainder after dividing *a* by *b*). This is a trivial example of a very common class of hash function called **division/remainder**. (For reasons beyond the scope of this book, the divisor in a division/remainder hash is usually chosen to be prime, as in our example.) The hash addresses for our five suppliers are thus *10, 6, 2, 11,* and *7,* respectively, giving us the hash structure shown in Fig. 2.6.

suppliers file

	S#	SNAME	STATUS	CITY
1				
2	S300	Blake	30	Paris
3				
4				
5				
6	S200	Jones	10	Paris
7	S500	Adams	30	Athens
8				
9				
10	S100	Smith	20	London
11	S400	Clark	20	London
12				
13				

Fig. 2.6: Suppliers with a supplier number hash

In addition to showing how hashing works, the example also shows why the hash function is necessary. It would theoretically be possible to use an *identity* hash function, thereby storing the record for supplier S100 at location *100,* the record for supplier S200 at location *200,* and so on. Such a technique would generally be inadequate in practice, however, because the range of values to be hashed will usually be much greater than the range of available addresses. For instance, suppose supplier numbers are in fact in the range S000-S999, as in the example; then there would be 1000 possible distinct supplier numbers, whereas there might in fact be only ten or so actual suppliers at any given time. In order to avoid a considerable waste of storage space, therefore, it would be nice to find a hash function that will map any value in the range 000-999 to one in the range *1-10* (say). To allow room for growth, it's usual to extend the target range by 25 percent or so; that's why I chose a function in the example that generated values in the range *1-13* instead of *1-10*.

The example also shows clearly why, as mentioned earlier, hashing typically provides no support for sequential access. Indeed, the sequence of records within the hash structure will almost certainly not have any sensible logical interpretation (again, unless the hash function is order-preserving).[9] What's more, there will typically be gaps of arbitrary size between consecutive stored records.

Another disadvantage of hashing in general is the possibility of **collisions**—that is, two or more distinct records might hash to ("collide at") the same hash address. For example, suppose the suppliers file (with suppliers S100, S200, etc.) also includes a supplier with supplier number S139. Given the hash function "divide by 13 and add one" as discussed above, that supplier will collide at hash address *10* with supplier S100. The hash function as it stands is thus clearly inadequate—it needs to be extended somehow to deal with the collision problem. One way to do this is to treat the remainder after division by 13, not as the hash address as such, but rather as the start point for a sequential search to find that address. Thus, to insert supplier S139 (assuming that suppliers S100-S500 already exist), we go to hash address *10* and search forward from that position for the first free location. The new supplier is stored at hash address *12*. To access that supplier subsequently, we go through a similar procedure.

2.6 Data Compression

There's one last storage issue I'd like to address briefly. We've seen that one problem with auxiliary structures such as indexes is that they increase storage space requirements. By contrast, **data compression** techniques are ways of *reducing* storage space requirements. In fact, such techniques can also, and perhaps more significantly, reduce the amount of I/O—for if the data occupies less storage, then fewer I/O operations will be needed to access it. (On the other hand, extra processing cycles might be needed to decompress the data after it has been retrieved, and that extra processing could have the effect of slowing down queries and updates, thereby negating the benefits of reducing I/O in the first place.)

In general, compression techniques are designed to exploit the fact that data values are almost never completely random but instead display some degree of predictability. As a trivial example, if a given person's name in a name-and-address file starts with the letter "R", then it's extremely likely that the next person's name will start with the letter "R" as well (assuming, of course, that the file is kept in alphabetical order by name).

One simple compression technique is thus to replace each individual data value by some representation of the difference between it and the value that immediately precedes it: **differential** compression. Note immediately, however, that this technique requires that the data be accessed sequentially, because to decompress any given stored value requires knowledge of the immediately preceding stored value. Differential compression thus has its main applicability in situations in which the data must be accessed sequentially anyway, as in the case of (for example) the entries in an index. Note in particular that—in the case of a clustering index specifically—the pointers can be compressed as well as the data; this is because, if the logical ordering imposed by the index is the same as (or close to) the physical ordering of the underlying stored file, then successive pointer values in the index will be quite similar to one another, and pointer compression is likely to be beneficial. In fact, indexes almost always stand to gain from the use of compression, at least for the data if not for the pointers.

To illustrate differential compression, let's forget suppliers for a moment and consider an index on, say, employee names. Suppose, realistically enough, that the index entries are grouped into blocks or pages, and suppose the first four entries on some particular page are for the following employees:

```
Roberton
Robertson
Robertstone
Robinson
```

Suppose also that the employee name field in the indexed file is 12 bytes wide, so that each of these names must be considered, in its uncompressed form, to be padded at the right with an appropriate number of blanks. Then one way to apply differential compression to this set of values is to replace those characters at the front of each entry that are the same as those in the previous entry on the same page by a corresponding count: **front** compression. Here's the result:

```
0 - Roberton++++
6 - son+++
7 - tone+
3 - inson++++
```

(trailing blanks now shown explicitly as "+"). If the counts occupy one byte each, the space requirements for these four values have been reduced from 48 to 36 bytes—a 25 percent reduction.

Another possible compression technique for this set of data is simply to eliminate the trailing blanks entirely (again, replacing them by an appropriate count): **rear** compression. Result:

```
0 - 8 - Roberton
6 - 3 - son
7 - 4 - tone
3 - 5 - inson
```

The first of the two counts in each entry here is as in the previous example, the second is a count of the number of characters recorded (not a count of the number of omitted blanks, observe). The space requirement is now just 28 bytes, a reduction of nearly 42 percent compared to the original.

Further rear compression can be achieved by dropping all characters to the right of the one required to distinguish the entry from its immediate neighbor(s), thus:

```
0 - 7 - Roberto
6 - 2 - so
7 - 1 - t
3 - 1 - i
```

(I'm assuming the first four characters of the next entry when decompressed aren't "Robi"). The space requirement is now just 19 bytes, a reduction of over 60 percent. Note, however, that now we've actually lost some information—that is, when decompressed, the four entries look like this:

```
Roberto?????
Robertso????
Robertst????
Robi????????
```

(where "?" represents an unknown character). Such loss of information is obviously permissible only if the data is recorded in full *somewhere* (in the example, it's recorded in the corresponding indexed file).

2.7 Concluding Remarks

So much for our quick survey of the direct-image approach to implementation, and in particular of some of the problems it seems to drag along in its wake. Surely we can do better than this. Indeed, relational advocates have claimed for years that the relational model doesn't have to be implemented this way; that's the whole point—or, at least, a large part of the point—of that clean "model vs. implementation" distinction I was emphasizing so much in Chapter 1. In fact, Codd himself, in his famous 1970 paper on the relational model [6], had this to say:

> [Although] implementation problems are [not] discussed ... the material presented should be adequate for experienced systems programmers to visualize several approaches.
>
> *—E. F. Codd*

The sad fact is, however, that the mainstream SQL vendors, at least, didn't do a very good job of that visualization. As a result, the idea that the model and implementation levels are supposed to be distinct, although a fundamental feature of relational products in theory, has very largely been overlooked in practice.

Incidentally, please note that I don't mean to imply by the foregoing remarks that *nobody* "did a good job of that visualization." Certain early prototypes and research proposals did depart—somewhat, though nothing like as far as the TR model does—from the direct-image style I've been describing (but the prototypes and proposals in question unfortunately had little or no commercial impact). I also don't mean to imply that I think all of the mainstream SQL products are implemented in exactly the same way; clearly, each of those products has its own internal architecture, unique to that product alone. But I think it's fair to say that, at least to a first approximation, the products do all store the data in the same kind of direct-image way. As a consequence, they all suffer from the kinds of deficiencies this chapter has been concerned with.

Data Independence

I'd like to close with a few more remarks on this business of clearly distinguishing between the model and its implementation. As you've probably realized, what we're really talking about here is the concept usually known as **data independence:** *physical* data independence, to be precise.[10] The basic idea behind physical data independence is that we should be able to make changes (for performance reasons in particular) to the way the data is physically stored and accessed, without having to make any corresponding changes in what that data looks like to the user, and hence without having to make any corresponding changes in the source code of applications that use that data. And while it's true that data independence isn't an absolute—different systems provide it to different degrees, and few if any provide none at all—it's also true that systems that adopt a direct-image style of implementation provide far less data independence than relational systems are theoretically capable of. We need to fix this problem.

Endnotes

1. Throughout this book, I choose to overlook the fact that "#" is not included in the SQL standard character set and thus cannot be used in what the standard calls a "regular identifier" [39].
2. It might be thought of as an *abstraction* of a table, with tuples instead of rows and attributes instead of columns.
3. Strictly speaking, this sentence is incorrect, because "tuples" in SQL have a left-to-right ordering to their components and so aren't true tuples, and the result of the SQL query is thus not a true relation. In what follows, I'll usually ignore this point.
4. As Keith Devlin says in his book *The Math Gene* [46]: "The human mind is a pattern recognizer ... The ability to see patterns and similarities is one of [its] greatest strengths" (page 60).
5. I pointed out in Chapter 1 that this assumption isn't always valid (the correspondence between relations and stored files isn't always one-to-one), but this fact doesn't materially affect the arguments of the present chapter. I make the assumption merely for reasons of simplicity and definiteness.
6. Actually, bitmap indexing is not much like conventional indexing at the detail level. A detailed explanation would be out of place here; suffice it to say that bitmap indexing is nonetheless still indexing, albeit of a somewhat novel kind, and it suffers from the same kinds of problems as conventional indexing does. If you're interested and want to learn more about it, you can find informal explanations in references [49] and [50].
7. Pointer chains were used very heavily in preSQL systems, especially in CODASYL systems such as IDMS [14,25]; few of today's SQL systems use them. I should add, however, that those early systems didn't just use pointer chains, they typically exposed them to the user (with significant negative consequences), precisely because—as noted in Chapter 1—those systems failed to make a clear distinction between model and implementation. I'll have more to say on this topic in Chapter 5 (Section 5.2).
8. This is probably why pointer chains aren't much used in today's SQL systems.
9. Of course, we can always impose any logical sequence we want on a given hash structure by means of a suitable index—indeed, we could impose several such sequences, by means of several indexes. Or we could use pointer chains. But these possibilities all introduce further complications, as we already know.
10. For a discussion of the distinction between physical and logical data independence, see reference [32]. Throughout the remainder of this book, I'll take the unqualified term *data independence* to mean physical data independence specifically.

3 Three Levels of Abstraction

3.1 Introduction

In order to understand the TR approach to implementing the relational model, it's necessary to be very clear over three distinct levels of the system, which I'll refer to as *the three levels of abstraction* (since each level is an abstraction of the one below, loosely speaking). The three levels, or layers, are:

1. The relational (or user) level
2. The file level
3. The TR level

They're illustrated in Fig. 3.1. In a nutshell:

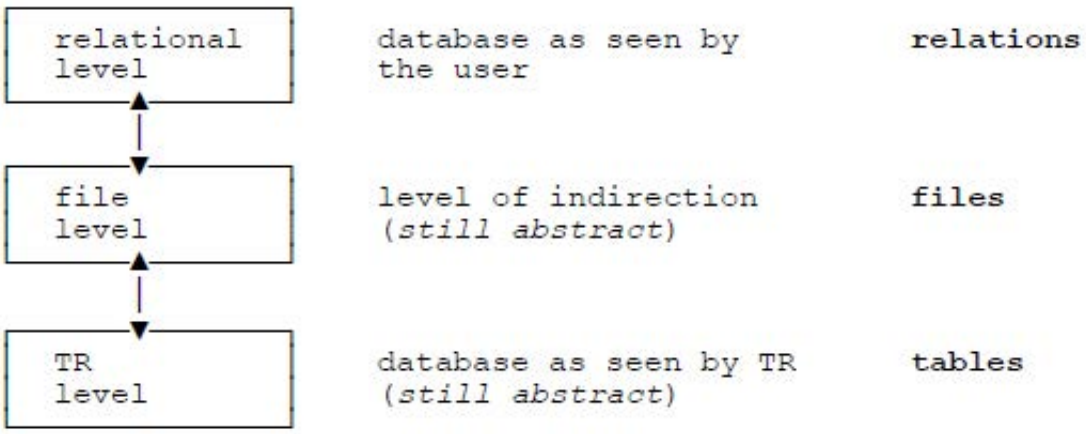


Fig. 3.1: The three levels of abstraction

- Level 1, which corresponds to the database as seen by the user, is the relational level. At this level, the data is perceived as **relations,** including, perhaps, the suppliers relation S discussed in Section 2.1 (and illustrated in Fig. 2.1) in the previous chapter.

- Level 3 is the fundamental TR implementation level. At this level, data is represented by means of a variety of internal structures called **tables**. *Please note immediately that those TR tables are NOT tables in the SQL sense and do NOT correspond directly to relations at the user level.*

- Level 2 is a level of indirection between the other two. Relations at the user or relational level are mapped to **files** at this level, and those files are then mapped to tables at the TR level. Of course, the mappings go both ways; that is, tables at the TR level map to files at the next level up, and those files then map to relations at the top level. *Note:* As I'm sure you know, *map* is a synonym for *transform* (and I'll be using the term in that sense throughout this book); thus, we're already beginning to touch on the TR transforms that were mentioned in Chapter 1. However, there's a great deal more to it, as we'll soon see.

Please now observe that each level has its own terminology: relational terms at the user level, file terms at the file level, and table terms at the TR level. Using different terms should, I hope, help you keep the three levels distinct and separate in your mind; for that reason, I plan to use the three sets of terms consistently and systematically throughout the rest of this book.

Having said that, I now need to say too that I'm well aware that some readers might object to my choice of terms—perhaps even find them confusing—for at least the following two reasons:

- First, the industry typically uses the terminology of tables, not relations, at the user level—almost exclusively so, in fact. But I've already explained some of my rationale for wanting to use relational terms at that level (see the previous chapter, Section 2.1), and I'm going to give some additional reasons in the next section.

- Second, the industry also typically tends to think of files as a fairly "physical" construct. In fact, I did the same thing myself in the previous chapter, somewhat, though I was careful in that chapter always to be quite clear that the files I was talking about were indeed *physically stored* files specifically. By contrast, the files I'll be talking about in the rest of the book are *not* physically stored; instead, they're an abstraction of what's physically stored, and hence a "logical" construct, not a physical one. (Though it wouldn't be wrong to think of them as "slightly more physical" than the user-level relations, if you like.)

If you still think my terms are confusing, then I'm sorry, but for better or worse they're the terms I'm going to use.

One final point: When I talk of three levels, or layers, of abstraction, I don't mean that each of those levels is physically materialized in any concrete sense—of course not. The relational level is only a way of looking at the file level, a way in which certain details are ignored (that's what "level of abstraction" *means*). Likewise, the file level in turn is only a way of looking at the TR level. Come to that, the TR level in turn is only a way of looking at the bits and bytes that are physically stored; that is, the TR level is itself—as already noted in Chapter 1, Section 1.2—still somewhat abstract. In a sense, the bits-and-bytes level is the *only* level that's physically materialized.[1]

3.2 The Relational Level

Since the focus of this book is on the use of TR technology to implement the relational model specifically, the topmost (user) level is relational by definition. In other words, the user sees the database as a set of **relations,** made up of **attributes** and **tuples** as explained in Chapter 2. For simplicity, I'm going to assume those relations are all **base** relations specifically (again, see Chapter 2); that is, I'll simply assume, barring explicit statements to the contrary, that any relation that's named and is included in the database is in fact a base relation specifically, and I won't usually bother to use the "base" qualifier.

Also, of course, the user at the relational level has available a set of **relational operators**—restrict, project, join, and so forth—for querying the relations in the database, as well as the usual INSERT, DELETE, and UPDATE operators for updating them. *Note:* If I wanted to be more precise here, I'd have to get into the important distinction between relation **values** and relation **variables**. Relational operators like join operate on relation *values,* while update operators like INSERT operate on relation *variables*. Informally, however, it's usual to call them all just relations, and—somewhat against my better judgment—I've decided to follow that common usage (for the most part) in the present book. For further discussion of such matters, see either reference [32] or reference [40].

Now, given the current state of the IT industry, the user level in a real database system will almost certainly be based on SQL, not on the relational model. As a consequence, users will typically tend to think, not in terms of relational concepts as such, but rather in terms of SQL analogs of those concepts. For example, there isn't any explicit project operator, as such, in SQL; instead, such an operation has to be formulated in terms of SQL's SELECT and FROM operators, and the user has to think in terms of those SQL operators, as in this example ("Project suppliers over supplier number and city name"):

```
SELECT S.S#, S.CITY
FROM S ;
```

Precisely because most of today's database systems are in fact SQL systems specifically, I'll show most of my examples in what follows in SQL, not in pure relational form. But I do still want to use the terms *relation, tuple,* and *attribute* at the user level (sometimes *user* relations, tuples, and attributes, for emphasis), instead of the more familiar SQL terms *table, row,* and *column,* and—as I promised I would, both in Chapter 2 and in the previous section—I'd like to give my reasons for adopting this perhaps rather purist or academic position. In essence, it seems to me that to use the SQL terms would lead to at least three problems:

- First of all, we're going to need to use the terminology of tables, rows, and columns—as we very often do when discussing software internals—at the implementation level (which is to say the TR level), and the TR and SQL constructs are, as already noted, completely different things. So there would be an obvious potential for confusion right away.

- Second, the SQL terminology tends to obscure the crucial distinction alluded to above between *relation values* and *relation variables*. (SQL doesn't clearly distinguish between these concepts at all, referring to them both simply as *tables,* a state of affairs that has demonstrably led to some confusion in the past.)

- Third, considered as possible user-level terms, *table, row,* and *column* are in fact actively misleading (indeed, I wish we'd never used them, not even in SQL), for at least the following reasons:

  - They lend weight to the "duplicate tuples" heresy. In fact, an SQL table can have duplicate rows, although as we know a relation can't have duplicate tuples. *Note*: The TR model itself doesn't care whether there are duplicates or not, and hence can support SQL's nonrelational tables as well as proper relations—but I don't propose to discuss that nonrelational support in any detail in this book.

  - They suggest there's a top-to-bottom ordering to the rows, though in fact there isn't.

  - They suggest there's a left-to-right ordering to the columns. (In fact there is, in SQL—another departure from the relational model, as noted in Chapter 1.)

  - They suggest that "row-and-column intersections" in those tables can be accessed via [*i,j*]-style subscripting, instead of associatively. That is, tables—but definitely not relations—are often thought of as being something like *arrays* (two-dimensional arrays, to be precise). *Note:* The term "associatively" refers to the fact that data at the relational level is accessed by value, not by address. For example, "Get tuples for suppliers in London" is a relational request, but "Get the first and fourth supplier tuples" isn't. Likewise, "Get status values from tuples for suppliers in Paris" is a relational request, but "Get values of the third attribute from tuples for suppliers in Paris" isn't.

  - Most significantly, they tend to obscure the important connections between the relational model and mathematics and logic. (Those connections are important because they're what make it possible to treat database management as a science; without them, the field becomes a mere ragbag of ad hoc tricks, techniques, and rules of thumb.)

This isn't an exhaustive list.[2]

3.3 The File Level

The first step, conceptually speaking, in mapping a given relation to an appropriate TR representation is to convert that relation into a **file,** with **records** corresponding to the tuples and **fields** corresponding to the attributes. For example, Fig. 3.2 shows a possible file corresponding—in a trivially obvious way—to the suppliers relation of Fig. 2.1 in Chapter 2.

field sequence: *1* *2* *3* *4*

record sequence:	S#	SNAME	STATUS	CITY
1	S1	Smith	20	London
2	S2	Jones	10	Paris
3	S3	Blake	30	Paris
4	S4	Clark	20	London
5	S5	Adams	30	Athens

Fig. 3.2: A file corresponding to the suppliers relation of Fig. 2.1

Within such a file, records do have a top-to-bottom ordering and fields do have a left-to-right ordering, as the record numbers and field numbers in the figure are meant to suggest.[3] However, the orderings in question are essentially arbitrary; thus, for example, the suppliers relation of Fig. 2.1 could map equally well to any of 2,880 different files (120 different orderings for the five records and 24 different orderings for the four fields). By way of illustration, Fig. 3.3 shows another possible file corresponding to the suppliers relation of Fig. 2.1.

field sequence: 1 2 3 4

record sequence:	STATUS	S#	CITY	SNAME
1	20	S4	London	Clark
2	30	S5	Athens	Adams
3	10	S2	Paris	Jones
4	20	S1	London	Smith
5	30	S3	Paris	Blake

Fig. 3.3: Another file corresponding to the suppliers relation of Fig. 2.1

Of course, those 2,880 different files are all equivalent to one another, in the sense that they all represent exactly the same information; in other words, they're all **information-equivalent**. It's sometimes convenient, therefore, to regard them, not so much as 2,880 distinct files as such, but rather as 2,880 different versions of "the same" file.[4] This perception will turn out to be important in the next chapter—also, especially, in Chapter 7.

Files, records, and fields (sometimes *user* files, records, and fields for emphasis, since in many respects the file level is still quite close to the user or relational level) can be operated upon by obvious counterparts to the operators available at the relational level. Also, reconstructing the corresponding relation from a given file (any version) is trivial: Just ignore the orderings.

Files such as those shown in Figs. 3.2 and 3.3 can now be represented by tables at the TR level and can be reconstructed from those TR tables. In fact (important!), many different versions of the same file can all be reconstructed from the same TR tables equally easily (using the term "versions" in the special sense explained above—that is, record and field orderings might be different, but content remains the same). We'll see how this works out in the next chapter.

One last point: I've called this level the *file* level and the next more specifically the *TR* level because most of the ingenuity, inventiveness, and novelty of the TR model is to be found at that next level. However, the file level too is part of the overall TR implementation approach, of course.

3.4 The TR Level

Files at the file level map to **tables** at the TR level, and those tables are made up of **rows** and **columns**. Like records and fields within files, rows in a TR table do have a top-to-bottom ordering and columns in such a table do have a left-to-right ordering. And, very importantly, a row-and-column intersection within such a table, which I'll refer to as a **cell,** can be addressed via [*i,j*]-style subscripting (where *i* is the row number and *j* is the column number); in other words, TR tables, unlike SQL tables, can legitimately, and usefully, be thought of as *two-dimensional arrays*. Cells in such a table or array contain **values**. What's more, those values can sometimes be composite; for example, a given cell might contain an ordered pair of pointer values, and an ordered pair of values can certainly be regarded as a value—a composite value—in its own right.

Now, the mapping of files to TR tables is quite a complex business, and I don't want to start getting into details of how it's done until the next chapter. Suffice it to say that it's nothing like the direct-image kind of mapping discussed in Chapters 1 and 2. In particular, rows in TR tables do *not* correspond in any one-to-one kind of way to records at the file level, nor a fortiori do they correspond in any one-to-one kind of way to tuples at the relational level. By way of illustration, Fig. 3.4 shows a TR table, the **Field Values Table,** corresponding to the file of Fig. 3.2. As I've already indicated, I don't want to get into details yet of just how that table is obtained from that file, but you might like to try to figure it out for yourself (it's not very difficult). All I want to do now is draw your attention to the fact that indeed, as claimed, the rows don't correspond in any obvious way to the records shown in Fig. 3.2.

column sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
row sequence:	1	S1	Adams	10	Athens
	2	S2	Blake	20	London
	3	S3	Clark	20	London
	4	S4	Jones	30	Paris
	5	S5	Smith	30	Paris

Fig. 3.4: Field Values Table corresponding to the file of Fig. 3.2

In order to be able to reconstruct the file of Fig. 3.2 from the Field Values Table of Fig. 3.4, we need another table, the **Record Reconstruction Table**.[5] Again, I don't want to get into details yet of how the Record Reconstruction Table is obtained, nor how it's used in the reconstruction process; I'll just show, in Fig. 3.5, a possible Record Reconstruction Table corresponding to the file shown in Fig. 3.2 (and to the Field Values Table shown in Fig. 3.4)—and point out that the entries in the Record Reconstruction Table aren't supplier numbers or status values, etc., any longer (despite the column labels) but are **row numbers** instead. For further explanation, see the next chapter.

column sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
row sequence:	1	5	5	4	5
	2	4	4	2	1
	3	2	3	3	4
	4	3	1	5	2
	5	1	2	1	3

Fig. 3.5: Record Reconstruction Table corresponding to file of Fig. 3.2 (and Field Values Table of Fig. 3.4)

By the way, it's a little misleading to talk (as I've just been doing) in terms of *the* Field Values Table and *the* Record Reconstruction Table, because there'll probably be many such tables in any real implementation—one of each for each file at the file level, loosely speaking (but see Chapters 9 and 11-14 later). However, it's much easier to talk in terms of, for example, "*the* Field Values Table" instead of having to say something like "the particular Field Values Table that corresponds to the file of Fig. 3.2" every time we need to refer to such a thing. So I'll continue to talk this way for most of the rest of this book, and hope you won't find the practice confusing.

I'll be discussing what's involved in building and using the Field Values Table and the Record Reconstruction Table in the next few chapters. For now, let me close by stressing a point I've made a couple of times already: namely, that the TR level, though obviously at a much lower level of detail than the relational level, is nevertheless still abstract. In fact, TR is a *model* in the sense of Chapter 1, meaning it can be regarded as a layer of abstraction over something deeper down. In particular, the TR tables discussed above, and their associated operators, can be physically implemented in a variety of different ways, some of which I'll be talking about in later chapters. Very importantly, of course, they can be implemented in either main memory or secondary storage; indeed, they can be implemented on absolutely any hardware platform whatsoever, from a handheld or palmtop computer, to a laptop or desktop machine, to a mainframe, to a client/server or other distributed system, to the most massively parallel supercomputer. Now, this book is primarily concerned with the TR model as such, not so much with specific implementations of that model; however, Part III does specifically address the question of a disk-based implementation, since there are clearly special issues to be addressed in such an environment. By contrast, Part II doesn't assume any particular implementation environment at all (at least, not explicitly); however, you can think of it for the most part as implicitly assuming a main-memory environment, if you find it helpful to do so.

Endnotes

1. Well ... to be pedantic about it, those bits and bytes are an abstraction too, of course, and so on, all the way down to the level of electrons (and beyond!). But bits and bytes are physical enough for our purposes.
2. In particular, I'd like to point out that certain very important relational "tables"—namely, the ones that references [12] and [24] call TABLE_DEE and TABLE_DUM—don't have any "columns" anyway. (The analogy between relations and tables breaks down here.) Further discussion of this particular issue would take us much too far afield, however; if you're intrigued and want to know more, see either reference [32] or reference [40].
3. In practice, like the record numbers discussed in Chapter 2, those record and field numbers probably won't be simple sequential numbers as shown in the figure. The same is true for row and column numbers at the TR level (see the next section).
4. Incidentally, note that those different versions can't all be obtained by means of a simple ORDER BY.
5. I could logically have called this table the *File* Reconstruction Table, but I wanted to emphasize the point that it can be used to reconstruct individual records of the file as well as the file in its entirety. In fact, it can be used to reconstruct any subset of the records in that file, and any subset of the fields in those records, as we'll see in the course of the next few chapters.

Part II: The Transrelational Model

4 Core Concepts

4.1 Introduction

Now (at last) I can begin to explain the TR model in detail. As I mentioned several times in Part I, TR is indeed still a model, and thus, like the relational model, still somewhat abstract. At the same time, however, it's at a much lower level of abstraction than the relational model; it can be thought of as being closer to the physical implementation level ("closer to the metal"), and accordingly more oriented toward issues of performance. In particular, it relies heavily on the use of **pointers**—a concept deliberately excluded from the relational model, of course, for reasons discussed in references [9], [30], [40], and many other places—and its operators are much more procedural in nature than those of the relational model. (What I mean by this latter remark is that code that makes use of those operators is much more procedural than relational code is, or is supposed to be.) What's more, reference [63] includes detailed, albeit still somewhat abstract, algorithms for implementing those operators. *Note:* These remarks aren't meant to be taken as criticisms, of course; I'm just trying to capture the essence of the TR model by highlighting some of its key features.

Despite its comparatively low-level nature, the fact remains that, to say it again, TR is indeed a model, and thus capable of many different physical realizations. In what follows, I'll talk for much of the time in terms of just one possible realization—it's easier on the reader to be concrete and definite—but I'll also mention some alternative implementation schemes on occasion. Note that the alternatives in question have to do with the implementation of both data structures and corresponding access algorithms. In particular, bear in mind that both main-memory and secondary-storage implementations are possible.

Now, this book is meant to be a tutorial; accordingly, I want to focus on showing the TR model in action (as it were)—that is, showing how it works in terms of concrete examples—rather than on describing the abstract model as such. Also, many TR features are optional, in the sense that they might or might not be present in any given implementation or application of the model, and it's certainly not worth getting into all of those optional features in a book of this kind. Nor for the most part is it worth getting into the optionality or otherwise of those features that are discussed—though I should perhaps at least point out that options do imply a need for decisions: Given some particular option *X,* some agency, at some time, has to decide whether or not *X* should be exercised. For obvious reasons, I don't want to get into a lot of detail on this issue here, either. Suffice it to say that I don't think many of those decisions, if any at all, should have to be made at database design time (by some human being) or at run time (by the system itself); in fact, I would expect most of them to be made during the process of designing the DBMS that is the specific TR implementation in question. In other words, I don't think the fact that those decisions do have to be made implies that a TR implementation will therefore suffer from the same kinds of problems that arise in connection with direct-image systems, as discussed in Chapter 2.

It follows from all of the above that this book is meant as an introduction only; many topics are omitted and others are simplified, and I make no claims of completeness of any kind.

Now let's get down to business. In this chapter and the next,[1] we'll be looking at what are clearly the most basic TR constructs of all: namely, the Field Values Table and the Record Reconstruction Table, both of which were mentioned briefly in the final section of the previous chapter. These two constructs are absolutely fundamental—everything else builds on them, and I recommend as strongly as I can that you familiarize yourself with their names and basic purpose before you read much further. Just to remind you:

- The *Field Values Table* contains the field values from a given file, rearranged in a way to be explained in Section 4.3.
- The *Record Reconstruction Table* contains information that allows records of the given file to be reconstructed from the Field Values Table, in a way to be explained in Section 4.4.

In subsequent chapters I'll consider various possible refinements of those core concepts. *Note:* Those refinements might be regarded in some respects as "optional extras" or "frills," but some of them are very important—so much so, that they'll almost certainly be included in any concrete realization of the TR model, as we'll see.

4.2 The Crucial Idea

Let *r* be some given record within some given file at the file level. Then the crucial insight underlying the TR model can be characterized as follows:

> **The stored form of *r* involves two logically distinct pieces, a set of field values and a set of "linkage" information that ties those field values together, and there's a wide range of possibilities for physically storing each piece.**

In direct-image systems, the two pieces (the field values and the linkage information) are kept together, of course; in other words, the linkage information in such systems is represented by *physical contiguity*. In TR, by contrast, *the two pieces are kept separate;* to be specific, the field values are kept in the Field Values Table, and the linkage information is kept in the Record Reconstruction Table. That separation makes TR strikingly different from virtually all previous approaches to implementing the relational model (see Chapters 1 and 2), and is the fundamental source of the numerous benefits that TR technology is capable of providing. In particular, it means that TR data representations are categorically not a direct image of what the user sees at the relational level.

Note: One immediate advantage of the separation is that the Field Values Table and the Record Reconstruction Table can both be physically stored in a way that is highly efficient in terms of storage space and access time requirements. However, we'll see many additional advantages as well, both in this chapter and in subsequent ones.

4.3 The Field Values Table

Consider the file shown in Fig. 4.1. The figure is basically a repeat of Fig. 3.2, except that for the sake of the example I've rearranged the records into a different top-to-bottom sequence (after all, we know from Chapter 3 that record sequence at the file level is effectively arbitrary anyway; in fact, the same is true of left-to-right field sequence as well, but for simplicity I've kept that unchanged). Fig. 4.2, a repeat of Fig. 3.4, shows the corresponding **Field Values Table**.

field sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
record sequence:	1	S4	Clark	20	London
	2	S5	Adams	30	Athens
	3	S2	Jones	10	Paris
	4	S1	Smith	20	London
	5	S3	Blake	30	Paris

Fig. 4.1: A file corresponding to the suppliers relation of Fig. 2.1

column sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
row sequence:	1	S1	Adams	10	Athens
	2	S2	Blake	20	London
	3	S3	Clark	20	London
	4	S4	Jones	30	Paris
	5	S5	Smith	30	Paris

Fig. 4.2: Field Values Table corresponding to the file of Fig. 4.1

Note: Together with Fig. 2.1, which shows the original suppliers relation, Figs. 4.1 and 4.2 form the basis for a running example that I'll be using throughout this chapter (and indeed throughout the next two chapters as well). You might want to keep a copy of those figures by you for ease of subsequent reference.

Now, you've probably figured out for yourself how the Field Values Table is obtained from the corresponding file: Basically, each column of the table contains the values from the corresponding field of the file, **rearranged into ascending sort order**. Note immediately, therefore, that no matter what order the records of the file appear in initially, we wind up with the same Field Values Table; that's why Figs. 4.2 and 3.4 are identical, even though Figs. 4.1 and 3.2 are not. In other words, record ordering is irrelevant so far as the Field Values Table is concerned. (By contrast, field ordering is not irrelevant; that is, the left-to-right column ordering of the Field Values Table is the same as the left-to-right field ordering in the corresponding file. However, this point isn't very important so far as the user is concerned.)

Incidentally, it should be immediately clear from the example that one way to think about TR is that it's a technology that stores the data "attribute-wise" rather than "tuple-wise"—though I hasten to add that this informal characterization doesn't even begin to capture all of the implications and advantages of the TR approach. Now, although by contrast most mainstream SQL products store the data "tuple-wise" (as we saw in Chapter 2), there have been a few systems, both prototypes and commercial products, that have stored the data "attribute-wise" instead (see, for example, references [2], [49], [52], [65], and [66]); indeed, some of those products are still available in the marketplace at the time of writing. But none of those systems carried (or carry) the "attribute-wise" idea to anything like the same lengths that TR does. *Note:* By the same token, some of those systems used or use various kinds of data compression on the attributes, too, but again not nearly to the same extent that TR does (see Chapters 8 and 9).

It should also be clear from the example that TR takes the concept of *data independence* much further than previous systems have done. To be specific, there's essentially no concept of a user-level tuple at all at the TR level, whereas (again as we saw in Chapter 2) conventional systems typically do store direct images of user-level tuples, albeit in a variety of different ways. (Even those systems that store data "attribute-wise" still retain fairly close ties between the user level and the physical storage level—for example, by ensuring that the attribute values from a given user-level tuple all appear at the same relative position within the individual attribute representations.)

Anyway, let's get back to the Field Values Table. I'm clearly not in a position yet to describe exactly how that table is used, nor to explain its advantages (I need to discuss the Record Reconstruction Table first); nevertheless, I'd still like to mention a few points that I think should at least make some intuitive sense, even before we start to look at the Record Reconstruction Table as such.

- First of all, the fact that each column of the Field Values Table is in sorted order is clearly going to help with user-level ORDER BY requests. For example, a request to see suppliers in city name sequence shouldn't require a run-time sort, nor an index.

- The same is true of a request to see suppliers in *reverse* city name sequence (meaning descending sort order, instead of ascending)—the implementation can simply process the Field Values Table bottom to top instead of top to bottom.

- Analogous remarks apply to every single attribute; that is, the Field Values Table effectively represents several different sort orders simultaneously (in effect, a sort order in both directions on every individual attribute).

- Requests involving specific value lookups—for example, a request to see suppliers in London—can be implemented by means of a binary search. And, again, analogous remarks apply to every attribute. *Note:* Binary search is also known as *logarithmic* search, on account of the fact that it's an O(log *N*) algorithm, where *N* is the number of items in the list to be searched and O(log *N*) means the execution time is proportional to log *N* (O here stands for "order of magnitude"). Sequential search, by contrast, is an O(*N*) algorithm. For example, if *N* = 1,000,000, then we might say, loosely, that binary search is some 50,000 times more efficient than sequential search.

These points will all be expanded and made clearer in Sections 4.4, 4.5, and 4.6 below. For now, here are a couple of final remarks to close out this section:

- In some respects, the Field Values Table can be thought of as a kind of bridge between the user perception of the data (meaning the original user-level relation and/or the corresponding file) and other internal TR structures. Note in particular that **the Field Values Table is the only TR table that contains user data as such**—all of the others contain internal information, encoded in ways that make sense to TR but aren't directly relevant to, or exposed to, the user at all.

- As I explained in Chapter 2, at the end of Section 2.2, there's only one physical sequence available to us at the hardware level, so we want to make the best use of it we can. *In the TR approach, we store the Field Values Table in physical sequence by row number*. (It should be clear from what I said a few paragraphs back—regarding, for example, ORDER BY requests—that we often need to process the Field Values Table sequentially by row number, so storing it as just indicated is clearly advantageous.) Of course, storing the Field Values Table in physical sequence in this manner doesn't preclude us from exploiting physical sequence appropriately for other internal structures as well, but it's vitally important that we do so in the case of the Field Values Table in particular.

4.4 The Record Reconstruction Table

Fig. 4.3 shows the Field Values Table from Fig. 4.2 side by side with an appropriate **Record Reconstruction Table**. Note that the two tables both have the same number of rows and columns; indeed, there's a direct one-to-one correspondence between the cells of the two tables, as we'll see in a moment. (In fact, each table has the same number of rows and columns as the file in Fig. 4.1 has records and fields, respectively.) Note too that the entries in the Record Reconstruction Table cells aren't supplier numbers or supplier names (etc.) any longer; instead, they're **row numbers,** and those row numbers can be thought of as **pointers** to the rows of either or both of the Field Values Table and the Record Reconstruction Table, depending on the context in which they're used. (For this reason, the columns in the Record Reconstruction Table really ought not to be labeled S#, SNAME, etc., as I've shown them in the figure; however, I think those labels help to make certain later explanations easier to follow.) *Note:* You might want to keep a copy of the Record Reconstruction Table from Fig. 4.3 by you as well for purposes of subsequent reference.

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	S1	Adams	10	Athens
2	S2	Blake	20	London
3	S3	Clark	20	London
4	S4	Jones	30	Paris
5	S5	Smith	30	Paris

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	5	4	4	5
2	4	5	2	4
3	2	2	3	1
4	3	1	1	2
5	1	3	5	3

Fig. 4.3: Field Values Table of Fig. 4.2 and a corresponding Record Reconstruction Table

Now, I deliberately don't want to get into details just yet as to how the Record Reconstruction Table is built in the first place; instead, I want to show how it's used. To that end, please consider the following sequence of operations. (Recall from Chapter 3 that, in the subscript expression [*i,j*], *i* is a row number and *j* is a column number.)

Step 1: Go to cell [*1,1*] of the Field Values Table and fetch the value stored there—namely, the supplier number S1. That value is the **first** field value (that is, the S# field value) within a certain supplier record in the suppliers file.

Step 2: Go to the *same* cell (that is, cell [*1,1*]) of the Record Reconstruction Table and fetch the value stored there—namely, the row number *5*. That row number is interpreted to mean that the *next* field value (which is to say, the **second** or SNAME value) within the supplier record whose S# field value is S1 is to be found in the SNAME position of the **fifth** row of the Field Values Table—in other words, in cell [*5,2*] of the Field Values Table. Go to that cell and fetch the value stored there (supplier name Smith).

Step 3: Go to the corresponding Record Reconstruction Table cell [*5,2*] and fetch the row number stored there (*3*). The next (**third** or STATUS) field value within the supplier record we're reconstructing is in the STATUS position in the **third** row of the Field Values Table—in other words, in cell [*3,3*]. Go to that cell and fetch the value stored there (status 20).

Step 4: Go to the corresponding Record Reconstruction Table cell [*3,3*] and fetch the value stored there (which is *3* again). The next (**fourth** or CITY) field value within the supplier record we're reconstructing is in the CITY position in the **third** row of the Field Values Table—in other words, in cell [*3,4*]. Go to that cell and fetch the value stored there (city name London).

Step 5: Go to the corresponding Record Reconstruction Table cell [*3,4*] and fetch the value stored there (*1*). Now, the "next" field value within the supplier record we're reconstructing looks like it ought to be the *fifth* such value; however, supplier records have only four fields, so that "fifth" wraps around to become the *first*. Thus, the "next" (first or S#) field value within the supplier record we're reconstructing is in the S# position in the first row of the Field Values Table—in other words, in cell [*1,1*]. But that's where we came in, and the process stops.

As I hope you can see, the foregoing sequence of operations allows us to reconstruct one particular record from the suppliers file—to be specific, the one shown as record number *4* in Fig. 4.1:

	S#	SNAME	STATUS	CITY
4	S1	Smith	20	London

(I don't mean to suggest that the record number itself—*4,* in the example—is produced in the reconstruction process; I've shown it here merely to help you relate the output from that process back to the file as shown in Fig. 4.1.)

By the way, note how the row-number pointers we followed in the foregoing example form a *ring*—in fact, two isomorphic rings, one in the Field Values Table and one in the Record Reconstruction Table. See Fig. 4.4.

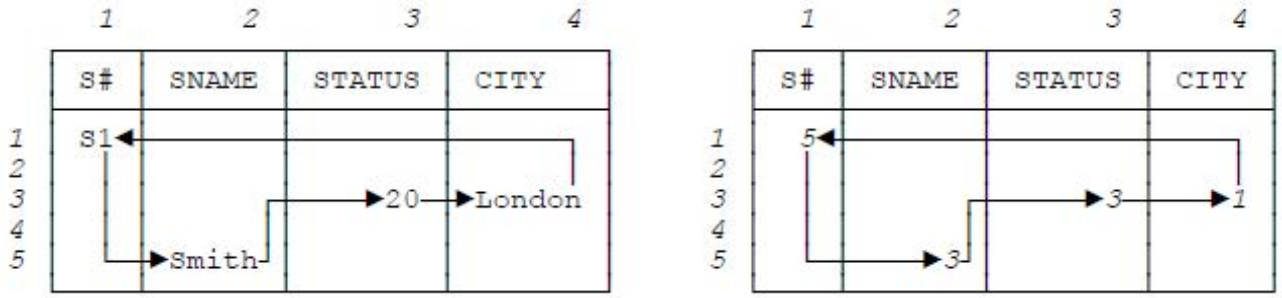


Fig. 4.4: Pointer rings (examples)

As an exercise—**Exercise 1**[2]—I strongly recommend you try reconstructing another supplier record for yourself. If you start with cell [*2,1*] in the Field Values Table, you should obtain record number *3* from Fig. 4.1:

	S#	SNAME	STATUS	CITY
3	S2	Jones	10	Paris

Similarly, starting with cell [*3,1*] gives record *5;* starting with cell [*4,1*] gives record *1;* and starting with cell [*5,1*] gives record *2*. **Observe the net effect:** If we process the entire Field Values Table in supplier number order by going top to bottom down the S# column—that is, if we carry out the record reconstruction process five times, starting respectively with cells [*1,1*], [*2,1*], [*3,1*], [*4,1*], and [*5,1*], in that order—then we reconstruct a version of the entire original suppliers file in which the records appear in ascending supplier number order. In other words, we've just implemented the following SQL query—

```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY
FROM S
ORDER BY S# ;
```

Likewise, to implement this SQL query—

```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY
FROM S
ORDER BY S# DESC ;
```

(where DESC means descending sequence)—all we have to do is process the supplier number column of the Field Values Table in reverse order and do the reconstructions starting from cell [*5,1*], then [*4,1*], and so on. What's more, we haven't had to do a run-time sort in either case, nor have we had to use an index.

Ordering by Other Attributes

Now consider this SQL query:

```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY
FROM S
ORDER BY STATUS ;
```

Precisely because (as noted earlier) the pointers in the Record Reconstruction Table form *rings,* we can enter those rings at any point. When we apply the reconstruction algorithm, therefore, we can start at any cell we like. In particular, if we start with cell [*1,3*]—that is, the first cell in the STATUS column—we obtain the record:

	S#	SNAME	STATUS	CITY
3	S2	Jones	10	Paris

(More precisely, we obtain a version of this record in which the left-to-right field ordering is STATUS, then CITY, then S#, then SNAME.) Following on down the STATUS column—that is, starting the reconstruction process successively with cells [*2,3*], [*3,3*], [*4,3*], and [*5,3*]—we'll eventually obtain the entire suppliers file in ascending status order.

In analogous fashion, if we process the Record Reconstruction Table in sequence by entries in the SNAME column, we obtain the suppliers file in ascending supplier name order; likewise, if we process it in sequence by entries in the CITY column, we obtain the file in ascending city name order. In other words, the Record Reconstruction Table and the corresponding Field Values Table together represent all of these orderings simultaneously—without (to repeat) any need for either indexes or run-time sorting. *This fact constitutes one of the major benefits of the TR approach.*

By the way, this is as good a point as any to mention that the reconstruction algorithm is known informally as **the zigzag algorithm** (and the individual pointer rings are known as **zigzags**), for obvious reasons.

And by the way again: Notice that, to be precise, we can't sensibly talk about the Record Reconstruction Table that corresponds to a given Field Values Table; rather, we have to talk in terms of the Record Reconstruction Table that corresponds to a given *file* (and therefore, in a sense, to the unique Field Values Table that corresponds to that file as well). The reason is that—obviously enough—several logically distinct files can all have the same Field Values Table, and such files will clearly need different Record Reconstruction Tables in order to support the corresponding reconstruction process properly. For example, this state of affairs would obtain if we had a suppliers file that was identical to the one shown in Fig. 4.1 except that supplier S1 was named Jones and supplier S2 was named Smith.

Equality Restrictions

Now let's take a look at an SQL query involving a simple *equality restriction:*

```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY
FROM S
WHERE S.CITY = 'London' ;
```

Since the CITY column (like every column) of the Field Values Table is kept in sorted order, a binary search—or simple variant thereof—can be used to find the cells containing London. Given the Field Values Table of Fig. 4.2, those cells turn out to be [*2,4*] and [*3,4*]. Zigzags can now be constructed by following the pointer rings running through cells [*2,4*] and [*3,4*] of the Record Reconstruction Table. In the example, those zigzags look like this:

```
[2,4], [4,1], [3,2], [2,3]
```

and

```
[3,4], [1,1], [5,2], [3,3]
```

Superimposing these zigzags on the Field Values Table, we obtain the field values for the desired records:

	S#	SNAME	STATUS	CITY
1	S4	Clark	20	London
4	S1	Smith	20	London

Other User-Level Operations

It should be clear that the Field Values Table and the Record Reconstruction Table together offer direct support for many other user-level operations too, in addition to simple ORDER BY and equality restriction operations. In fact, most if not all of the fundamental relational operations—restrict, project, join, summarize, and others (not to mention the operation of duplicate elimination, which is needed internally, even in true relational systems)—have implementation algorithms that rely on the ability to access the data in some specific sequence. By way of example, consider join. We saw in Chapter 2 that sort/merge is a good way to implement join. Well, TR lets us do a sort/merge join without having to do the sort!—or, at least, without having to do the *run-time* sort (the sort's done when the Field Values and Record Reconstruction Tables are built, which is to say at load time, loosely speaking). Suppose, for example, that the database involves a parts relation as well as the suppliers relation, and suppose both relations have a CITY attribute. In order to join suppliers and parts over city names, then, we simply have to access each of the two Field Values Tables in city name sequence and do a merge-style join.

One important implication of all of the above is that life becomes much easier for the system optimizer; to be more specific, the access path selection process (see Chapter 2) becomes much simpler—even completely unnecessary, in some cases. Another implication is that many of the auxiliary structures found in traditional DBMSs become unnecessary too (though it might be a good idea to use hashing on either the Field Values Table or the Record Reconstruction Table or both, if those tables get very large[3]). Yet another implication is that physical database design becomes much easier, involving as it does far fewer options and choices, and the same is true for performance tuning.

For further discussion of the use of TR structures in implementing the relational operators, see Chapter 10. Meanwhile, I'll close this section with a nice analogy that might help you understand and remember how the Field Values and Record Reconstruction Tables fit into the overall scheme of things:

- The Field Values Table is like a *parts list* that's used in some manufacturing process.
- The Record Reconstruction Table is like *instructions for assembling parts*—that is, instructions for using that parts list to manufacture finished products.

Incidentally, it should be clear from this analogy that the "assembly" process is bound to have some associated costs, especially in a disk-based environment, and we clearly want to keep those costs to a minimum. I'll address this issue in subsequent chapters.

4.5 Building the Record Reconstruction Table

I've now shown in outline what the Record Reconstruction Table looks like and how it's used, but I haven't shown how it's built in the first place. Now it's time to take a look at this latter question. Please note, however, that I'll be revisiting this topic at several points in later chapters (as well as in the final section of the present chapter); all I want to do for the moment is consider the simple case. Once again I'll base my discussions and explanations on the suppliers file shown in Fig. 4.1, together with the corresponding Field Values Table shown in Fig. 4.2 and repeated in Fig. 4.3.

Note first that the Record Reconstruction Table is built directly from the *file* (the Field Values Table plays no part in the process at all). We begin by considering the effect of applying various sort orderings to that file. For example, if we sort the file by ascending supplier number, we get the records in the sequence *4, 3, 5, 1, 2*. I'll call this sequence **the record permutation corresponding to the ordering "ascending S#"** (*the S# permutation* for short). Other permutations are as follows:

- Ascending SNAME: *2, 5, 1, 3, 4*
- Ascending STATUS: *3, 1, 4, 2, 5*
- Ascending CITY: *2, 1, 4, 3, 5*

We can summarize these permutations by means of the following **Permutation Table**:

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	4	2	3	2
2	3	5	1	1
3	5	1	4	4
4	1	3	2	3
5	2	4	5	5

Note: It follows from the way we built it that, in this table, cell [*i,j*] contains the record number within the suppliers file of the record that appears in the *i*th position when that file is sorted by ascending values of the *j*th field. (You might want to read that sentence again.) For example, cell [*3,2*] contains the value *1;* if the original file is sorted by ascending SNAME value—SNAME being the *second* field—the record that appears in the *third* position is indeed record number *1* (since that record contains the third lowest SNAME value, Clark).

Now, the foregoing Permutation Table is *not* the desired Record Reconstruction Table, but it could certainly be used to perform the function of that table (that is, it could be used to reconstruct records of the original file), as follows. Suppose we want to reconstruct the fourth record of that file. Noting that the value *4* appears in the first position in column *1,* the fifth position in column *2,* the third position in column *3,* and the third position again in column *4,* we can go to the Field Values Table and pick out the supplier number in cell [*1,1*], the supplier name in cell [*5,2*], the status value in cell [*3,3*], and the city name in cell [*3,4*], to obtain the record (once again)

	S#	SNAME	STATUS	CITY
4	S1	Smith	20	London

In other words, the sequence of Permutation Table cells

```
[1,1], [5,2], [3,3], [3,4]
```

indicates that record number *4* appears first in the S# permutation ("ORDER BY S#"), fifth in the SNAME permutation ("ORDER BY SNAME"), third in the STATUS permutation ("ORDER BY STATUS"), and third again in the CITY permutation ("ORDER BY CITY"). And if that sequence of Permutation Table cells seems familiar, then so it should—it's exactly the sequence of cells we passed through (albeit in the Field Values and Record Reconstruction Tables, not the Permutation Table) when we were reconstructing record number *4* in the previous section (Section 4.4).

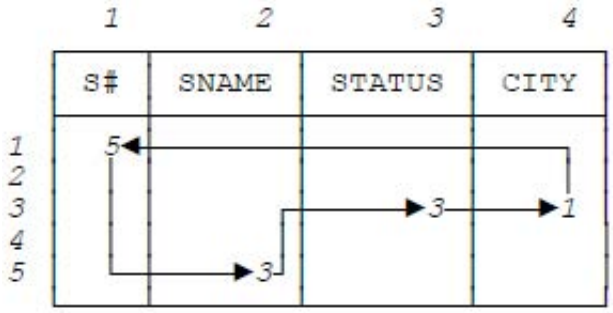
Now, the trouble with the foregoing algorithm—the algorithm, that is, for reconstructing records from the Permutation Table—is that the record numbers are effectively stored in each column of that table in random order. As a consequence, a sequential search is needed to find the desired record number (*4,* in the example) in each column. However, we can overcome this difficulty by using the Record Reconstruction Table in place of the Permutation Table. The Record Reconstruction Table differs from the Permutation Table in the following important respect:

> Where the Permutation Table has a sequence of cells (one cell per column) that each contain some particular *record number,* the Record Reconstruction Table has a sequence of cells (corresponding to the record with that record number) that each contain *a pointer to the next cell in that sequence.*

(As we know, the pointers in question are row numbers, and those row numbers identify both rows in the Record Reconstruction Table itself *and* rows in the corresponding Field Values Table.) Thus, for example, considering only record number *4,* the Permutation Table looks like this:

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	4			
2				
3			4	4
4				
5		4		

By contrast, the Record Reconstruction Table looks like this (I've shown the pointer ring or zigzag explicitly for the sake of the example)—



—as indeed we already know from the previous section. And of course it's much faster to follow a ring of pointers than to do a series of sequential searches.

Incidentally, note that the zigzag just shown in the Record Reconstruction Table—unlike its counterpart in the Permutation Table—includes no information as to which particular record in the suppliers file it corresponds to; all we know is that the cells linked together in that zigzag do all correspond to the *same* record in that file. But no information has really been lost, because the original record orderings (and hence record numberings) were arbitrary anyway. In other words, if there are *M* records altogether, we can in principle generate *M!* ("factorial *M*" = *M* * (*M*−1) * (*M*-2) * ... * 3 * 2 * 1) different versions of the original file from the same Record Reconstruction Table. Of course, those versions are all information-equivalent, as explained in Chapter 3.

(In contrast to the foregoing paragraph, the Record Reconstruction Table does still include information regarding the left-to-right *field* ordering of the corresponding file, inasmuch as its left-to-right column ordering is exactly that ordering. However, this fact, although it does have some bearing on certain internal operations, is irrelevant to the user at the relational level.)

Here then is the algorithm for building the Record Reconstruction Table from the Permutation Table:

> *Step 1:* Let *PT* be the Permutation Table. Build a table *RRT* with the same number of rows and columns as *PT* and with all cells empty.
>
> *Step 2:* For all records in the user file, do *Step 3*.
>
> *Step 3:* For all columns of *PT,* do *Step 4*.
>
> *Step 4:* Let the current record of the user file be the *r*th record, and let the current column of *PT* be the *j*th column. Let cell [*i,j*] of *PT* be that cell of column *j* that contains the record number *r*. At cell [*i,j*] of *RRT,* place the value *i′* where cell [*i′,j*+1] of *PT* is that cell of column *j*+1 that contains the record number *r*. If column *j* is the last column, take column *j*+1 as the first column.

After this algorithm has been executed, *RRT* is the desired Record Reconstruction Table.

As an exercise (**Exercise 2**), you might like to check that the foregoing algorithm, when applied to the Permutation Table shown earlier in this section together with the suppliers file of Fig. 4.1, does indeed yield the Record Reconstruction Table shown in Fig. 4.3. You might also like to check—this is **Exercise 3**—that the Record Reconstruction Table shown in Fig. 3.5 in the previous chapter is correct for the file shown in Fig. 3.2 (and the Field Values Table shown in Fig. 3.4). By the way, did you notice that the Record Reconstruction Tables shown in Figs. 3.5 and 4.3 are different? Why do you think that is?

4.6 The Record Reconstruction Table is not Unique

Well, you probably answered the question at the end of the previous section easily enough: The Record Reconstruction Tables of Figs. 3.5 and 4.3 are different because the Permutation Tables from which they were built are different. And the reason the Permutation Tables are different is because they in turn were built from different versions of the original suppliers file, with different record orderings. Of course, the differences in question aren't very important, in a sense, because every possible record ordering in the original file can in principle be reconstructed from either of the two Record Reconstruction Tables.

However, there's another reason (a more important reason) why the Record Reconstruction Table is, in general, not unique. Indeed, we can obtain different Record Reconstruction Tables even without starting from different versions of the file, as I'll now demonstrate.

First of all, consider the Permutation Table from the previous section once again:

	1	*2*	*3*	*4*
	S#	SNAME	STATUS	CITY
1	*4*	*2*	*3*	*2*
2	*3*	*5*	*1*	*1*
3	*5*	*1*	*4*	*4*
4	*1*	*3*	*2*	*3*
5	*2*	*4*	*5*	*5*

For definiteness, let's focus on the STATUS permutation, which, if you'll glance back at the beginning of the previous section, you'll see is *3, 1, 4, 2, 5* (as indeed you can also see from column *3* of the Permutation Table itself). As you'll recall, the meaning of that permutation is that if we sort the suppliers file of Fig. 4.1 by ascending status value, the records of that file will appear in the indicated sequence *3, 1, 4, 2, 5*. However, I wasn't being entirely honest with you when I discussed these ideas previously. Since records *1* and *4* (for suppliers S4 and S1, respectively) both contain the status value 20, and records *2* and *5* (for suppliers S5 and S3, respectively) both contain the status value 30, *the STATUS permutation is not unique*. In fact, there are four possible STATUS permutations that are all equally valid (and all equivalent, in a sense):

- *3, 1, 4, 2, 5*
- *3, 4, 1, 2, 5*
- *3, 1, 4, 5, 2*
- *3, 4, 1, 5, 2*

Analogous remarks apply to the CITY permutation, though not to the S# permutation (nor to the SNAME permutation, as it happens, in this particular example).

It follows from the foregoing that the Permutation Table is not unique, and hence that the Record Reconstruction Table is not unique either. For example, here's another valid Permutation Table corresponding to the file of Fig. 4.1:

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	4	2	3	2
2	3	5	4	1
3	5	1	1	4
4	1	3	5	5
5	2	4	2	3

And here's the corresponding Record Reconstruction Table:

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	5	5	5	5
2	4	4	3	4
3	2	3	2	1
4	3	1	4	3
5	1	2	1	2

Let's just confirm that this Record Reconstruction Table can indeed be used to reconstruct the records of the original file of Fig. 4.1. Let's start (arbitrarily) at cell [*4,1*]. Then:

- Cell [*4,1*] of the Field Values Table contains the supplier number S4; cell [*4,1*] of the Record Reconstruction Table contains *3,* so next we go to cell [*3,2*].

- Cell [*3,2*] of the Field Values Table contains the supplier name Clark; cell [*3,2*] of the Record Reconstruction Table contains *3* again, so next we go to cell [*3,3*].

- Cell [*3,3*] of the Field Values Table contains the status value 20; cell [*3,3*] of the Record Reconstruction Table contains *2,* so next we go to cell [*2,4*].

- Cell [*2,4*] of the Field Values Table contains the city name London; cell [*2,4*] of the Record Reconstruction Table contains *4,* and we're back where we started, having reconstructed the supplier record:

	S#	SNAME	STATUS	CITY
1	S4	Clark	20	London

The fact that the Record Reconstruction Table is, in general, nonunique in the foregoing sense will turn out to be very important in Chapter 7. Note, however, that although the Record Reconstruction Table is indeed nonunique as we've just seen, in what follows I'll continue to talk in terms of "the" Record Reconstruction Table much of the time, just to keep things simple.

Endnotes

1. I've split the material across two chapters simply because there's such a lot of ground to cover—I didn't want you to have to deal with one great big monolithic and indigestible chapter, especially at this point in the book, and especially when the topics involved are so fundamental.
2. The reference is to Exercise 1 in Appendix A. I'll follow this numbering style for exercises throughout the rest of the book. (By the way, this is as good a place as any to remind you that Appendix A doesn't just contain the exercises as originally stated—it also includes much of the necessary background material. In the case of Exercise 1, for example, it includes a repeat of the Field Values Table and Record Reconstruction Table from Fig. 4.3 and a repeat of the pointer rings from Fig. 4.4.)
3. The hash in question would have to be *indirect*, however [48,60]—it couldn't be a simple "direct" hash as described in Chapter 2, because of the inherently ordered nature of both the Field Values Table and the Record Reconstruction Table. But we can have as many hashes as we like, so long as they *are* indirect; in the very unlikely extreme, we could even have a hash on every column of each of the two tables. Note, however, that the performance improvements that hashing might provide are likely to be small in comparison to the fundamental improvements that the TR structures offer in the first place.

5 Core Concepts (Continued)

5.1 Introduction

This chapter continues our examination of the core constructs of the TR model (principally the Field Values and Record Reconstruction Tables). However, the chapter is rather more of a potpourri than the previous one. Its structure is as follows. Following this short introductory section, Section 5.2 offers some general observations regarding performance. Section 5.3 then briefly surveys the TR operators, and Sections 5.4 and 5.5 take another look at how the Record Reconstruction Table is built and how record reconstruction is done. Sections 5.6 and 5.7 describe some alternative perspectives on certain of the TR constructs introduced in Chapter 4. Finally, Section 5.6 takes a look at some alternative ways of implementing some of the TR structures and algorithms also first described in that previous chapter.

5.2 Some Remarks on Performance

It seems to me undeniable that the mechanisms described in the previous chapter for representing and reconstructing records and files are vastly different from those found in conventional DBMSs, and I presume you agree with this assessment. At the same time, however, they certainly look pretty *complicated* ... How does all of that complexity square with the claims I made in Chapter 1 regarding good performance? Let me remind you of some of the things I said there:

> [TR is] a technology that lets us build database management systems (DBMSs) that are ... orders of magnitude faster than any previous system ... [A] relational system ... using TR technology should dramatically outperform even the fastest of those [previous] systems ... [and] I don't just mean that queries should be faster ... [Updates] should be faster as well.
>
> *—from Chapter 1*

Well, let me say a little more now regarding query performance specifically (I haven't really discussed updates yet, so I'll have to come back to the question of update performance later—actually in the next chapter). Now, any given query involves two logically distinct processes:

a) Finding the data that's required, and then

b) Retrieving that data.

TR is designed to exploit this fact. Precisely because it separates field value information and linkage information, it can treat these two processes more or less independently. To find the data, it uses the Field Values Table; to retrieve it, it uses the Record Reconstruction Table. (These characterizations aren't 100 percent accurate, but they're good to a first approximation—good enough for present purposes, at any rate.) And the Field Values Table in particular is designed to make the finding of data very efficient (for example, via binary search), as we saw in Chapter 4. Of course, it's true that subsequent retrieval of that data then involves the record reconstruction process, and this latter process in turn involves a lot of pointer chasing, but:

- Even in a disk-based implementation, the system will do its best to ensure that pertinent portions of both the Field Values Table and the Record Reconstruction Table are kept in main memory at run time, as we'll see in Part III. Assuming this goal is met, the reconstruction will be done at main-memory speeds.

- The "frills" to be discussed in Chapters 7-9 (as well as others that are beyond the scope of this book) have the effect, among other things, of dramatically improving the performance of various aspects of the reconstruction process.

- *Most important of all:* Almost always, finding the data that's wanted is a much bigger issue than returning that data to the user is. In a sense, the design of the TR internal structures is biased in favor of the first of these issues at the expense of the second. Observe the implication: *The more complex the query, the better TR will perform*—in comparison with traditional approaches, that is. (Of course, I don't mean to suggest by these remarks that record reconstruction is slow or inefficient—it isn't—nor that TR performs well on complex queries but not on simple ones. I just want to stress the relative importance of finding the data in the first place, that's all.)

I'd like to say more on this question of query performance. In 1969, in his very first paper on the relational model [5], Codd had this to say:

> Once aware that a certain relation exists, the user will expect to be able to exploit that relation using any combination of its attributes as "knowns" and the remaining attributes as "unknowns," because the information (like Everest) is there. This is a system feature (missing from many current information systems) which we shall call (logically) symmetric exploitation of relations. Naturally, symmetry in performance is not to be expected.
>
> *—E. F. Codd*

Note: I've reworded Codd's remarks just slightly here. In particular, the final sentence (the caveat concerning performance) didn't appear in the original 1969 paper [5] but was added in the expanded 1970 version [6].

Anyway, the point I want to make is that the TR approach gives us symmetry in performance, too—or, at least, it comes much closer to doing so than previous approaches ever did. This is because, as we saw in Chapter 4, the separation of field values from linkage information effectively allows the data to be physically stored in several different sort orders simultaneously. When Codd said "symmetry in performance is not to be expected," he was tacitly assuming a direct-image style of implementation, one involving auxiliary structures like those described in Chapter 2. However, as I said in that chapter:

> [Auxiliary structures such as pointer chains and] indexes can be used to impose different orderings on a given file and thus (in a sense) "level the playing field" with respect to different processing sequences; all of those sequences are equally good from a logical point of view. But they certainly aren't equally good from a performance point of view. For example, even if there's a city index, processing suppliers in city name sequence will involve (in effect) random accesses to storage, precisely because the supplier records aren't physically stored in city name sequence but are scattered all over the disk.
>
> *—from Chapter 2*

As we've seen, however, these remarks simply don't apply to the TR data representation.

And now I can address another issue that might possibly have been bothering you. We've seen that the TR model relies heavily on pointers. Now, the CODASYL "network model" [14,25] also relies heavily on pointers—as the "object model" [3,4,28,29] and "hierarchic model" [25,56] both do also, as a matter of fact—and I and many other writers have criticized it vigorously in the past on exactly that score (see, for example, references [10], [21], and [37]). So am I arguing out of both sides of my mouth here? How can TR pointers be good while CODASYL pointers are bad?

Well, in fact there are several differences between TR pointers and CODASYL pointers. The biggest of those differences has to with the question of the *target audience:* Who is the user[1] of the technology supposed to be in each case?

- The target audience for TR is clearly **system programmers,** whose job it is to build DBMSs and other data management systems—for example, data mining tools—on top of a TR implementation. In other words, a TR implementation, viewed in isolation, is not and is not meant to be a complete DBMS as such, and the TR model is not and is not meant to be the application programming interface to such a DBMS.

- By contrast, the target audience for CODASYL is **application programmers,** whose job it is to build application systems—for example, a payroll system—on top of a CODASYL DBMS. In other words, a CODASYL implementation definitely *is* (or was) meant to be a complete DBMS as such,[2] and the "CODASYL model" is (or was) meant to be the application programming interface to such a DBMS.

And, of course, it's well established that system programmers do need to be able to make use of pointers, for all kinds of reasons. On the other hand, it's equally well established that allowing—or, worse, requiring—application programmers to make use of pointers is a very bad idea, again for all kinds of reasons (indeed, this fact is a major justification for the exclusion of pointers from the relational model [40]).

Just as an aside, I simply can't let the foregoing remarks go by without mentioning the distressing fact that, as I write, most of the mainstream SQL vendors (following the current SQL standard [53]) are busily incorporating pointers—pointers, that is, that are visible to the application programmer—into their "model." Reference [40] refers to this "feature" of SQL as a **Great Blunder,** and explains just why it *is* a blunder; for example, it shows among other things that pointers and a good model of type inheritance are fundamentally incompatible. And reference [30] gives numerous additional reasons as to why the relational model should categorically not be extended or "improved" to include pointers.

Back to the comparison with CODASYL. Another big difference between CODASYL pointers and TR pointers is that CODASYL pointers apply at the *record* level, while TR pointers apply at the *field* level. One consequence of this difference is that CODASYL structures are in fact parent/child structures, in the sense of Chapter 2; as a direct consequence, they suffer from all of the problems of such structures identified in that chapter. In particular, therefore, while CODASYL pointers might in principle be used to provide "symmetric exploitation" (although they certainly aren't used that way in practice), they certainly don't provide symmetry in performance, because the records can be physically clustered in at most one way (again, see Chapter 2). The same is not true with TR pointers, as we know.

5.3 TR Operators

Now we come to another issue that I've been ducking slightly so far. I've claimed repeatedly that TR is a model. As such, it must provide some operators to operate on the "objects"—the Field Values Table, etc.—that I've been concentrating on so far (as well as providing those "objects" themselves, of course). So what are the TR operators?

Well, at the most fundamental level, of course, TR certainly includes everything necessary to build, search, access, and maintain tables such as the Field Values Table, including in particular all of the obvious subscripting, assignment, and comparison operators. It also includes operators for allocating and deallocating storage and carrying out other such utility functions. All of these operators are only to be expected.

At a slightly higher level, TR also includes a set of operators that are described in some detail in reference [63]. However, most of those "higher-level" operators are still quite low-level in nature; indeed, most of them are intended for use in the implementation of still higher-level operators that will presumably be used by the system programmers mentioned in the previous section. For that reason, I don't think it's worth getting into details of those lower-level operators here. However, I do want to say a little about the "system programming interface" ones, even though those operators aren't really primitive operators of the TR model as such. (Indeed, reference [63] shows how they could actually be implemented in terms of the lower-level operators that it does describe.)

Let's assume that techniques such as those discussed in Chapter 4—for example, binary searches on columns of the Field Values Table—have already been used to determine that some particular record is of interest. Let me immediately explain what I mean when I say that some record has been "determined to be of interest." To be specific:

- When I say "some particular record," I mean a record of the applicable user file.

- When I say such a record is "of interest," I mean we want the record in question—or the tuple corresponding to that record, rather—to be retrieved, deleted, or updated. (Inserting a new record or tuple is different, of course; we can't sensibly talk about the new record having been "determined to be of interest," because the record doesn't exist yet—at least, not in the database.)

- And when I say techniques have been used "to determine" that the record in question is of interest, I mean it's been determined that some cell [*i,j*] of the Record Reconstruction Table corresponds to some portion of that record; more precisely, it's been determined that cell [*i,j*] of the Record Reconstruction Table contains a pointer that points to a cell in the Field Values Table that contains some portion of that record. In other words, the record in question is that unique record that corresponds to that particular cell [*i,j*] of the Record Reconstruction Table. It's convenient to say, loosely, that the record in question "passes through" that cell [*i,j*] of the Record Reconstruction Table.

  With all of that preamble out of the way, then, the TR operators I want to consider are as follows:

- **Retrieve** the record passing through cell [*i,j*] of the Record Reconstruction Table.

- **Delete** the record passing through cell [*i,j*] of the Record Reconstruction Table.

- **Update** the record passing through cell [*i,j*] of the Record Reconstruction Table.

- **Insert** a new record.

Of these operators, **retrieve** has effectively been discussed at length already in Chapter 4—it's essentially just the business of record reconstruction as described in that chapter (in Section 4.4 in particular). The other three operators are discussed in detail in the next chapter.

One last remark to close the present section: If you happen to be familiar with traditional approaches to implementing the relational model, you might have been expecting to see certain other operators mentioned in the discussion above. For example, the System R prototype [1] consisted of a frontend called the Relational Data System (RDS) and a backend called the Relational Storage System (RSS);[3] the RDS translated user requests—SQL statements, in other words—into RSS operations, and those RSS operations performed such functions as searching indexes, committing and rolling back transactions, and so forth. And those RSS operators included many things that have no direct counterpart in the TR model at all. Some of those operators (for example, those to do with indexes) are omitted from TR because TR simply has no need of them. However, others (for example, COMMIT and ROLLBACK) are omitted because such functionality is meant to be provided above the TR interface. (Indeed, the RSS was really an entire multiuser DBMS in its own right, albeit one whose user interface was rather low-level. By contrast, TR—or a TR implementation, rather—is *not* a complete DBMS in its own right; rather, it's meant among other things to serve as the storage manager component for such a DBMS.)

5.4 Building the Record Reconstruction Table: An Alternative Approach

In the introduction to Chapter 4, I said I'd occasionally make some mention of alternative implementation schemes for certain aspects of the TR model. In keeping with that promise, I'd now like to take a look at an alternative way of building the Record Reconstruction Table. *Note:* It might help to repeat the point from Chapter 4 that the Record Reconstruction Table is built directly from the file (the Field Values Table isn't involved in the process at all).

Now, you might recall that in Chapter 4, Section 4.5, I showed how we could use the Permutation Table instead of the Record Reconstruction Table in order to perform the record reconstruction process. However, I also said it wouldn't be very efficient to use the Permutation Table in that way, because we'd have to do sequential searches on the columns of that table in order to find the record numbers (that's why we replaced the Permutation Table by the Record Reconstruction Table in the first place). The trouble is, though, the algorithm for *building* the Record Reconstruction Table from the Permutation Table still involves doing those same sequential searches—admittedly only when the Record Reconstruction Table is built, not every time it's used, but those searches still represent overhead, and it would be nice to eliminate that overhead if we can.

It turns out we can improve matters by exploiting the **inverses** of the permutations in the Permutation Table. Consider once again the original Permutation Table from Chapter 4, Section 4.5 (see Fig. 5.1). As you can see from that table, the S# permutation (for example) is the sequence

```
4, 3, 5, 1, 2
```

The meaning, to remind you, is that if the records of the original file (see Fig. 4.1 in Chapter 4) are sorted into ascending S# order, record *4* will appear first, record *3* will appear second, and so on. And the inverse of this permutation is the sequence

```
4, 5, 2, 1, 3
```

This inverse permutation is that unique permutation that, if applied to the original sequence *4, 3, 5, 1, 2,* will produce the sequence *1, 2, 3, 4, 5*. (If *SEQ* is the original sequence *4, 3, 5, 1, 2,* then the fourth entry in *SEQ* is *1,* the fifth is *2,* the second is *3,* and so on.)

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	4	2	3	2
2	3	5	1	1
3	5	1	4	4
4	1	3	2	3
5	2	4	5	5

Fig. 5.1: Permutation Table for the suppliers file of Fig. 4.1

More generally, if we think of any given permutation as a vector *V,* then the inverse permutation *V′* can be obtained in accordance with the simple rule that if *V*[*i*] = *i′*, then *V′*[*i′*] = *i*. Applying this rule to each of the permutations in our given Permutation Table, we obtain the **Inverse Permutation Table** shown in Fig. 5.2. (**Exercise 4:** Check that the table is correct.)

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	4	3	2	2
2	5	1	4	1
3	2	4	1	4
4	1	5	3	3
5	3	2	5	5

Fig. 5.2: Inverse Permutation Table corresponding to Fig. 5.1

We can now use the Inverse Permutation Table to build the Record Reconstruction Table without doing any sequential searches. For example, the first (S#) column of the Record Reconstruction Table can be built as follows:

> Go to cell [*i,1*] of the Inverse Permutation Table. Let that cell contain the value *r;* also, let the next cell to the right, cell [*i,2*], contain the value *r′*. Go to the *r*th row of the Record Reconstruction Table and place the value *r′* in cell [*r,1*].

Executing this algorithm for *i = 1, 2, ..., 5* yields the entire S# column of the Record Reconstruction Table. The other columns are built analogously. **Exercise 5:** Check that the foregoing algorithm, when applied to the given Inverse Permutation Table, does indeed produce the Record Reconstruction Table shown in Fig. 4.3 in Chapter 4. (Doing this exercise should convince you that this algorithm is much easier to apply than the one given in Chapter 4; it should also make you understand why the algorithm works, if you haven't figured it out already. In future chapters, when I need to build a Record Reconstruction Table, I'll use this new algorithm.)

5.5 Record Reconstruction Revisited

Like the previous section, this one too is concerned with a possible implementation alternative. In that previous section, the Permutation and Inverse Permutation Tables served as purely temporary structures, used in building the Record Reconstruction Table but then discarded. However, it would be possible not to discard them after all, but rather to use them together as a replacement for the Record Reconstruction Table. For example, consider the following SQL query (a projection of a restriction):

```
SELECT S.S#, S.STATUS
FROM S
WHERE S.CITY = 'London' ;
```

We can implement this query as follows:

- *Step 1:* Use a binary search to find the London entries in the Field Values Table (see Fig. 4.2 in Chapter 4) and extract the corresponding row numbers. In the example, this step yields the row numbers *2* and *3*.

- *Step 2:* Use those row numbers to look up entries in the CITY column of the Permutation Table (see Fig. 5.1). This step yields the corresponding *record* numbers, *1* and *4*.

- *Step 3:* Use those record numbers as *row* numbers to look up entries in the S# column of the Inverse Permutation Table (see Fig. 5.2). This step yields the row numbers *4* and *1,* and these values can be used to access the corresponding S# values in the Field Values Table, S1 and S4.

- *Step 4:* Likewise, use the record numbers from Step 2 to look up entries in the STATUS column of the Inverse Permutation Table. This step yields the row numbers *2* and *3,* and these values can be used to access the corresponding status values in the Field Values Table, which are both 20, as it happens. Execution of the query is now complete.

Comparing the foregoing with what we would have had to have done using the Record Reconstruction Table, we can see that one advantage is that we don't have to chase pointers through columns that aren't involved in the query (a fact that could be useful in implementing projection operations, for example). On the other hand, the Permutation and Inverse Permutation Tables together occupy twice as much space as the Record Reconstruction Table does.

Having said all of the above, let me now say that for definiteness I'll assume an implementation from this point forward that does do reconstruction via the Record Reconstruction Table, not via the Permutation and Inverse Permutation Tables (barring explicit statements to the contrary). In other words, I'll assume the Permutation and Inverse Permutation Tables aren't kept around at run time.

5.6 Pointers are Field Value Surrogates

Consider Fig. 5.3, a repeat of Fig. 4.3 from Chapter 4, which shows the Field Values Table for the suppliers file of Fig. 4.1 together with a corresponding Record Reconstruction Table; more specifically, consider the Field Values Table in that figure. Clearly, the position—that is to say, the row number—of any given field value within its containing column in that table serves as a unique encoding, or **surrogate,** for the value in question (in other words, the table provides an *encoding mechanism* for its values). For example, consider the CITY column, which contains, in sequence, the city names Athens, London, London, Paris, and Paris; clearly, the corresponding row numbers *1, 2, 3, 4, 5* can be regarded as surrogates for those values (in sequence as indicated).[4] What's more, those very same row numbers can also be regarded as surrogates for the supplier numbers S1, S2, S3, S4, and S5; the names Adams, Blake, Clark, Jones, and Smith; and the status values 10, 20, 20, 30, and 30 (in sequence as indicated in every case).

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	S1	Adams	10	Athens
2	S2	Blake	20	London
3	S3	Clark	20	London
4	S4	Jones	30	Paris
5	S5	Smith	30	Paris

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	5	4	4	5
2	4	5	2	4
3	2	2	3	1
4	3	1	1	2
5	1	3	5	3

Fig. 5.3: Field Values Table of Fig. 4.2 and a corresponding Record Reconstruction Table

It follows from the foregoing that the Record Reconstruction Table can be regarded as containing such field value surrogates (and likewise for the Permutation and Inverse Permutation Tables, of course). For example, the STATUS column in the Record Reconstruction Table of Fig. 5.3 contains, in sequence, the row numbers *4, 2, 3, 1, 5*. These row numbers are surrogates for CITY values (not STATUS values); they stand for the values Paris, London, London, Athens, and Paris, respectively, and this sequence is the sequence in which the city names will appear if we ask to see suppliers in status sequence, thus:

```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY
FROM S
ORDER BY STATUS ;
```

So now we know that row numbers serve as surrogates for field values, and the Record Reconstruction Table in particular contains such surrogates. This alternative perspective is occasionally useful, as we'll see in Part III of this book. Now, in the TR model as I've described it so far (and indeed as I'll continue to describe it throughout the remainder of this book), the surrogates in question are *always* row numbers. But other surrogate schemes are possible and could be useful in different implementation environments—and so such alternative schemes are yet another illustration of the fact that the TR model is capable of many different concrete implementations. Further details are beyond the scope of this book.

5.7 The Field Values Table is a Directory

In this section, I want to consider (briefly) another alternative perspective that can also be helpful on occasion. Consider the Field Values Table in Fig. 5.3 once again; more specifically, consider the CITY column in that table. Let *c* be a value (city name) in that column, and let the containing cell *C* be cell [*i,4*] (meaning city *c* has surrogate *i*). Then that subscript [*i,4*] identifies the unique cell, *C′* say, in the Record Reconstruction Table that corresponds to cell *C* in the Field Values Table. That cell *C′* in turn is part of a zigzag that allows a record containing the CITY value *c* to be reconstructed.

All of the foregoing should really be familiar to you by now, and I mention it here mainly by way of review. However, let me now point out something that I deliberately haven't mentioned before: namely, that each column of the Field Values Table effectively serves as a kind of **directory**—an index, almost!—to the Record Reconstruction Table and thence, eventually, to the corresponding records. For example, consider the city name Athens, which appears in cell [*1,4*] of the Field Values Table. Following the zigzag through cell [*1,4*] of the Record Reconstruction Table, we obtain the record :

	S#	SNAME	STATUS	CITY
2	S5	Adams	30	Athens

(More precisely, we obtain a version of this record in which the left-to-right field ordering is CITY, then S#, then SNAME, then STATUS.)

Please note carefully, however, that though we might indeed say that each column of the Field Values Table is "almost an index," it certainly isn't an index in the conventional sense of that term. Perhaps a better way to put it would be to say that the column in question—together with the Record Reconstruction Table, which is certainly needed too—*provides the functionality of* an index. That is, the column in question and the Record Reconstruction Table together provide indexing functionality (both direct- and sequential-access functionality) on the user file on the basis of values of the corresponding field.

5.8 Miscellaneous Implementation Alternatives

I'd like to close this chapter by briefly mentioning a few miscellaneous points regarding alternative implementation possibilities for various other TR constructs.

- I've been talking so far as if the linkage information that ties together the field values for a given record must be implemented as a pointer ring or zigzag specifically. But other possibilities exist. For example, we could replace each such ring (within any given Record Reconstruction Table) by two subrings that are connected by means of some common "bridging" column. In the case of suppliers, for example, we might have one subring connecting S# and CITY and another connecting S#, SNAME, and STATUS (column S# being the bridging column, in this particular example). Such an arrangement would be advantageous if projection over S# and CITY is a frequently requested operation—in other words, if queries of the form SELECT S.S#, S.CITY FROM S are common, in SQL terms. What's more, the pointers in such rings or subrings could be either one-way (as I've been assuming so far) or two-way. Other options are also available; one such will turn out to be important in connection with disk-based implementations, and I'll discuss it in detail in Chapter 14.

- I've also been talking so far as if every column in the Field Values Table *has* to be maintained in sorted order. In practice, however, such is not the case; there might well be some columns for which such sorting is just not worth the overhead. An example might be a text column in which the entries are natural-language comments.

- Furthermore, those columns that are sorted don't all have to be sorted in the same way. In our examples, I've shown all columns sorted in ascending sequence. However, it might be better to keep some columns in descending sequence instead; and in the case of columns defined over a user-defined data type, the sort order might be defined in terms of a user-defined "<" operator [40] or in some other way (see Chapter 15, Section 15.5). As reference [63] puts it: "A sort order should be chosen based on its usefulness for display or retrieval purposes *in actual applications*" (my italics).

- Since there's a one-to-one correspondence between the cells of the Field Values Table and the cells of the Record Reconstruction Table, the two tables could if desired be physically collapsed into one. *Note:* This option will cease to be available, however, if the refinements to be discussed in Chapters 8 and 9 are adopted (which in practice they probably will be).

- The same is true, and is perhaps a more sensible proposition, in the case of the Permutation and Inverse Permutation Tables (assuming, of course, that those two tables are indeed both kept around at run time, as we saw in Section 5.5 was a possibility—but I remind you that I'm not going to make that assumption).

- The Permutation and Inverse Permutation Tables differ from the Field Values Table and the Record Reconstruction Table in that there's no reason why their left-to-right column order need be the same as the left-to-right field order of the corresponding file. As a consequence, their left-to-right column orders can be arbitrarily rearranged. *Note:* Actually, the same is effectively true of the Field Values Table and the Record Reconstruction Table as well, inasmuch as such ordering has no meaning at the relational level. In practice, it's probably a good idea to choose a left-to-right column order for those tables such that, if attributes *A* and *B* often appear together in user-level queries (especially if WHERE clauses often include conditional expressions of the form WHERE *A* = ... AND *B* = ...), then the columns corresponding to those attributes are adjacent in the two tables. In particular, these remarks are true of attributes that are components of the same key; that is, columns that correspond to attributes in a multiattribute key should generally be adjacent. See also the further remarks on this topic at the very end of Chapter 8.

Endnotes

1. The word *user* is always a little ambiguous. I don't mean it here in the sense of the Chapter 3 "user level," I mean whoever is the direct, immediate user of the technology in question.
2. It's true that, with hindsight, we might regard CODASYL (like TR) not as a model for "a complete DBMS as such" but rather as an implementation technology, even though such was not the original intent. However, CODASYL is fundamentally unsuited to that role for the kinds of reasons discussed in Chapter 2, as well as many others.
3. The backend name was rather inappropriate, because relations aren't a storage-level concept at all. In any case, the name was subsequently changed for political reasons to *Research* Storage System.
4. Note that the very same field value can have two or more distinct surrogates; for example, the value London has surrogates *2* and *3*. If the refinements to be discussed in Chapters 8 and 9 are adopted, however, surrogates will be unique, in the sense that every field value will have just one of them.

6 Implementing the Update Operators

6.1 Introduction

By now I hope it's clear that, even without the refinements to be discussed in later chapters, the TR model is certainly good for retrieval. (At least in principle! I'll describe in more detail how retrievals are actually implemented in Chapter 10.) But what about updates?[1] Conventional wisdom has always been that a given data structure can be good for either retrieval or update, but not both. In a direct-image implementation, for example, indexes are generally held to be good for retrieval but bad for update. So what about TR? How are updates done in TR? This chapter examines this question.

To repeat from Chapter 5, then, the operators we need to consider are as follows (see Section 5.3):

- *INSERT:* Insert a new record.
- *DELETE:* Delete the record "passing through" cell [*i,j*] of the Record Reconstruction Table.
- *UPDATE:* Update the record "passing through" cell [*i,j*] of the Record Reconstruction Table.

Note: The notion of a record "passing through" some cell of the Record Reconstruction Table was also explained in Section 5.3.

Section 6.2 immediately following discusses the three update operators in general terms; Sections 6.3 then presents a detailed example, and Section 6.4 discusses *the swap algorithm*. Section 6.5 briefly describes an alternative implementation technique that makes use of an *overflow structure*. Finally, Section 6.6 offers some observations regarding the performance aspects of TR update operations.

6.2 Overview

It's convenient to begin by discussing the INSERT operator specifically. Consider the suppliers file shown in Fig. 6.1 (it's the same as the one shown in Fig. 4.1 in Chapter 4, except that the last record, the one for supplier S3, has been omitted). Figs. 6.2 and 6.3 show the corresponding Field Values Table and a corresponding Record Reconstruction Table, respectively. **Exercise 6:** Check that these tables are correct.

field sequence: 1 2 3 4

record sequence:	S#	SNAME	STATUS	CITY
1	S4	Clark	20	London
2	S5	Adams	30	Athens
3	S2	Jones	10	Paris
4	S1	Smith	20	London

Fig. 6.1: A suppliers file

column sequence: 1 2 3 4

row sequence:	S#	SNAME	STATUS	CITY
1	S1	Adams	10	Athens
2	S2	Clark	20	London
3	S4	Jones	20	London
4	S5	Smith	30	Paris

Fig. 6.2: Field Values Table corresponding to the file of Fig. 6.1

column sequence: 1 2 3 4

row sequence:	S#	SNAME	STATUS	CITY
1	4	4	4	4
2	3	2	2	3
3	2	1	3	1
4	1	3	1	2

Fig. 6.3: Record Reconstruction Table corresponding to the file of Fig. 6.1

Now suppose the user asks the system to insert the following tuple into the suppliers relation:

S#	SNAME	STATUS	CITY
S3	Blake	30	Paris

In terms of the file of Fig. 6.1, of course, we can imagine a new record corresponding to this tuple simply being appended at the end, in position *5* (since record ordering within files is arbitrary). If we now rebuild the Field Values Table, it'll appear as shown on the left-hand side of Fig. 6.4 (a copy of the Field Values Table from Fig. 4.3 in Chapter 4). And if we then build a corresponding Record Reconstruction Table, it might appear as shown on the right-hand side of Fig. 6.4

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	S1	Adams	10	Athens
2	S2	Blake	20	London
3	S3	Clark	20	London
4	S4	Jones	30	Paris
5	S5	Smith	30	Paris

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	5	4	4	5
2	4	5	2	4
3	2	2	3	1
4	3	1	1	2
5	1	3	5	3

Fig. 6.4: Field Values Table and Record Reconstruction Table after inserting supplier S3

As you can see by comparing Fig. 6.4 with Figs. 6.2 and 6.3, respectively, inserting supplier S3 has caused both the Field Values Table and the Record Reconstruction Table to change dramatically. It follows that INSERT operations have the potential to be quite disruptive, and hence (possibly) to display very poor performance. What can be done about this problem?

Well, let me say right away that the effect on the Field Values Table is actually not as dramatic as it might appear. Although I've been calling it a table and showing it as a table in figures like Fig. 6.4, the Field Values Table doesn't necessarily have to be physically stored as a table; in fact, it almost certainly won't be. Much more likely, it'll be stored "column-wise" as a set of *vectors* (one-dimensional arrays), or possibly as a set of *chained lists,* one such vector or list for each column. Indeed, such an implementation is virtually certain to be used in practice if the refinements to be discussed in Chapters 8 and 9 are adopted, as we'll see.[2]

For definiteness, let's assume a vector implementation. Of course, those vectors will be kept in the sort orders associated with the corresponding columns of the Field Values Table. As a consequence, the insert point in each such vector for the pertinent field value from the new record is easily determined—for example, by binary search—and the vectors themselves, and hence the overall Field Values Table, are thus easily maintained.

The Record Reconstruction Table is another matter, however. Is there a way to avoid rebuilding the entire table every time a new record is inserted into the user file? The answer, of course, is *yes*. One possible approach is as follows (the details are a little complicated, but the fundamental idea is straightforward): When a record is *deleted* from the user file, we[3] don't physically remove the corresponding entries from the Field Values and Record Reconstruction Tables, we just *flag* those entries as "logically deleted." Those flagged cells can then be regarded as *free space* in each of the two tables. Then, when we subsequently insert a new record, it might be possible to use such flagged cells for the record in question (removing the flags, of course), thereby avoiding the overhead of completely rebuilding the Record Reconstruction Table (and the overhead of completely rebuilding the Field Values Table also, as a matter of fact). Detailed examples illustrating this process are given in Sections 6.3 and 6.4 below.

I should immediately add that the scheme just described in outline makes considerably more sense if the refinements to be discussed in Chapters 8 and 9 are adopted. If they are—and in practice it's virtually certain they will be—then it becomes possible for distinct records at the file level to *share* entries in the Field Values Table. For example, the supplier records for suppliers S2 and S3 might share the entry in that table that contains the city name Paris. Thus, when a new record is inserted, it might well be the case that most if not all of the field values in that record already exist in the Field Values Table—perhaps logically deleted, perhaps not—and such values can simply be shared by that new record with previously existing records. In effect, the ability to share field values in this way means that INSERT operations work at the field level instead of the usual record level—yet another significant difference between the TR approach and conventional implementation technology. Of course, analogous remarks apply to DELETE and UPDATE operations also, as you'd surely expect.

But what if we want to insert new records before any existing records have been deleted (or, perhaps, before *enough* have been deleted)? Well, in conventional database systems, it's customary to leave a certain amount of free space when the database is initially loaded, in order to accommodate future growth more gracefully. In the same kind of way, with TR, we can specify when we first load the database that certain cells in the Field Values and Record Reconstruction Tables are to be initialized (flagged) as "free-space" cells. In this way, the database can start out with the ability to handle subsequent INSERT operations in a nondisruptive manner.

It follows from all of the above that the algorithm for implementing INSERT operations looks something like this (and here I'm assuming that field values are indeed shared among records in the manner to be explained in detail in Chapter 8):

> *Step 1:* Let *r* be the record to be inserted, and let *f1, f2,* ..., *fn* be the field values within *r*.
>
> *Step 2:* For each *j* in turn (*j* = 1, 2, ..., *n*), do *Step 3*.
>
> *Step 3:* Search column *j* of the Field Values Table for *fj*. If *fj* is not found, insert it. Adjust the Field Values Table to show that the *fj* entry is used by record *r*. Adjust the Record Reconstruction Table accordingly.

Note: Just what it means to adjust the Field Values Table to show that the *fj* entry is used by record *r* is explained in Chapter 8 (Section 8.2, subsection "Row Ranges").

As for DELETE operations, we already know in essence how these operations are implemented: The appropriate entries in the Field Values and Record Reconstruction Tables are simply flagged as logically deleted and thus become free-space cells. As noted earlier, DELETEs are thus effectively done at the field level, rather than the more usual record level.

Analogous remarks apply to UPDATE operations as well, of course, since they can be thought of logically as a DELETE followed by an INSERT. Note in particular that if record *r* is updated to become record *r′*,[4] but the value of field *F* in *r′* is the same as that in *r* (meaning, loosely, that "field *F* hasn't been updated"), then the internal-level operation that the implementation has to execute for field *F* is essentially "Do nothing"!—in effect, the new record *r′* and the old record *r* can simply share the applicable field value. (Of course, the foregoing is just a manner of speaking; I don't mean to suggest that the old record *r* is still kept around after the update has been done. Though it might be, if the database in question is a temporal one [42].)

6.3 A Detailed Example

Now let's take a closer look at exactly how updates are done in TR. *Note:* Actually TR supports a variety of distinct update techniques, and it's obviously not possible to cover them all in a book of this nature. The explanations that follow are thus certainly not meant to be exhaustive; rather, they're offered just as an indication of the kinds of techniques that might be used in practice.

By way of an example, consider the suppliers file shown in Fig. 6.5 (it's the same as the one shown in Fig. 6.1, except that supplier S3 has been reinstated and two new suppliers, S6 and S7, have been added). Figs. 6.6 and 6.7 show the corresponding Field Values Table and a corresponding Record Reconstruction Table, respectively. **Exercise 7:** Once again, check that these tables are correct.

field sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
record sequence:	1	S4	Clark	20	London
	2	S5	Adams	30	Athens
	3	S2	Jones	10	Paris
	4	S1	Smith	20	London
	5	S3	Blake	30	Paris
	6	S6	Brady	30	Athens
	7	S7	Patel	40	Haifa

Fig. 6.5: The suppliers file of Fig. 6.1 after inserting suppliers S3, S6, and S7

column sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
row sequence:	1	S1	Adams	10	Athens
	2	S2	Blake	20	Athens
	3	S3	Brady	20	Haifa
	4	S4	Clark	30	London
	5	S5	Jones	30	London
	6	S6	Patel	30	Paris
	7	S7	Smith	40	Paris

Fig. 6.6: Field Values Table corresponding to the file of Fig. 6.5

column sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
row sequence:	1	7	4	6	5
	2	5	6	4	6
	3	2	5	5	7
	4	4	3	1	1
	5	1	1	2	4
	6	3	7	7	2
	7	6	2	3	3

Fig. 6.7: Record Reconstruction Table corresponding to the file of Fig. 6.5

Now suppose the user asks for the records—or, rather, the tuples corresponding to the records—for suppliers S3 (Blake) and S7 (Patel) to be deleted. All the implementation does at this point is follow the applicable zigzags and flag the applicable cells in the Field Values and Record Reconstruction Tables as free space (see Figs. 6.8 and 6.9, where the flags are shown as asterisks). *Note:* I'll refer to such flagged cells as *free cells* from this point forward.

column sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
row sequence:	1	S1	Adams	10	Athens
	2	S2	*Blake	20	Athens
	3	*S3	Brady	20	*Haifa
	4	S4	Clark	30	London
	5	S5	Jones	30	London
	6	S6	*Patel	*30	Paris
	7	*S7	Smith	*40	*Paris

Fig. 6.8: Field Values Table after deleting suppliers S3 and S7

column sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
row sequence:	1	7	4	6	5
	2	5	*6	4	6
	3	*2	5	5	*7
	4	4	3	1	1
	5	1	1	2	4
	6	3	*7	*7	2
	7	*6	2	*3	*3

Fig. 6.9: Record Reconstruction Table after deleting suppliers S3 and S7

Now suppose the user asks the system to insert the following tuple into the suppliers relation:

S#	SNAME	STATUS	CITY
S3	Paige	40	Paris

Note: In practice, the DELETE that caused the old S3 tuple to be deleted and the INSERT that's now asking for the new S3 tuple to be inserted might have been bundled into a single UPDATE request, of course:

```
UPDATE S
SET SNAME = NAME('Paige'),
STATUS = 40
WHERE S# = S#('S3') ;
```

Be that as it may, you can see from Fig. 6.8 that free cells for supplier number S3, status 40, and city name Paris do all exist in the Field Values Table. As for the supplier name, Paige, at least there is a free cell at the right place in the applicable sort order (namely, that for Patel), so we can use that one, too, so long as we change the name it contains from Patel to Paige. Figs. 6.10 and 6.11 show the revised versions of the Field Values Table and the Record Reconstruction Table, respectively. Note that we've removed the flags in both tables from the free cells we've used (namely, cells [*3,1*], [*6,2*], [*7,3*], and [*7,4*]). We've also revised the Record Reconstruction Table so that the corresponding cells are linked together into a zigzag appropriately (to be specific, we've changed the contents of cell [*3,1*] from *2* to *6* and the contents of cell [*7,3*] from *3* to *7*).

column sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
row sequence:	1	S1	Adams	10	Athens
	2	S2	*Blake	20	Athens
	3	S3	Brady	20	*Haifa
	4	S4	Clark	30	London
	5	S5	Jones	30	London
	6	S6	Paige	*30	Paris
	7	*S7	Smith	40	Paris

Fig. 6.10: Field Values Table after inserting (a revised version of) supplier S3

column sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
row sequence:	1	7	4	6	5
	2	5	*6	4	6
	3	2	5	5	*7
	4	4	3	1	1
	5	1	1	2	4
	6	3	7	*7	2
	7	*6	2	3	3

Fig. 6.11: Record Reconstruction Table after inserting (a revised version of) supplier S3

6.4 The Swap Algorithm

As indicated in Sections 6.2 and 6.3, a key notion underlying the TR update algorithms is that cells in the Field Values Table and Record Reconstruction Table can be "recycled" (that is, they can be used and reused, over and over again). To be more specific, deletions cause certain "holes" (free cells) to open up in the TR tables, and those "holes" can then be used by subsequent insertions. However, one important point I deliberately glossed over previously is that it might be necessary *to move those "holes" around from time to time within their containing tables* (I'm speaking pretty loosely here, as I'm sure you'll appreciate). The update algorithms are designed to work in such a way as to keep that moving around localized to as small a region as possible, with the overall objective of keeping the TR tables as static as possible and thereby minimizing the overhead. A variety of techniques can be used to achieve this desirable effect; in this section, I want to focus on just one of those techniques: namely, the so-called *swap algorithm* [63].

In order to illustrate the swap algorithm in action, as it were, let me revise the example from the previous section as follows. First, assume that the Field Values Table and Record Reconstruction Table are again as given in Figs. 6.8 and 6.9, respectively. Now assume that the tuple to be inserted looks like this:

S#	SNAME	STATUS	CITY
S3	Blake	20	Athens

Referring again to Fig. 6.8, we see that:

- Free cells exist in the Field Values Table for supplier number S3 and supplier name Blake, so we can use those cells directly, as in the example in the previous section. That takes care of the S# and SNAME values.

- As for the city name, Athens, there's no free cell for Athens as such, but at least there's a free cell in a suitable position (where by "suitable" I mean a position that doesn't disturb the sort order)—namely, the free cell for Haifa. So we can use that cell too, changing the name it contains from Haifa to Athens. That takes care of the CITY value.

But what do we do about the STATUS value? Not only is there no free cell for status 20, there isn't even any free cell in the right position (that is, immediately before or after the sequence of cells for status 20 that do currently exist).

Observe, however, that at least there *is* a free STATUS cell, for status 30, in row *6* of the Field Values Table (that is, the cell in question is cell [*6,3*]). If we could somehow *swap* that cell with cell [*4,3*]—which also contains the status value 30—the free cell would then be immediately adjacent to the existing sequence of cells for status 20, and we could then use it for the new record, just as we used the Haifa cell for Athens.

In order to implement that swap, we need to reroute the zigzag in the Record Reconstruction Table that runs through cell [*4,3*] so that it runs through cell [*6,3*] instead. That zigzag corresponds (as it happens) to supplier S5, and the effect of the swap will be that supplier S5's status value, 30, will then be the one in cell [*6,3*]—instead of cell [*4,3*]—of the Field Values Table. After making the swap, we can flag cell [*4,3*] as free and "unflag" cell [*6,3*] to mark it "unfree." To spell out the details:

- From Fig. 6.9, we can see that the zigzag for supplier S5 currently looks like this (that is, it currently involves the following sequence of Record Reconstruction Table cells):

```
[5,1], [1,2], [4,3], [1,4]
```

- The swap can thus be done by:

  a) Changing the contents of cell [*1,2*] of the Record Reconstruction Table from *4* to *6*, and

  b) Changing the contents of cell [*6,3*] of the Record Reconstruction Table from *7* to *1*.

- The zigzag for supplier S5 will then look like this:

  ```
  [5,1], [1,2], [6,3], [1,4]
  ```

  (as required).

Now we can replace the status value 30 in cell [*4,3*] of the Field Values Table by the status value 20 (and flag that cell as free and remove the flag from cell [*6,3*], at the same time flagging and unflagging the corresponding cells in the Record Reconstruction Table analogously).

After all the foregoing activity has been completed, there's a free cell in the Field Values Table for every value in the record that we're trying to insert. We can therefore accomplish the desired insertion by linking those free cells into a zigzag in the Record Reconstruction Table—to be specific, a zigzag that looks like this:

```
[3,1], [2,2], [4,3], [3,4]
```

In other words, we set cell [*3,1*] to contain *2,* cell [*2,2*] to contain *4,* cell [*4,3*] to contain *3,* and cell [*3,4*] to contain *3* as well. Also, of course, we remove the flags from all of these previously free cells, in both the Field Values Table and the Record Reconstruction Table. The net effect is shown in Figs. 6.12 and 6.13.

column sequence:	1	2	3	4
row sequence:	S#	SNAME	STATUS	CITY
1	S1	Adams	10	Athens
2	S2	Blake	20	Athens
3	S3	Brady	20	Athens
4	S4	Clark	20	London
5	S5	Jones	30	London
6	S6	*Patel	30	Paris
7	*S7	Smith	*40	*Paris

Fig. 6.12: Field Values Table after inserting (a different revised version of) supplier S3

column sequence:	1	2	3	4
row sequence:	S#	SNAME	STATUS	CITY
1	7	6	6	5
2	5	4	4	6
3	2	5	5	3
4	4	3	3	1
5	1	1	2	4
6	3	*7	1	2
7	*6	2	*3	*3

Fig. 6.13: Record Reconstruction Table after inserting (a different revised version of) supplier S3

Points Arising

The swap algorithm as described above has an interesting side effect in our particular example, as follows: While the Field Values Table still has one set of free cells after the INSERT (because we started with seven records, deleted two, and then inserted one), those cells are no longer chained together. Equivalently, the Record Reconstruction Table also has one set of free cells, but those cells don't form a valid zigzag—the pointer chain is broken. In general, in fact, if we start chasing pointers from a free cell in the Record Reconstruction Table, we won't necessarily find ourselves in a closed ring. Of course, this fact isn't very important, because we never need to do record reconstruction on deleted records anyway.

Another consequence is that, given any particular column of the Record Reconstruction Table, certain row numbers will be duplicated in that column and others will be missing (in general). For example, in Fig. 6.13, columns *3* and *4* both include two *3*'s and no *7*. Note, however, that within any such column:

- No two "unfree" cells will ever contain the same row number.
- Any missing row number would, if present, point to a free cell.

The algorithm has another side effect, too. Suppose we process column *3* (the STATUS column) of the Record Reconstruction Table top to bottom in order to reconstruct the suppliers file in ascending status sequence. With the original version of that table as shown in Fig. 6.7, supplier S5 will precede supplier S6 in the result; with the version of the table shown in Fig. 6.13, by contrast, supplier S6 will precede supplier S5 instead.

And one more point: In the particular example discussed above, the necessary free cell for the new status value, 20, was found in the Field Values Table in the immediately adjacent sequence of cells (namely, the sequence of cells for the next recorded status value, 30). Suppose there had been no free "30" cell but (say) a free "40" cell instead. Then two swaps would have been necessary, one to make that free "40" cell into a free "30" cell, and then another to make that free "30" cell into a free "20" cell. In general, if there are several intervening sequences without any free cells, then several swaps will need to be carried out, each swap moving the free cell one position closer to the place where it's really needed.

6.5 Using an Overflow Structure

In Section 6.2, we saw that insertion of a new record doesn't always require insertion of brand new field values; in fact, we'll see in Chapters 8 and 9 that it *very rarely* requires insertion of brand new field values. And if there are no new field values to insert, the INSERT operation will clearly be faster than it would otherwise be. In Sections 6.3 and 6.4, by contrast, we considered what happens if there are indeed new field values to insert; to be specific, in Section 6.4 we described the swap algorithm for moving "holes" around to get them into the right place for the new values. But there's another way to deal with such new values—one that involves no swapping as such—that might be more efficient in practice, especially in a disk-based implementation. I don't want to get into a lot of detail here, but in essence the technique involves storing the new values in a separate overflow structure and periodically merging the data from that structure into the main database.[5] This technique has the properties that:

- The overflow structure can be thought of as containing its own private Field Values Table and Record Reconstruction Table; thus, for example, binary searches can be used on the overflow data.

- The principal Field Values Table and Record Reconstruction Table—and hence the main database—remain unchanged most of the time; they change only during the process of performing the periodic merge (see the point immediately following).

- The period between successive merges can be quite lengthy. For example, imagine a database containing sales records for every day of the past five years, with a nightly batch insert for that day's figures. Assume that batch insert corresponds to roughly one twentieth of one percent of the total database size. If we allow the overflow structure to grow to (say) ten percent of the total database size before we do the merge, the period between successive merges will be of the order of six or seven *months*.

Using an overflow structure has another big advantage, too: It greatly simplifies the familiar backup and recovery process. In outline, that process works as follows:

- We create a full backup copy of the entire database only when we do a merge (which, as we've just seen, isn't likely to be all that often).

- We create a backup copy of the (comparatively small) overflow structure every time it's been significantly changed—perhaps every night.

- If it's necessary to perform recovery, we simply restore the most recent full backup copy and the most recent overflow backup copy; it's *not* like having to apply a whole series of "incremental backups," which is what conventional systems typically do have to do.

6.6 Some Remarks on Performance

I'll close this chapter with a few remarks on the performance aspects of TR update operations. The fact is, a TR implementation should significantly outperform traditional DBMSs on updates as well as on queries. There are several reasons for this state of affairs:

- *DELETE and UPDATE:* Note first that certain of the remarks made in connection with retrieval performance in Chapter 5 (Section 5.2) apply to the performance of DELETE and UPDATE operations also. To be specific, it's often the case that a significant part of the work involved in deleting or updating data is in finding the relevant data in the first place—and TR is very good at finding data, and finding it fast.

- *INSERT:* Suppose we're trying to insert a new supplier tuple, for supplier S9, say. Then we want to check—or rather, as users, we want the *system* to check—that there's no tuple for supplier S9 in the suppliers relation already. At the implementation level, this requirement implies that the system has to search for a record for supplier S9 before doing the INSERT (if it finds one, the INSERT will have to be rejected, of course). As we saw in Section 6.2, however, it's going to do that search anyway. Once again, therefore, we're talking about the problem of "finding data and finding it fast" (see the previous paragraph; see also the discussion of integrity constraints in Chapter 10, Section 10.10).

- *Lack of redundancy:* There are two points here. First, since there aren't any auxiliary structures such as indexes, there aren't any auxiliary structures such as indexes to update. Second, the refinements to be discussed in Chapters 8 and 9 have the effect of reducing data redundancy still further—dramatically so, in fact—thereby simplifying the update process still further (and considerably).

- *Sorted data:* The precise means (that is, the Field Values Table) by which the stored data is kept in many sort orders simultaneously makes it easy to maintain those sort orders when updates occur—certainly much easier than the corresponding task with indexes or other conventional auxiliary structures.

- *Limiting the scope of impact:* In the real world, even in extremely active systems, updates tend to affect only a tiny portion of the overall database. The TR update algorithms are designed to take advantage of this fact; in effect, they regard the database as a large and static thing, and they keep all changes in a much smaller dynamic repository.[6] In other words (as I said near the beginning of Section 6.4, more or less), they generally try to keep the impact of any given user-level update confined to as small a portion of the database as possible, thereby minimizing the amount of update overhead. This philosophy is in marked contrast to that found in conventional systems; in particular, it's very different from what happens with conventional indexing, where everything is assumed to be completely dynamic (at least potentially), and individual updates can ripple out and cause further updates that need to be applied "all over the database."

Endnotes

1. I remind you from Chapter 1 that I use the term "update" (lower case) to mean the INSERT, DELETE, and UPDATE operators considered generically, and the term "UPDATE" (upper case) to mean the UPDATE operator specifically.
2. It isn't particularly relevant to the present discussion, but you should be aware that analogous remarks apply to the Record Reconstruction Table as well—that is, that table too will almost certainly be stored column-wise. And the same is true for the Permutation and Inverse Permutation Tables also, if those tables are physically stored.
3. "We" here really means the DBMS.
4. I'm being sloppy here. As explained in references [32] and [40], it would be more accurate to talk in terms of record *r* being replaced by record *r′*—but it's conventional to talk in terms of records being updated, even though, strictly speaking, records are values and can't possibly be updated. Analogous remarks apply to fields also.
5. In fact, the technique can be used for values that *aren't* brand new, too.
6. This perception is supported very directly by the overflow structure mechanism sketched in the previous section, but it's effectively supported by TR's other update techniques as well.

7 Major-to-Minor Orderings

7.1 Introduction

The core concepts of the TR model are the Field Values Table and the Record Reconstruction Table, and I've now completed my description of those concepts, at least in their simplest form. As I explained in Chapter 4, however, the TR model also includes many "optional extras" or "frills" (some of which are so important that they'll almost certainly be included in any real implementation), and the time has come to start taking a look at some of those optional extras.

The present chapter is all about a "frill"—*refinement* is really a better word—that applies to the Record Reconstruction Table specifically. (By contrast, the refinements to be discussed in Chapters 8 and 9 apply to the Field Values Table, at least primarily, though in both cases there are implications for the Record Reconstruction Table as well.) Anyway, the refinement I want to discuss right now has to do with **major-to-minor ordering,** by which I mean, in SQL terms, the kind of ordering that results from a query that includes an ORDER BY specification of the form

```
ORDER BY A, B, C, ..., Z
```

As I'm sure you know, the tuples in the result of such a query are ordered, first, by ascending *A* value; then, for any given *A* value, by ascending *B* value; then, for any given *A*-*B* value combination, by ascending *C* value; and so on, finishing up with, for any given *A*-*B*-*C*-... value combination, by ascending *Z* value. The sequence of attribute names *A*, then *B*, then *C*, ..., then *Z*, is said to specify a *major-to-minor ordering* (where *A* is the major attribute, *B* is the next, *C* is the next, ..., and *Z* is the minor attribute).

Note: For each attribute mentioned within a given ORDER BY specification, SQL also lets us specify either ASC or DESC, where ASC means ascending sequence and DESC means descending sequence, and ASC is the default. I showed an example using DESC in Section 4.4 in Chapter 4. In what follows, I'll assume for simplicity that we always want ascending sequence specifically, barring explicit statements to the contrary.

7.2 The Suppliers-Parts-Projects Example

The suppliers relation S has served us well as a basis for examples ever since Chapter 2; however, it isn't really adequate to illustrate the points I want to make in the present chapter. Consider instead, therefore, the shipments relation SPJ depicted in Fig. 7.1.[1] That relation is meant to be interpreted as follows: The indicated supplier (S#) is supplying, or *shipping,* the indicated part (P#) to the indicated project (J#) in the indicated quantity (QTY). The attribute combination {S#,P#,J#} is a key (that is, no two tuples appearing in the relation at the same time ever have the same value for that combination of attributes); in fact, that combination is the *only* key. For definiteness, let's assume that attributes S#, P#, J#, and QTY are defined on types S#, P#, J#, and INTEGER, respectively (where INTEGER is a system-defined type and the other three are user-defined types).

S#	P#	J#	QTY
S1	P1	J1	200
S1	P3	J2	100
S2	P1	J1	200
S2	P1	J2	500
S2	P2	J2	500
S3	P1	J1	100
S3	P2	J2	500
S3	P3	J1	200
S3	P3	J2	200

Fig.7.1: The shipments relation SPJ

Note that I've deliberately chosen sample values for relation SPJ such that:

- No single attribute *A* has the property that every tuple has a value for *A* that's different from the value of *A* in all other tuples in the relation.

- Likewise, no attribute pair *A*-*B* has the property that every tuple has a value for *A*-*B* that's different from the value of *A*-*B* in all other tuples in the relation.

- Likewise, no attribute triple *A*-*B*-*C* has the property that every tuple has a value for *A*-*B*-*C* that's different from the value of *A*-*B*-*C* in all other tuples in the relation—except, of course, for the attribute triple {S#,P#,J#}, which (as we already know) constitutes a key and therefore must have a unique value in every tuple, by definition.

Incidentally, the main reason why the suppliers relation is inadequate for the purposes of the present chapter is that it has a single-attribute key and thus necessarily does have a single attribute that "happens" to have a unique value in every tuple. (In fact, it also has another single attribute, SNAME, that happens to have a unique value in every tuple, but this latter fact truly is a matter of happenstance—that is, {SNAME} isn't a key.) Of course, it follows from the fact that it has a single-attribute key—also from the fact that supplier names happen to be unique—that the suppliers relation also has several attribute pairs and several attribute triples that also have unique values in every tuple, a fortiori.

Fig. 7.2 shows a possible file corresponding to the relation of Fig. 7.1 (for simplicity, I've shown the records and fields of that file in the orderings suggested by Fig. 7.1), and Fig. 7.3 shows the corresponding Field Values Table.

	1	2	3	4
	S#	P#	J#	QTY
1	S1	P1	J1	200
2	S1	P3	J2	100
3	S2	P1	J1	200
4	S2	P1	J2	500
5	S2	P2	J2	500
6	S3	P1	J1	100
7	S3	P2	J2	500
8	S3	P3	J1	200
9	S3	P3	J2	200

Fig.7.2: File corresponding to the shipments relation of Fig. 7.1

	1	2	3	4
	S#	P#	J#	QTY
1	S1	P1	J1	100
2	S1	P1	J1	100
3	S2	P1	J1	200
4	S2	P1	J1	200
5	S2	P2	J2	200
6	S3	P2	J2	200
7	S3	P3	J2	500
8	S3	P3	J2	500
9	S3	P3	J2	500

Fig.7.3: Field Values Table corresponding to the file of Fig. 7.2

7.3 A Preferred Record Reconstruction Table

Recall now from Chapter 4 that the Record Reconstruction Table corresponding to a given file (and corresponding Field Values Table) is, in general, not unique. *We can turn this fact to our advantage*. It turns out that, given a particular file (and Field Values Table), certain Record Reconstruction Tables are "more equal than others," in the sense that they have certain very desirable properties. In this section, I'll show an example of what such a "preferred" Record Reconstruction Table might look like, and I'll explain what some of those desirable properties are. In the next section, I'll show how such preferred Record Reconstruction Tables can be built in the first place.

The particular preferred Record Reconstruction Table I want to discuss first is shown in Fig. 7.4.

	1	2	3	4
	S#	P#	J#	QTY
1	2	1	2	2
2	8	2	3	6
3	3	3	4	1
4	4	7	5	3
5	5	8	1	8
6	1	9	6	9
7	6	4	7	4
8	7	5	8	5
9	9	6	9	7

Fig.7.4: A preferred Record Reconstruction Table for the file of Fig. 7.2

Let's just do a spot check on the table in Fig. 7.4 to make sure it does indeed reconstruct at least one record correctly. Starting arbitrarily at cell [*4,3*], we have:

- Cell [*4,3*] of the Field Values Table contains the project number J1; cell [*4,3*] of the Record Reconstruction Table contains *5,* so—remembering that that *5* means *row* number *5* and the next *column* is column number *4*—we go next to cell [*5,4*].

- Cell [*5,4*] of the Field Values Table contains the quantity 200; cell [*5,4*] of the Record Reconstruction Table contains *8,* so now we go to cell [*8,1*] (the next or "fifth" column in fact wrapping around to the first).

- Cell [*8,1*] of the Field Values Table contains the supplier number S3; cell [*8,1*] of the Record Reconstruction Table contains *7,* so we go to cell [*7,2*].

- Cell [*7,2*] of the Field Values Table contains the part number P3; cell [*7,2*] of the Record Reconstruction Table contains *4,* and so we're back where we started, having reconstructed the shipment record:

	S#	P#	J#	QTY
8	S3	P3	J1	200

(More precisely, we've reconstructed a version of this record in which the left-to-right field sequence is J#, then QTY, then S#, then P#.)

Anyway, at least we do now have some empirical evidence tending to confirm that the table shown in Fig. 7.4 is indeed a valid Record Reconstruction Table for the file of Fig. 7.2. But what's special about it? Why is it "preferred"?

Well, suppose we try reconstructing the entire file, starting at cell [*1,1*] of each of the two tables for the first record in that reconstruction and then continuing down column *1* (the S# column)—that is, starting successive record reconstructions with cells [*2,1*], [*3,1*], ..., [*9,1*] for the second, third, ..., ninth record in the overall file reconstruction process. Suppose we then do the same thing again, but this time going down column *2* (the P# column) instead; and then again, going down column *3* (the J# column); and one final time, going down column *4* (the QTY column). What happens?

Unfortunately, I'm afraid you're going to have to do some work here—it's no good my just presenting you with the results, you really need to work through the process and determine those results for yourself (this is **Exercise 8**). When you do, however, you'll find that the results are in fact as shown in Fig. 7.5 (I've labeled them a., b., c., and d. for purposes of future reference).

a.

	1	2	3	4
	S#	P#	J#	QTY
1	S1	P1	J1	200
2	S1	P3	J2	100
3	S2	P1	J1	200
4	S2	P1	J2	500
5	S2	P2	J2	500
6	S3	P1	J1	100
7	S3	P2	J2	500
8	S3	P3	J1	200
9	S3	P3	J2	200

b.

	1	2	3	4
	S#	P#	J#	QTY
1	S3	P1	J1	100
2	S1	P1	J1	200
3	S2	P1	J1	200
4	S2	P1	J2	500
5	S2	P2	J2	500
6	S3	P2	J2	500
7	S3	P3	J1	200
8	S1	P3	J2	100
9	S3	P3	J2	200

c.

	1	2	3	4
	S#	P#	J#	QTY
1	S3	P1	J1	100
2	S1	P1	J1	200
3	S2	P1	J1	200
4	S3	P3	J1	200
5	S1	P3	J2	100
6	S3	P3	J2	200
7	S2	P1	J2	500
8	S2	P2	J2	500
9	S3	P2	J2	500

d.

	1	2	3	4
	S#	P#	J#	QTY
1	S1	P3	J2	100
2	S3	P1	J1	100
3	S1	P1	J1	200
4	S2	P1	J1	200
5	S3	P3	J1	200
6	S3	P3	J2	200
7	S2	P1	J2	500
8	S2	P2	J2	500
9	S3	P2	J2	500

Fig.7.5: Reconstructed files corresponding to the preferred Record Reconstruction Table of Fig. 7.4

Of course, Fig. 7.5 effectively shows four different versions of the original shipments file of Fig. 7.2. What I want to draw your attention to here, though, is just which versions they are. To be specific, note that the four versions represent, respectively, the results of the following four SQL queries (I've deliberately shown the attribute names in the various ORDER BY specifications in **bold**):

```
a. SELECT S#,P#,J#,QTY
   FROM SPJ
   ORDER BY S#,P#,J#,QTY;

b. SELECT S#,P#,J#,QTY
   FROM SPJ
   ORDER BY P#, J#, QTY, S# ;

c. SELECT S#, P#, J#, QTY
   FROM SPJ
   ORDER BY J#, QTY, S#, P#;

d. SELECT S#,P#,J#,QTY
   FROM SPJ
   ORDER BY QTY, S#, P#, J#;
```

In other words, the "preferred" Record Reconstruction Table of Fig. 7.4 doesn't just reflect four single-attribute orderings simultaneously—it actually reflects four major-to-minor orderings simultaneously. That is, column P# (for example) corresponds not just to a simple ORDER BY of the form ORDER BY P#, but rather to an ORDER BY of the form ORDER BY P#, J#, QTY, S#—and similarly for the other three columns.

Observe next that, since column P# of the preferred Record Reconstruction Table does reflect the major-to-minor ordering on P#-J#-QTY-S#, it also reflects, a fortiori, the major-to-minor ordering on P#-J#-QTY, the major-to-minor ordering on P#-J#, and the "major-to-minor" ordering on P# as well. Again, analogous remarks apply to the other three columns. *Note:* In the particular case of column S#, however, I should point out that there's no real difference between the specifications ORDER BY S#, P#, J#, QTY and ORDER BY S#, P#, J# (that is, the QTY specification is irrelevant in this particular example), because the combination {S#,P#,J#} is a key.

What's more, if we process our preferred Record Reconstruction Table by starting with (say) the S# column but processing it in reverse order (that is, from bottom to top), then it should be clear that we will obtain a result identical to part a. of Fig. 7.5, except that the rows will be in reverse sequence. In other words, we will have implemented the SQL query

```
SELECT S#, P#, J#, QTY
FROM SPJ
ORDER BY S# DESC, P# DESC, J# DESC, QTY DESC ;
```

So the preferred Record Reconstruction Table actually reflects an equal number of reverse major-to-minor orderings, too.

Note finally that we effectively get all of this extra functionality "for free"[2]—we have to have some Record Reconstruction Table, and it might just as well be a preferred one. And the foregoing discussion should be sufficient to explain why we regard such a Record Reconstruction Table as "preferred" in the first place.

7.4 Building a Preferred Record Reconstruction Table

In any given Record Reconstruction Table, each column effectively reflects a certain sort order that's associated with that column. Let me immediately explain this remark:

- First, the sort order "associated with" a given column is, in general, a major-to-minor ordering in which the corresponding attribute of the user relation is the major attribute.

- Second, when I say a given column "reflects" some particular sort order, I mean that if we process the Record Reconstruction Table in sequential order according to that column, then we will reconstruct the corresponding file in that specific sort order.

To obtain a preferred Record Reconstruction Table, therefore, it's necessary to associate the particular sort order we want with each individual column. How? Well, let's do it; let's build the specific Record Reconstruction Table used in the previous section. I'll use the technique described in Chapter 5, Section 5.4 (the one involving the Inverse Permutation Table).

We start by considering, for each field in turn of the shipments file, the sort order we do in fact want. For the S# field, that sort order is the one produced by the ORDER BY specification

```
ORDER BY S#, P#, J#, QTY
```

Take another look at the file in Fig. 7.2. It turns out, as it happens, that the file has been shown in that figure in exactly this sort order, so the "S# permutation" we want—see Chapter 4, Section 4.5, or Chapter 5, Section 5.4, if you need to refresh your memory regarding this notion—is as follows:

- S# - P# - J# - QTY : *1, 2, 3, 4, 5, 6, 7, 8, 9*

The P# permutation is the permutation that corresponds to the ORDER BY specification ORDER BY P#, J#, QTY, S#:

- P# - J# - QTY - S# : *6, 1, 3, 4, 5, 7, 8, 2, 9*

(you can easily check this by comparing parts a. and b. of Fig. 7.5). And the other two permutations are:

- J# - QTY - S# - P# : *6, 1, 3, 8, 2, 9, 4, 5, 7*
- QTY - S# - P# - J# : *2, 6, 1, 3, 8, 9, 4, 5, 7*

Next, we figure out the corresponding *inverse* permutations:

- S# - P# - J# - QTY : 1, 2, 3, 4, 5, 6, 7, 8, 9
  Inverse : 1, 2, 3, 4, 5, 6, 7, 8, 9
- P# - J# - QTY - S# : 6, 1, 3, 4, 5, 7, 8, 2, 9
  Inverse : 2, 8, 3, 4, 5, 1, 6, 7, 9
- J# - QTY - S# - P# : 6, 1, 3, 8, 2, 9, 4, 5, 7
  Inverse : 2, 5, 3, 7, 8, 1, 9, 4, 6
- QTY - S# - P# - J# : 2, 6, 1, 3, 8, 9, 4, 5, 7
  Inverse : 3, 1, 4, 7, 8, 2, 9, 5, 6

Incidentally, note that the S# permutation is its own inverse.

Here then is the Inverse Permutation Table:

	1	2	3	4
	S#	P#	J#	QTY
1	1	2	2	3
2	2	8	5	1
3	3	3	3	4
4	4	4	7	7
5	5	5	8	8
6	6	1	1	2
7	7	6	9	9
8	8	7	4	5
9	9	9	6	6

To build the corresponding Record Reconstruction Table, we use the algorithm from Chapter 5 (Section 5.4):

> Go to cell [*i,1*] of the Inverse Permutation Table. Let that cell contain the value *r;* also, let the next cell to the right, cell [*i,2*], contain the value *r′*. Go to the *r*th row of the Record Reconstruction Table and place the value *r′* in cell [*r,1*].

Executing this algorithm for *i = 1, 2, ..., 9* yields the entire S# column. The other columns are built analogously. The result is the "preferred" Record Reconstruction Table shown in Fig. 7.4. **Exercise 9:** Check this claim.

7.5 Another Example

In my discussions so far, I've considered only major-to-minor orderings that correspond to the left-to-right sequence of fields in the shipments file as shown in Fig. 7.2, together with cyclic shifts of that sequence, such as J#-QTY-S#-P#. Don't assume that such always has to be the case, however; the only hard requirement is that the sort order associated with a given column must be one for which the corresponding attribute of the user relation serves as the major attribute. By way of example, notice that the "preferred" Record Reconstruction Table discussed so far doesn't support either of the potentially useful sort orders P#-J#-S# and J#-S#-P#. (The minor attribute QTY can be ignored in both of these examples, of course, since {S#,P#,J#} is a key.) But there's no reason why we shouldn't build a Record Reconstruction Table that does support those sort orders, or indeed any others we might desire. To be specific, suppose the sort orders we want are as follows:

- For column S# : `S# - P# - J#`
- For column P# : `P# - J# - S#`
- For column J# : `J# - S# - P#`
- For column QTY : `QTY - S# - P# - J#`

I'll leave it as an exercise for you—this is **Exercise 10**—to determine that the corresponding "preferred" Record Reconstruction Table is as shown in Fig. 7.6, and that it does indeed exhibit the desired behavior.

	1	2	3	4
	S#	P#	J#	QTY
1	1	1	3	2
2	8	2	4	6
3	2	3	2	1
4	4	6	5	3
5	5	7	1	8
6	3	8	7	9
7	6	4	8	4
8	7	5	9	5
9	9	9	6	7

Fig.7.6: Another preferred Record Reconstruction Table for the file of Fig. 7.2

Actually, there's a sense in which the "preferred" Record Reconstruction Table of Fig. 7.4 might be "more preferred" than that of Fig. 7.6. Let's agree to say that the table of Fig. 7.4 is *cyclic,* since (unlike the table of Fig. 7.6) it corresponds to all possible cyclic shifts of a certain sequence of the pertinent attributes. Then it turns out that, within such a cyclic table, the pointers that correspond to a given field value within the Field Values Table are guaranteed to be *in sorted order*. For example, consider the J# value J2. The pointers corresponding to that value in the cyclic Record Reconstruction Table of Fig. 7.4 are as follows:

```
1, 6, 7, 8, 9
```

By contrast, the pointers corresponding to that same value in the Record Reconstruction Table of Fig. 7.6 are as follows:

```
1, 7, 8, 9, 6
```

Thanks to this property of the cyclic table, certain additional efficiencies become possible in implementation; for example, certain compression techniques can be applied, and binary searches can be used, on (portions of) columns within such tables. In particular, queries involving the aggregate operators MAX and MIN can now be very efficient, as we'll see in Chapter 10. Further details are beyond the scope of this book.

7.6 Analysis

How many possible major-to-minor orderings are there for a given file? In the case of shipments, there are four fields, and hence 4! = 24 "complete" orderings if we consider ascending sequence only, or 2^4 * 4! = 16 * 24 = 384 such orderings if descending sequence is taken into account as well. (By the term "complete ordering" here, I mean the applicable ORDER BY specifies all four attributes.) And each of the Record Reconstruction Tables discussed in this chapter so far supports just four of those orderings, or eight if we include reverse orderings too. On the face of it, therefore, it looks as if we'd need six different Record Reconstruction Tables for the shipments file to support all possible "complete" major-to-minor orderings, or *forty-eight* if we wanted to take descending sequence into account as well.

In practice, of course, the prospect is usually not nearly so bleak. Here are some reasons why not:

- First of all, many orderings that are logically possible are simply not interesting. Some attributes might never participate in an ORDER BY specification at all (I gave the example near the end of the previous chapter of an attribute whose values are text strings representing natural-language comments). Other attributes might never participate in the major position (QTY might be an example here, in the case of relation SPJ).

- Second, ORDER BY specifications that involve all of the attributes of a user-level relation are quite rare in practice. And I've already pointed out that if the Record Reconstruction Table supports, say, a major-to-minor ordering on attributes *A*-*B*-*C*-*D* (in that sequence), then it implicitly supports the major-to-minor orderings on attributes *A*-*B*-*C* (in that sequence), on *A*-*B* (in that sequence), and on *A*.

- Third, in any ORDER BY specification that includes all of the attributes of some key, any attributes to the right of the rightmost of those key attributes within that specification can simply be ignored. Thus, for example, *any* Record Reconstruction Table for suppliers (not shipments) will certainly support ordering on the sole key {S#}, and will therefore automatically support all six of the following major-to-minor orderings:

  ```
  S# - SNAME — STATUS - CITY
  S# - SNAME — CITY - STATUS
  S# - STATUS — CITY - SNAME
  S# - STATUS — SNAME - CITY
  S# - CITY — SNAME - STATUS
  S# - CITY — STATUS - SNAME
  ```

- Fourth, it's easy to guess in practice which particular orderings are going to be useful.

- Last, it's easy to build additional Record Reconstruction Tables if they're needed.

Endnotes

1. As you might know, both this relation and the original suppliers relation are based on a running example used extensively in reference [32] and other database writings of mine.
2. Well, not entirely for free; if we're not careful, updates to the shipments relation could imply updates to the Record Reconstruction Table that could in turn cause that table to lose the desirable properties we're talking about. So the implementation has to make sure that such effects don't occur. Details of what's involved in this process are beyond the scope of this book; suffice it to say that the problem can be and has been solved (and implemented), and the solution involves comparatively little performance overhead.

8 Condensed Columns

8.1 Introduction

In this chapter, I want to look at another extremely important refinement to the basic TR model, **condensed columns**. Condensed columns can be thought of as a highly TR-specific approach to data compression (see Chapter 2)—though there's much more to the concept than that, as we'll soon see. Unlike the refinement discussed in the previous chapter, which affected the Record Reconstruction Table, condensed columns affect the Field Values Table (and possibly the Record Reconstruction Table as well), as we'll also soon see.

I'll use a new example to illustrate the basic idea. Consider the parts relation P depicted in Fig. 8.1.[1] Note that each part has a part number (P#), unique to that part; a part name (PNAME), not necessarily unique; a color (COLOR); a weight (WEIGHT); and a location (CITY). The sole key is {P#}. For definiteness, let's assume that attributes P#, PNAME, COLOR, WEIGHT, and CITY are defined over types P#, NAME, CHAR, NUMERIC, and CHAR again, respectively, where P# and NAME are user-defined types and CHAR and NUMERIC are system-defined types.

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

Fig. 8.1: The parts relation P

Fig. 8.2 shows a possible file corresponding to the relation of Fig. 8.1; Fig. 8.3 shows the corresponding Field Values Table; and Fig. 8.4 shows a corresponding Record Reconstruction Table, based on the following permutations:

- P# — PNAME — COLOR — WEIGHT — CITY : *1, 2, 3, 4, 5, 6*
- PNAME — COLOR — WEIGHT — CITY — P# : *2, 5, 6, 1, 3, 4*
- COLOR — WEIGHT — CITY — P# — PNAME : *5, 3, 2, 1, 4, 6*
- WEIGHT — CITY - P# — PNAME — COLOR : *1, 5, 4, 3, 2, 6*
- CITY — P# — PNAME — COLOR — WEIGHT : *1, 4, 6, 3, 2, 5*

As an aside, let me remind you that any attribute appearing to the right of attribute P# in any of the foregoing attribute lists can safely be ignored, because {P#} is a key (see Chapter 7, Section 7.6).

	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	P1	Nut	Red	12.0	London
2	P2	Bolt	Green	17.0	Paris
3	P3	Screw	Blue	17.0	Oslo
4	P4	Screw	Red	14.0	London
5	P5	Cam	Blue	12.0	Paris
6	P6	Cog	Red	19.0	London

Fig. 8.2: File corresponding to the parts relation of Fig. 8.1

	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	P1	Bolt	Blue	12.0	London
2	P2	Cam	Blue	12.0	London
3	P3	Cog	Green	14.0	London
4	P4	Nut	Red	17.0	Oslo
5	P5	Screw	Red	17.0	Paris
6	P6	Screw	Red	19.0	Paris

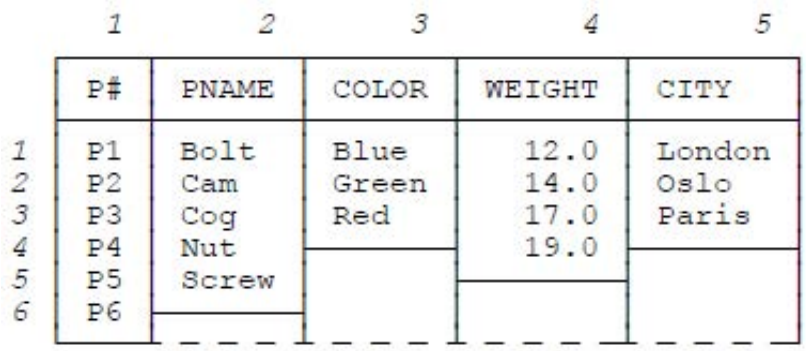
Fig. 8.3: Field Values Table corresponding to the file of Fig. 8.2

	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	4	3	2	1	1
2	1	1	4	6	4
3	5	6	5	2	6
4	6	4	1	4	3
5	2	2	3	5	2
6	3	5	6	3	5

Fig. 8.4: Record Reconstruction Table corresponding to the file of Fig. 8.2

8.2 Condensing the Field Values Table

Observe now that the Field Values Table of Fig. 8.3 involves a considerable amount of **redundancy**—for example, the city name London appears three times, the weight 17.0 appears twice, and so on. "Condensing" the columns of that table simply eliminates that redundancy. The result is thus a table in which each column contains just the pertinent *distinct* values, as shown in Fig. 8.5.



	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	P1	Bolt	Blue	12.0	London
2	P2	Cam	Green	14.0	Oslo
3	P3	Cog	Red	17.0	Paris
4	P4	Nut		19.0	
5	P5	Screw			
6	P6				

Fig. 8.5: Condensed version of the Field Values Table of Fig. 8.3

Numerous points arise immediately from this simple idea of condensing columns. Here are some of them:

- The condensed table is no longer really a table as such—I mean it isn't just a simple two-dimensional array any more—because certain cells are missing; for example, there's no [*5,5*] cell. Internally, therefore, the condensed table will probably be implemented as a set of *vectors* or *chained lists,* one such for each column, not as a two-dimensional array (you might recall that I mentioned this point before, in Chapter 6, when I was discussing INSERT operations, but now we see another good reason for adopting such an implementation). For pedagogic purposes, however, it's convenient to keep on referring to the condensed version of the table *as* a table (and showing it in a kind of semitabular form, as in Fig. 8.5), and so I will.

- There's no point in condensing the part number column, because part numbers are unique (meaning no part number ever appears more than once in the column anyway). What's more, there might not be much point in condensing the part name column either, if part names are "almost unique"; for the sake of the example, however, I *have* shown that column as condensed in Fig. 8.5. As you can see, therefore, it's perfectly legitimate, and indeed desirable, to apply the condensing process selectively.

- Field values in condensed columns are effectively shared across records of the parts file (I touched on this point in Chapter 6 as well). For example, the city name London in cell [*1,5*] is shared by three part records: namely, those for parts P1, P4, and P6.

- Certain relational operations, especially join, now have the potential to run faster than before (essentially because there's less data to process). *Note:* Joins are fast in TR anyway because Field Values Table columns are kept in sorted order; as I pointed out in Section 4.4, this fact means we can do a sort/merge join without having to do the run-time sort. What's more, there's an even more important reason why joins are fast in TR, which we'll get to in the next chapter.

- Update operations, especially INSERT, also have the potential to run faster than before, because they might be able to use field values that already exist (even ones that aren't "logically deleted"—see Chapter 6), effectively sharing those values with other records. For example, consider what happens if the user tries to insert a part tuple for part P7, with part name Nut, color Red, weight 18.0, city London. *Note:* The update algorithms described in Chapter 6 clearly need some revision if they're to work with a condensed version of the Field Values Table; however, it's not worth getting into details of those revisions here.

- In the introduction to this chapter, I said that condensed columns constitute a particular kind of data compression. That's true, of course, but I want to point out that it's a kind of compression not found—indeed, not really possible—in conventional approaches to relational implementation, precisely because of the direct-image nature of those conventional approaches. Indeed, the kind of compression we're talking about isn't really like *any* of the compression techniques described in Chapter 2; rather, it's compression *on an individual field-by-field basis,* and it's made possible only by the fact that field values and linkage information are kept separate in the first place. By contrast, conventional compression—compression on the data as such, that is, as opposed to compression within some index—is typically done on the basis of records, not fields (if it's done at all).

- Following on from the previous point, I'd like to emphasize just how much compression is possible with condensed columns. By way of an example, imagine a Department of Motor Vehicles relation representing drivers' licenses, with a tuple for every license issued in (say) the state of California, for a total of perhaps 20 million tuples. But there certainly aren't 20 million different heights, or weights, or hair colors, or expiry dates; in other words, the compression ratio might quite literally be of the order of a million or so to one.

- Yet another advantage of condensed columns is as follows.[2] With conventional direct-image implementations, a trick that's often used to save storage space is to represent properties by coded values in the database. For example, the property "part color" might be represented as integer values, according to the mapping 1 = Red, 2 = Blue, and so on. But:

  a) This trick implies the need for an additional user-level relation to represent the mapping;

  b) It also implies that user-level requests are more complicated, because they require additional joins.

  With condensed columns, however, the need for this coded-values trick disappears. As a consequence, time and space requirements are both reduced, and user requests are simpler to formulate as well. (What's more, if the trick is used anyway—perhaps because the database has been migrated from some legacy system—the code values still might not need to be physically stored. See Section 8.5 for further explanation of this point.)

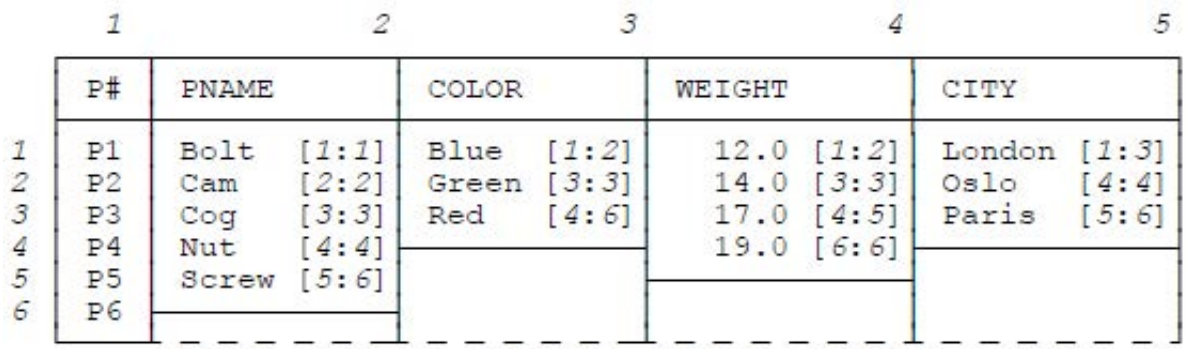
- For completeness, I should note that a column doesn't actually have to be sorted in order to be condensed—the benefits that follow from eliminating redundancy would apply even without sorting. But sorting provides so many additional benefits that it's reasonable to assume that any column that's condensed is sorted as well, and I'll make that assumption throughout what follows, barring explicit statements to the contrary. (In practice, in fact, it's hard to imagine a column being condensed but not sorted—in part because it's probably necessary to sort the column in order to do the condensing in the first place.)

- *Terminology:* From this point forward, I'll use the term "condensed Field Values Table" to mean any Field Values Table in which there's at least one column that's condensed. In fact, I'll use the term "Field Values Table," unqualified, to refer to a condensed Field Values Table specifically (in other words, I'll assume that all Field Values Tables are condensed ones, barring explicit statements to the contrary).

Row Ranges

Back to the specific example of Fig. 8.5. Of course, we can't just replace (for example) the original three appearances of the city name London by one such appearance, because we'd be losing information if we did. (The condensed CITY column contains three values, but there are six parts. How would we know which part is in which city?) So we need to keep some additional information that, in effect, allows us to reconstruct the original **un**condensed Field Values Table from its condensed counterpart. *Note:* I'm not saying we do actually want to reconstruct that uncondensed table; to do so would undermine the whole point (or a large part of the point, anyway) of condensing in the first place. I simply mean this is a way to think about the matter—if we *can* reconstruct the uncondensed table, at least in principle, then clearly no information has been lost.

One way to achieve the foregoing effect is to keep, alongside each field value in each condensed column in the Field Values Table, a specification of **the range of row numbers** for rows in the *un*condensed version of that table in which that value originally appeared, as shown in Fig. 8.6.



	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	P1	Bolt [1:1]	Blue [1:2]	12.0 [1:2]	London [1:3]
2	P2	Cam [2:2]	Green [3:3]	14.0 [3:3]	Oslo [4:4]
3	P3	Cog [3:3]	Red [4:6]	17.0 [4:5]	Paris [5:6]
4	P4	Nut [4:4]		19.0 [6:6]	
5	P5	Screw [5:6]			
6	P6				

Fig. 8.6: Condensed version of the Field Values Table of Fig. 8.3, with row ranges

To see how the row ranges work, consider (arbitrarily) cell [*3,4*] in the Field Values Table of Fig. 8.6, which contains the weight value 17.0. Alongside that weight value appears the row range "[*4:5*]." That row range means that if the Field Values Table were to be "uncondensed," as it were, then the weight value 17.0 would appear—in the WEIGHT column, of course, which is to say in column *4*—in rows *4* to *5,* inclusive, within that uncondensed table.

Incidentally, don't confuse a specification of the form [*4:5*] with one of the form [*4,5*]. The former (with a colon separator) denotes a certain range of row numbers, as just explained; the latter (with a comma separator) is a subscript that identifies a certain cell, at a certain row-and-column intersection.

Of course, the information represented by row ranges like those shown in Fig. 8.6 could be physically implemented in a variety of different ways. One way would be to move those row ranges out into a separate table of their own, isomorphic to the condensed Field Values Table. Another would be to give just the beginning or just the end of the range (I showed both in the figure for clarity, but obviously we don't need both). Another would be to replace each row range by a count of the number of times the corresponding value appears in the uncondensed table (the count would be two in the case of the weight value 17.0, for example). And so on.

There's one more point I want to make regarding row ranges. Take another look at (for example) column *3,* the COLOR column, in the Field Values Table of Fig. 8.6. Clearly, that column specifies exactly (a) the set of COLOR values that currently appear in the parts file, together with (b) for each such value, the number of times that value appears in that file. In other words, the column can be regarded as a **histogram,** as shown in Fig. 8.7. In general, in fact, the overall condensed Field Values Table, with its corresponding row ranges, can very usefully be thought of as a set of histograms, one for each condensed column. One consequence of this fact is that queries that conceptually involve such histograms are likely to perform well. By way of example, think how easy it is, given the histogram of Fig. 8.7, to answer the query "How many parts are there of each color?" I'll have more to say about such matters in Chapter 10 (especially in Section 10.5).

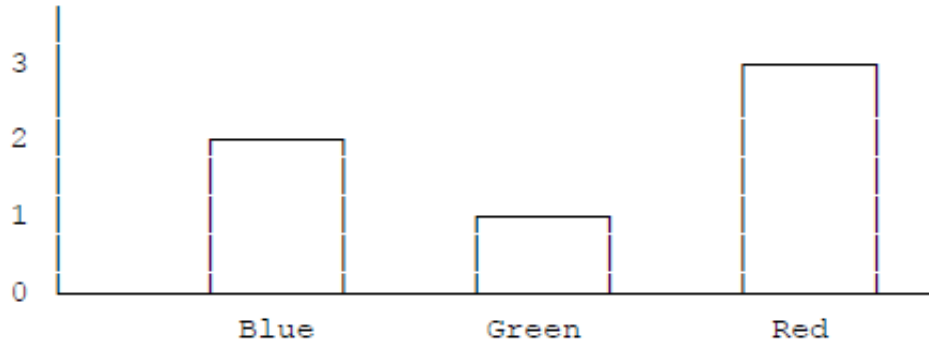


Fig. 8.7: Color histogram (based on Fig. 8.6)

To continue with the same point for a moment: If the Field Values Table can effectively be thought of as a set of histograms, then the Record Reconstruction Table—as we already know from previous chapters—can effectively be thought of as a set of **permutations**. For example, if we reconstruct the parts file using column *3* of the Record Reconstruction Table of Fig. 8.4, we'll obtain a version of the file that's ordered by part color; in other words, we get what we might call a "COLOR permutation" of that file. Thus, we can characterize the TR representation of any given set of data, informally, as *a set of histograms* plus *a set of permutations* (of the data in question). Such histograms and permutations are, in essence, what the TR representation is really all about.

8.3 Implications for Record Reconstruction

Condensing the Field Values Table clearly destroys the one-to-one relationship between cells of that table and cells of the Record Reconstruction Table. It follows that the record reconstruction algorithm we've been using up to this point (described in Chapter 4, Section 4.4) won't work any more. However, it's easy enough to fix it up, as follows:

> Consider cell [*i,j*] of the Record Reconstruction Table. Instead of going to cell [*i,j*] of the Field Values Table, we go to cell [*i',j*] of that table, where cell [*i',j*] is that unique cell within column *j* of that table that contains a row range that includes row *i*.

For example, consider cell [*3,4*] of the Record Reconstruction Table of Fig. 8.4, which appears (of course) in column *4*—the WEIGHT column—of that table. To find the corresponding weight value in the Field Values Table of Fig. 8.6, we search the WEIGHT column of that table, looking for the unique entry in that column that contains a row range that includes row *3*. From the figure, we see that the entry in question is cell [*2,4*] (the corresponding range of rows is [*3*:*3*]), and the required weight value is 14.0. **Exercise 11:** Use the Record Reconstruction Table of Fig. 8.4, together with the condensed Field Values Table of Fig. 8.6, to reconstruct the parts file in its entirety. Start with column *5* in order to obtain the result in ascending city name sequence.

However, there's a problem. With the original uncondensed Field Values Table, when we were reconstructing a given record, we could go *directly* from cell [*i,j*] of the Record Reconstruction Table to cell [*i,j*] of the Field Values Table. Now, by contrast, we have to do a *search* through column *j* of this latter table in order to find the relevant cell—the cell in question being that unique cell [*i',j*] that contains a row range that includes row *i*—and searches mean overhead. I'll fix this problem in the section immediately following.

Note: Before we get to that next section, however, I should make it clear that the amount of overhead we're talking about here is actually not all that great. The reason is that the row ranges within any given Field Values Table column are in ascending sequence (more precisely, they're in ascending sequence by either their begin points or their end points), and so the searches we have to do can at least be *binary* searches specifically. Expanding the Record Reconstruction Table in the manner to be described in the next section can thus be thought of as yet another optional extra.

8.4 Expanding the Record Reconstruction Table

The solution to the search problem identified toward the end of the previous section is essentially straightforward (indeed, you might have already figured it out for yourself): We just expand the Record Reconstruction Table such that, if column *j* of the Field Values Table is condensed, then each cell in column *j* of the Record Reconstruction Table now contains two pointers instead of one, as follows.

- One of those pointers is (as always) the row number of the next row to be inspected within the Record Reconstruction Table.

- The other is the row number *i′* of the cell [*i′,j*] of the Field Values Table that actually contains the required field value. In other words, it's that unique row number *i′* such that, if the row of the Record Reconstruction Table that we're currently looking at is row *i,* then the row range in the Field Values Table cell [*i′,j*] includes that row number *i*.

Fig. 8.8 is a revised version of Fig. 8.4, showing what happens to the Record Reconstruction Table in our example if this approach is adopted. Points to note:

- First, the P# column remains unchanged, because the P# column of the Field Values Table isn't condensed.

- Second, in those columns that do include two row numbers instead of just one, it's intuitively convenient to show those two row numbers "the wrong way round" (or what some people might think is the wrong way round, at any rate). That is, the first is the number of the desired row within the Field Values Table, while the second is the number of the next row to be inspected within the Record Reconstruction Table. The reason for this switch will, I think, become obvious if you try to use this expanded Record Reconstruction Table to reconstruct records of the parts file—which (as I'm sure you've already guessed) I'm going to ask you to do in just a moment.

	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	4	1▪3	1▪2	1▪1	1▪1
2	1	2▪1	1▪4	1▪6	1▪4
3	5	3▪6	2▪5	2▪2	1▪6
4	6	4▪4	3▪1	3▪4	2▪3
5	2	5▪2	3▪3	3▪5	3▪2
6	3	5▪5	3▪6	4▪3	3▪5

Fig. 8.8: Expanded version of the Record Reconstruction Table of Fig. 8.4

By way of example, consider cell [*2,4*], which contains the entry *1*▪*6* (note the "▪" separator):

- The *6* tells us, as usual, that the next cell to inspect in the Record Reconstruction Table is in the *sixth* row; in other words, that next Record Reconstruction Table cell is cell [*6,5*].

- By contrast, the *1* tells us that the cell in the Field Values Table that contains the field value corresponding to *this* cell [*2,4*] of the Record Reconstruction Table is in the *first* row; in other words, that Field Values Table cell is cell [*1,4*], which contains the weight value 12.0.

Exercise 12: Use the Record Reconstruction Table of Fig. 8.8, together with the condensed Field Values Table of Fig. 8.6, to reconstruct the parts file in its entirety. Again, start with column *5* to obtain the result in ascending city name sequence.

As you can see from the foregoing example, the record reconstruction process is just as fast as it was before (as fast, that is, as it was before we condensed the Field Values Table in the first place). Of course, the Record Reconstruction Table is now *bigger* than it was before ... Whether it's worth paying this price will depend on the benefit we obtain from speeding up the reconstruction process (it might be worth it in main memory but not on disk, for example).

Incidentally, notice that the row ranges in the Field Values Table aren't used or needed in the record reconstruction process, once the expanded Record Reconstruction Table has been built. However, they're still useful, and indeed important. By way of illustration, suppose cell *c* in the Field Values Table contains the row range [*i1*:*i2*]. Then there must be precisely (*i2*-*i1*)+1 cells in the Record Reconstruction Table that contain a pointer to cell *c*. For example, cell [*3,5*] of the Field Values Table contains the row range [*5*:*6*], and so it follows that there are precisely (6−5)+1 = two cells in the Record Reconstruction Table—namely, cells [*5,5*] and [*6,5*]—that include a pointer to cell [*3,5*] of the Field Values Table. In other words, the row ranges effectively tell us how many tuples in the original user relation contain a given value for a given attribute. As I mentioned at the end of Section 8.2 (when I was discussing histograms), the usefulness of this kind of information in responding to certain kinds of queries—for example, "How many parts are there in Paris?"—should be obvious. See Chapter 10 for further discussion.

Row ranges also turn out to be extremely important in connection with join operations, as we'll see in Chapters 9 and 10 (in Section 10.6 in particular).

Note: As you've probably come to expect by now, the expanded Record Reconstruction Table (like the condensed Field Values Table) can be physically implemented in many different ways. For example, the new pointers—the ones that point into the condensed Field Values Table—might be moved out into a separate table of their own, isomorphic to the Record Reconstruction Table (at least, isomorphic to those columns of that table that correspond to condensed columns in the Field Values Table). Other physical implementations are also possible (see reference [63] for more specifics).

A final point: Since I said in Section 8.2 that from this point forward I'm going to take the unqualified term "Field Values Table" to mean, specifically, a condensed version of that table, it makes sense to take the unqualified term "Record Reconstruction Table" to mean a correspondingly *expanded* version of *that* table, and so I will (barring explicit statements to the contrary in both cases, of course).

8.5 Further Space-Saving Techniques

We've seen that (among other things) condensed columns are a technique for saving storage space. In this final section of the chapter, I want to take a quick look at a few other space-saving techniques that can be applied in the context of the TR model. Although the techniques in question have little or nothing to do with condensed columns as such, I think this chapter is the best place to cover them nonetheless.

The basic point is that some kinds of information can be represented just as well (if not better) implicitly instead of explicitly. For example, suppose some user relation *R* has an attribute *A* whose values are precisely the integers from 1 to *M,* where *M* is the number of tuples. Let *F* be a file corresponding to *R,* with fields having the same names as the attributes of *R*. Then, no matter which (arbitrary) record ordering we choose for *F*—that is, no matter in what order the integers in field *A* actually appear in file *F*—column *A* of the Field Values Table will necessarily contain the integers 1 to *M* in sorted order. In other words, every *A* value in that table will be identical to the row number of the row that contains it, and there's therefore no point in having a column for *A* in the Field Values Table at all. *Note:* This particular idea might be useful in connection with system-generated key values [40].

Here are a few more examples of situations—certain aspects of which have already been touched on in passing—in which some space saving might possibly be realized:

- Let *A* be a field whose *i*th value (where *i* is the position of the value in question within the corresponding column of the Field Values Table) is computed as some function *f*(*i*) of *i*. Suppose further that the function *f* is such that, whenever *i1* < *i2,* then *f*(*i1*) < *f*(*i2*). (A simple example of such a function is "multiply *i* by *k,*" where *k* is some positive constant.) Then, again, field *A* needs no Field Values Table column at all. *Note:* The "integers 1 to *M*" example discussed above is a special case of this possibility (the function *f* in that example is the identity function, of course).

- Let *A* be a field whose values are all distinct. Assuming that field *A* does have a column in the Field Values Table (that is, we're not dealing with one of the cases already discussed above), then at least that Field Values Table column needs no associated row ranges (as we've already seen in the case of, for example, column P# in Fig. 8.6).

- Again let *A* be a field whose values are all distinct. If the values of *A* are sorted into order, it will often be the case that the result consists of a series of *runs* or *sequences* with no gaps in them, separated by gaps of arbitrary size. Here's a simple example:

  ```
  1,2,3,4,5    9,10,11,12    16,17,18,19,20,21,22,23,24    35,36,37,38
  ```

  In such a situation, it might well be better if the relevant column of the Field Values Table contains just *range* information, as here:

  ```
  [1:5] [9:12] [16:24] [35:38]
  ```

  (Please understand that the ranges shown are ranges of *field values,* not row ranges as previously discussed.) Alternatively, since there'll be one fewer of them, we might choose to represent the *gaps* instead of the values:

  ```
  [6:8] [13:15] [25:34]
  ```

  This technique is called *straight-line encoding*.

Other space-saving possibilities are described in reference [63]. In particular, a variety of more conventional compression techniques can be applied to the Field Values Table or the Record Reconstruction Table or both. I'd just like to mention one example of such compression here; it applies primarily to the Field Values Table.[3] The basic point is that, since the left-to-right column order within that table has no significance at the user level, those columns can appear in any order internally. In particular, they can be rearranged in such a way as to make the best use of boundary alignment requirements (if any) at the physical storage level. For example, suppose the Field Values Table has eight columns named *A, B, C, D, E, F, G, H;* suppose further that columns *B, D, F,* and *H* each have a column width of one word (four bytes) and require word alignment, while columns *A, C, E,* and *G* each have a column width of one byte and require only byte alignment. Then storing the table in left-to-right column order *A, B, C, D, E, F, G, H* would mean that each row occupies a total of 32 bytes, while storing it in left-to-right column order *A, C, E, G, B, D, F, H* would mean that each row occupies only 20 bytes (a 37.5 percent reduction).

Endnotes

1. This relation is taken from the same running example as the suppliers and shipments relations in previous chapters (see reference [32] and elsewhere)—except that, for the sake of an example in Chapter 10, I've taken the liberty of moving part P3 from Rome to Oslo.
2. Thanks to Tom Sawyer for pointing this one out.
3. On the other hand, it does tacitly assume that the table is stored row-wise, which we saw in Chapter 6 is probably not the case. It might perhaps make sense when reading the Field Values Table off the disk into memory.

9 Merged Columns

9.1 Introduction

Now I want to turn to yet another very important refinement to the basic TR model, **merged columns**. In the previous chapter, I discussed condensed columns, which can be characterized as a way of sharing field values across records—but the records in question all had to come from the same file. Merged columns, by contrast, can be characterized as a way of sharing field values across records, where the records in question might or might not all come from the same file.[1] I'll consider two examples, the first involving just one file, the second involving two.

Note: Columns can be merged without necessarily having to be either sorted or condensed, but the general idea of merged columns makes much more sense if the columns in question are both. In what follows, I'll assume that merged columns are indeed always both, barring explicit statements to the contrary. In practice, in fact, it's hard to imagine a column being merged without being both sorted and condensed as well.

9.2 The Bill-of-Materials Example

Essentially, the basic idea underlying merged columns is that distinct fields at the file level can map to the same Field Values Table column at the TR level (just so long as the fields in question are of the same data type, of course). For example, consider the bill-of-materials relation MMQ depicted in Fig. 9.1. That relation is meant to be interpreted as follows: The indicated "major" part (MAJOR_P#) includes the indicated "minor" part (MINOR_P#) in the indicated quantity (QTY); that is, the minor part is a component of the major part, and it takes the specified quantity of the minor part to make the major part. For example, it takes four P6's (among other things) to make one P3. The attribute combination {MAJOR_P#,MINOR_P#} is the sole key; attributes MAJOR_P# and MINOR_P# are both defined on type P#, and attribute QTY is defined on type INTEGER.

MAJOR_P#	MINOR_P#	QTY
P1	P2	2
P1	P3	4
P1	P4	1
P2	P3	3
P2	P4	8
P2	P5	6
P3	P4	3
P3	P6	4
P5	P6	3

Fig. 9.1: The bill-of-materials relation MMQ

In what follows, I'll first consider what happens in this example without merged columns, and then take a look at how the situation changes if we apply the merged-column refinement. Fig. 9.2, then, shows a possible file corresponding to the relation of Fig. 9.1. Note that I've deliberately shuffled the record ordering around, purely to make later parts of the discussion a little more interesting. (If we stick to the "obvious" ordering as suggested by Fig. 9.1, it turns out that too many coincidences occur in, for example, the Record Reconstruction Table, coincidences that suggest the existence of certain intrinsic relationships that don't in fact exist.) Fig. 9.3 shows the corresponding *uncondensed* Field Values Table, and Fig. 9.4 shows a corresponding Record Reconstruction Table, based on the following permutations:

- MAJOR_P# - MINOR_P# - QTY : `7, 2, 4, 9, 3, 5, 1, 6, 8`
- MINOR_P# - MAJOR_P# - QTY : `7, 2, 9, 4, 3, 1, 5, 6, 8`
- QTY - MAJOR_P# - MINOR_P# : `4, 7, 9, 1, 8, 2, 6, 5, 3`

Note: In the first two permutations, attribute QTY is irrelevant, because the two leading attributes constitute a key. In the third permutation, the choice of MAJOR_P# - then - MINOR_P# over MINOR_P# - then - MAJOR_P# is arbitrary on my part. **Exercise 13:** Confirm for yourself that Figs. 9.3 and 9.4 are correct.

	1	2	3
	MAJOR_P#	MINOR_P#	QTY
1	P3	P4	3
2	P1	P3	4
3	P2	P4	8
4	P1	P4	1
5	P2	P5	6
6	P3	P6	4
7	P1	P2	2
8	P5	P6	3
9	P2	P3	3

Fig. 9.2: File corresponding to the bill-of-materials relation of Fig. 9.1

	1	2	3
	MAJOR_P#	MINOR_P#	QTY
1	P1	P2	1
2	P1	P3	2
3	P1	P3	3
4	P2	P4	3
5	P2	P4	3
6	P2	P4	4
7	P3	P5	4
8	P3	P6	6
9	P5	P6	8

Fig. 9.3: Uncondensed Field Values Table corresponding to the file of Fig. 9.2

	1	2	3
	MAJOR_P#	MINOR_P#	QTY
1	1	2	3
2	2	6	1
3	4	3	4
4	3	1	7
5	5	9	9
6	7	4	2
7	6	8	8
8	8	7	6
9	9	5	5

Fig. 9.4: Record Reconstruction Table corresponding to the file of Fig. 9.2

Fig. 9.5 now shows a condensed version of the Field Values Table from Fig. 9.3, and Fig. 9.6 shows a corresponding expanded Record Reconstruction Table. Again, I recommend strongly that you confirm for yourself that these tables are correct (**Exercise 14**). Perhaps I should remind you that:

	1	2	3
	MAJOR_P#	MINOR_P#	QTY
1	P1 [1:3]	P2 [1:1]	1 [1:1]
2	P2 [4:6]	P3 [2:3]	2 [2:2]
3	P3 [7:8]	P4 [4:6]	3 [3:5]
4	P5 [9:9]	P5 [7:7]	4 [6:7]
5		P6 [8:9]	6 [8:8]
6			8 [9:9]

Fig. 9.5: Condensed version of the Field Values Table from Fig. 9.3

	1	2	3
	MAJOR_P#	MINOR_P#	QTY
1	1▪1	1▪2	1▪3
2	1▪2	2▪6	2▪1
3	1▪4	2▪3	3▪4
4	2▪3	3▪1	3▪7
5	2▪5	3▪9	3▪9
6	2▪7	3▪4	4▪2
7	3▪6	4▪8	4▪8
8	3▪8	5▪7	5▪6
9	4▪9	5▪5	6▪5

Fig. 9.6: Expanded version of the Record Reconstruction Table from Fig. 9.4

- In the case of the Field Values Table (Fig. 9.5), the numbers in brackets represent *row ranges*. For example, the row range [*4:6*] in cell [*3,2*] indicates that the corresponding field value—namely, part number P4—appears in rows *4, 5,* and *6* of the corresponding uncondensed Field Values Table (all in column *2,* of course).

- In the case of the Record Reconstruction Table (Fig. 9.6), the two numbers in each cell are both pointers (row numbers); the first refers to a row of the Field Values Table, the second refers to a row of the Record Reconstruction Table itself. For example, cell [*7,2*] contains the entry *4▪8*. The *4* means the relevant field value—namely, part number P5—is to be found in cell [*4,2*] of the Field Values Table. The *8* means the next cell to be inspected in the Record Reconstruction Table is cell [*8,3*].

Now (at last) we can start our examination of merged columns. Going right back to the user-level relation MMQ, it's clear that attributes MAJOR_P# and MINOR_P# are of the same data type (they're both of type P#, in fact), and hence that fields MAJOR_P# and MINOR_P# of the corresponding file are of the same data type, too. They can therefore be mapped to the same column of the Field Values Table. Fig. 9.7 shows what happens. Note the following points:

	1	2
	MAJOR_P# + MINOR_P#	QTY
1	P1 [1:3] [:]	1 [1:1]
2	P2 [4:6] [1:1]	2 [2:2]
3	P3 [7:8] [2:3]	3 [3:5]
4	P4 [:] [4:6]	4 [6:7]
3	P5 [9:9] [7:7]	6 [8:8]
4	P6 [:] [8:9]	8 [9:9]

Fig. 9.7: Field Values Table of Fig. 9.5 after merging the first two columns

- Columns MAJOR_P# and MINOR_P# have been merged into a single column. In the figure, I've labeled the resulting column, not very elegantly, "MAJOR_P# + MINOR_P#."

- The merged column contains all of the field values—part numbers, to be specific—that previously appeared in either column MAJOR_P# or column MINOR_P# in the table before merging. Duplicates have been eliminated.[2]

- Each cell in the merged column thus contains a single part number, together with *two* row ranges: The first indicates which rows of the uncondensed Field Values Table (see Fig. 9.3) the corresponding part number appears in as a major part number; the second indicates which rows of that uncondensed Field Values Table the corresponding part number appears in as a minor part number.

- Note that those row ranges are basically the same as they were in the previous version of the Field Values Table, except for occasional appearances of the special **empty** row range "[:]." The empty range is used when the indicated part number doesn't appear at all in the corresponding column of the uncondensed Field Values Table; for example, P1 never appears as a minor part number.

  I remark in passing, without going into details, that certain further refinements can usefully be applied to the Field Values Table if empty ranges are either particularly common or particularly rare. The refinements in question have the effect of saving storage space and speeding up searches (in the "common" case), or simplifying the task of finding the entries with empty ranges (in the "rare" case). For more details, see reference [63].

- In the merged table, the merged column is column *1* and the QTY column is column *2* (after all, the table does now have just two columns, not three). Column *2,* the QTY column, is the same as it was in Fig. 9.5. *Note:* From this point forward, I'll use the term "merged table" to mean any Field Values Table that includes at least one merged column.

Fig. 9.8 shows the corresponding Record Reconstruction Table. Note the following points:

	1	2	3
	MAJOR_P#	MINOR_P#	QTY
1	1•1	2•2	1•3
2	1•2	3•6	2•1
3	1•4	3•3	3•4
4	2•3	4•1	3•7
5	2•5	4•9	3•9
6	2•7	4•4	4•2
7	3•6	5•8	4•8
8	3•8	6•7	5•6
9	5•9	6•5	6•5

Fig. 9.8: Expanded Record Reconstruction Table corresponding to the merged Field Values Table of Fig. 9.7

- The Record Reconstruction Table still has three columns. However, columns *1* and *2* of that table now both correspond to column *1* (the merged column) of the Field Values Table; column *1* refers to the first row range in that merged column and column *2* to the second. Column *3* of the Record Reconstruction Table now refers to column *2* of the Field Values Table. These facts will obviously have to be taken into account when doing record or file reconstruction using the Record Reconstruction Table (see below).

- The algorithm for building the Record Reconstruction Table remains essentially unchanged. However, the table itself that results from executing that algorithm is *not* unchanged. To be specific, if you compare Figs. 9.8 and 9.6, you'll see that the last entry in column *1* and all of the entries in column *2* have changed, in that the first of the two row numbers—the one that refers to the Field Values Table—has increased by one in every case. This change is a result of the appearance of the aforementioned empty ranges in the merged Field Values Table.

Another strong recommendation (again you've probably already guessed this one): Try using the Record Reconstruction Table of Fig. 9.8, together with the Field Values Table of Fig. 9.7, to reconstruct a corresponding file. If you work down column *1* of the Record Reconstruction Table, you should wind up with a file that's a direct image of relation MMQ as shown in Fig. 9.1 (this is **Exercise 15**).

I'll finish up this section with a brief discussion of certain significant implications of the merged-columns idea. First, it obviously saves space. Suppose for the sake of the example that part numbers and quantities require four bytes each, while row numbers require two bytes each. Suppose too, realistically enough, that each row range is represented by a begin point only.[3] Then the unmerged Field Values Table of Fig. 9.5 would occupy a total of 90 bytes, while that of Fig. 9.7 would occupy a total of 78 bytes—a 13.3 percent reduction.

The next point is much more important. It has to do with join operations. Suppose we want to join relation MMQ to itself, matching minor part numbers in "the first copy" (as it were) of the relation with major part numbers in "the second copy." Such a join is very likely in practice, by the way; it's needed, for example, in computing the result of the well-known *part explosion* query "Get all components, at all levels, of some given part." Well, we can tell *in a single pass* over the merged

Field Values Table just which tuples join to which! For example, row *3* of that table (which contains the part number P3) shows a minor part number row range of [*2*:*3*] and a major one of [*7*:*8*]. It follows immediately that the second and third tuples in "the first copy" of relation MMQ both join to both the seventh and eighth tuples in "the second copy." And, of course, similar remarks apply to all of the other rows of that merged Field Values Table. In effect, therefore, we can do a sort/merge join without doing the sort *and without doing the merge, either!*[4]

Note: Lest I be accused of some hypocrisy, or at least inconsistency, in the way I've worded the previous paragraph, let me now try to state matters more precisely. Of course, there's no such thing as the "second" tuple, or the "third" tuple, or the "*i*th" tuple for any value of *i,* in any relation; the tuples of a relation aren't ordered. Thus, when I spoke of (for example) "the second tuple" of "the first copy" of relation MMQ, I was adopting a shorthand, and a pretty sloppy shorthand at that. What I really meant by such talk was as follows:

- Let *F1* be the reconstructed file obtained from the Field Values Table of Fig. 9.7 by processing column MAJOR_P# of the Record Reconstruction Table of Fig. 9.8 in top-to-bottom sequence. Then "the first copy" of relation MMQ is that file *F1,* and "the *i*th tuple" of that copy is that unique tuple of relation MMQ that corresponds to the *i*th record in *F1*.

- Likewise, let *F2* be the reconstructed file obtained from the Field Values Table of Fig. 9.7 by processing column MINOR_P# of the Record Reconstruction Table of Fig. 9.8 in top-to-bottom sequence. Then "the second copy" of relation MMQ is that file *F2,* and "the *i*th tuple" of that copy is that unique tuple of relation MMQ that corresponds to the *i*th record in *F2*.

The third and last point I want to mention is that merged columns can help improve update performance, especially for INSERT operations. Recall from Chapter 8 that condensed columns imply that such operations might be able to use field values that already exist, effectively sharing those values with other records. Well, the same is even more likely with merged columns, because the sharing can occur *across distinct fields*. By way of example, consider what happens if the user tries to insert an MMQ tuple with major part number P4, minor part number P6, and quantity 3.

9.3 A Foreign Key Example

For my second example, I want to return to the suppliers and shipments relations as discussed in earlier chapters. I've shown those two relations once again, side by side, in Fig. 9.9. Observe now that {S#} in the shipments relation SPJ is a **foreign key,** referencing the candidate key {S#} of the suppliers relation S (meaning that every value of {S#} in SPJ appears as a value of {S#} in S). Here's a slightly simplified definition of the concept:

- A *foreign key* is a subset of the attributes of some relation *R2* whose values are required to appear as values of some subset of the attributes of some relation *R1* (*R1* and *R2* not necessarily distinct). The attribute subset in question in relation *R1* must constitute a *candidate key* for that relation *R1*.

As I'm sure you know, joins over a foreign key and its corresponding candidate key are needed very frequently in relational systems.

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

S#	P#	J#	QTY
S1	P1	J1	200
S1	P3	J2	100
S2	P1	J1	200
S2	P1	J2	500
S2	P2	J2	500
S3	P1	J1	100
S3	P2	J2	500
S3	P3	J1	200
S3	P3	J2	200

Fig. 9.9: The suppliers and shipments relations S and SPJ

Let's assume the relations of Fig. 9.9 are mapped to files with field and record orderings that directly reflect those suggested by that figure. Then Figs. 9.10 and 9.11 show the corresponding Field Values Table and Record Reconstruction Table for suppliers, and Figs. 9.12 and 9.13 do the same for shipments. (**Exercise 16:** As usual, I recommend you check all of these tables carefully.) Note that I haven't condensed the supplier numbers column in Fig. 9.10, because each supplier is guaranteed to have a unique supplier number. By contrast, I *have* condensed the supplier names column, albeit to little effect.

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	S1	Adams [1:1]	10 [1:1]	Athens [1:1]
2	S2	Blake [2:2]	20 [2:3]	London [2:3]
3	S3	Clark [3:3]	30 [4:5]	Paris [4:5]
4	S4	Jones [4:4]		
5	S5	Smith [5:5]		

Fig. 9.10: Field Values Table for suppliers

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	5	1▪5	1▪4	1▪5
2	4	2▪4	2▪2	2▪1
3	2	3▪3	2▪3	2▪4
4	3	4▪1	3▪5	3▪2
5	1	5▪2	3▪1	3▪3

Fig. 9.11: Record Reconstruction Table for suppliers

	1	2	3	4
	S#	P#	J#	QTY
1	S1 [1:2]	P1 [1:4]	J1 [1:4]	100 [1:2]
2	S2 [3:5]	P2 [5:6]	J2 [5:9]	200 [3:6]
3	S3 [6:9]	P3 [7:9]		500 [7:9]

Fig. 9.12: Field Values Table for shipments

	1	2	3	4
	S#	P#	J#	QTY
1	1▪2	1▪1	1▪2	1▪2
2	1▪8	1▪2	1▪3	1▪6
3	2▪3	1▪3	1▪4	2▪1
4	2▪4	1▪7	1▪5	2▪3
5	2▪5	2▪8	2▪1	2▪8
6	3▪1	2▪9	2▪6	2▪9
7	3▪6	3▪4	2▪7	3▪4
8	3▪7	3▪5	2▪8	3▪5
9	3▪9	3▪6	2▪9	3▪7

Fig. 9.13: Record Reconstruction Table for shipments

Now let's combine the Field Values Tables of Figs. 9.10 and 9.12, merging the two supplier number columns together (the merging is clearly legitimate, because a foreign key and its corresponding candidate key must necessarily be of the same data type).[5] Refer to Fig. 9.14. Note the following points:

	1	*2*	*3*	*4*	*5*	*6*	*7*
	S#	SNAME	STATUS	CITY	P#	J#	QTY
1	S1[*1:2*]	Adams[*1:1*]	10[*1:1*]	Athens[*1:1*]	P1[*1:4*]	J1[*1:4*]	100[*1:2*]
2	S2[*3:5*]	Blake[*2:2*]	20[*2:3*]	London[*2:3*]	P2[*5:6*]	J2[*5:9*]	200[*3:6*]
3	S3[*6:9*]	Clark[*3:3*]	30[*4:5*]	Paris [*4:5*]	P3[*7:9*]		500[*7:9*]
4	S4[:]	Jones[*4:4*]					
5	S5[:]	Smith[*5:5*]					

Fig. 9.14: Merged Field Values Table for suppliers and shipments

- The merged table has seven columns, not eight. Column *1* is the merged column.[6] Columns *2-4* correspond to columns *2-4* of the suppliers Field Values Table; columns 5-7 correspond to columns *2-4* of the shipments Field Values Table. These facts will have to be taken into account when doing record or file reconstruction for shipments, but have no analogous implications for suppliers.

- The row ranges shown in column *1* indicate which rows of the uncondensed Field Values Table for *shipments* the corresponding supplier number appears in—we obviously don't need any analogous row ranges for *suppliers* (why not?). Note that the supplier numbers S4 and S5 don't appear in the shipments relation at all, and therefore don't appear in the shipments Field Values Table either (hence the empty row ranges for those suppliers in the *merged* table of Fig. 9.14).

- The corresponding Record Reconstruction Tables remain unchanged and are as shown in Figs. 9.11 and 9.13, respectively—with the trivial exception that, strictly speaking, we ought to replace the column numbers *2, 3, 4* for the Record Reconstruction Table for shipments by the column numbers *5, 6, 7,* respectively.

It should be clear that the advantages of merging columns in this example are analogous to those that applied in the bill-of-materials example in the previous section. In particular, joining suppliers and shipments on supplier numbers—which is a foreign-key-to-corresponding-candidate-key join, of course, and thus likely to be much needed in practice—now has the potential to be extremely fast (see Chapter 10).

Let me close this section by noting that foreign-key-to-corresponding-candidate-key joins are, by definition, many-to-one joins specifically, because a given tuple in the relation with the foreign key is guaranteed to join to exactly one tuple in the relation with the corresponding candidate key. (I discount the possibility that the foreign key might "be null" in some tuple, in which case it wouldn't join to any tuple at all. See the next chapter, Section 10.11.) By contrast, the join discussed in the previous section (a join of relation MMQ with itself) was a many-to-many join. And, while this latter example involved just a single relation, it should be clear that many-to-many joins between two distinct relations can also benefit from the merged-columns idea. It should also be clear that all of the concepts discussed in this chapter so far extend to three, four, ..., or any number of relations.

9.4 Another Kind of Merging

Toward the end of the previous chapter, I pointed out that column condensing was, among other things, a technique for saving storage space, and I took a brief look at certain other space-saving techniques that could be applied in the context of the TR model. Well, column merging too can be regarded among other things as a technique for saving storage space, and in the present section I'd like to take a quick look at a different kind of column merging that might also be used to save space.

The basic idea is that two distinct fields from the same file might map to a single combined column in the Field Values Table, even if they're of different data types. For example, consider the suppliers relation of Fig. 9.9 once again. Assume as before that the relation maps to a file with field and record orderings that directly reflect those suggested by that figure. Then, instead of mapping each field of that file to a Field Values Table column of its own as in Fig. 9.10, it would be possible to map—for example—the STATUS and CITY fields to a combined column, as shown in Fig. 9.15. Note the revised row ranges in particular.

	1	2	3
	S#	SNAME	STATUS / CITY
1	S1	Adams [1:1]	10 / Paris [1:1]
2	S2	Blake [2:2]	20 / London [2:3]
3	S3	Clark [3:3]	30 / Athens [4:4]
4	S4	Jones [4:4]	30 / Paris [5:5]
5	S5	Smith [5:5]	

Fig. 9.15: Field Values Table for suppliers with a combined STATUS / CITY column

Now, in this particular example, combining the STATUS and CITY columns in the Field Values Table as suggested in the figure probably doesn't save much space—at least, not in the Field Values Table, though it will certainly (and in fact more significantly) save space in the Record Reconstruction Table. But if there aren't very many distinct status values, or distinct city names, or (perhaps most important) distinct status-value / city-name combinations, then combining the columns has the potential to reduce space requirements significantly in the Field Values Table, too. However, there's a downside. Consider the problem of searching the Field Values Table for a particular status value or a particular city name. In the case of the status value, the search is no more difficult (and probably no more time-consuming) with the combined column than it was without it, because the combined column is still in status value order. But in the case of the city name, the search certainly is more difficult; in effect, separate searches will have to be done for each possible combination of a status value with the city name in question.

9.5 Concluding Remarks

In earlier chapters (Chapter 4 in particular), I noted that the TR model takes the concept of *data independence* much further than earlier systems did. Indeed, there's really no single thing, or combination of things, at all at the TR level that corresponds directly to a user-level tuple. From the discussions in this chapter, we can now see that there isn't necessarily any single thing or combination of things at the TR level that corresponds directly to a user-level relation, either—two or

more user-level relations might all map to the same combination of constructs at the TR level. Analogous remarks apply to user-level attributes as well.

Endnotes

1. I also pointed out in the previous chapter that column condensing can be regarded, in part, as a kind of field-level compression. The same is true of column merging also.
2. In fact, the merged column contains the set theory union of the two original condensed columns, and we could thus reasonably call it (as reference [63] in fact does) a "union column." I prefer my term because it suggests, correctly, that there's some connection between such columns and the sort/merge approach to implementing join operations, as we'll see shortly.
3. Instead of empty ranges, we would then have adjacent entries in a column of the Field Values Table with the same begin point.
4. More accurately, the sort and the merge don't have to be done at run time; instead, they're done ahead of time when the Field Values and Record Reconstruction Tables are built (basically at load time).
5. I remark in passing that attributes STATUS and QTY are of the same data type, too (they're both of type INTEGER), and so we could have merged the STATUS and QTY columns as well if we had wanted.
6. I've labeled that column just "S#", but it's really "S# values from relation S or relation SPJ or both."

10 Implementing the Relational Operators

10.1 Introduction

I've now completed my tutorial overview of the basic TR model; I've described the core concepts (principally the Field Values Table and the Record Reconstruction Table) in Chapters 4 and 5, and some very important refinements to those concepts (major-to-minor orderings, condensed columns, and merged columns) in Chapters 7, 8, and 9, respectively. I've also given some idea in Chapter 6 as to what's involved in implementing the INSERT, DELETE, and UPDATE operators. In the present chapter, I want to say a little more about what's involved in using the TR model to implement the relational operators restrict, project, and the rest: partly just to illustrate TR in action, as it were, and partly to reinforce my claim that TR is indeed an excellent foundation on which to implement the relational model (even without all of the additional refinements that I don't intend to discuss in this introductory book—refinements that, as I'm sure you'd expect, offer the possibility of numerous additional improvements).

Let me say immediately that I don't want to get into a lot of detail in what follows—I just want to indicate in outline how certain relational operators might be implemented in terms of the TR model, and offer some observations on the differences between TR and "prior art" in this regard. I'll base my examples on the suppliers and shipments relations shown in Fig. 10.1 (a repeat of Fig. 9.9). The corresponding Field Values Table—a *merged* table, please note—is shown in Fig. 10.2 (a repeat of Fig. 9.14); corresponding Record Reconstruction Tables are shown in Figs. 10.3 (a repeat of Fig. 9.11) and 10.4 (a repeat of Fig. 9.13, except that columns *2-4* have been renumbered as columns *5-7* in order to agree with the column numbering in Fig. 10.2). *Note:* You might want to make a copy of these figures for subsequent reference.

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

S#	P#	J#	QTY
S1	P1	J1	200
S1	P3	J2	100
S2	P1	J1	200
S2	P1	J2	500
S2	P2	J2	500
S3	P1	J1	100
S3	P2	J2	500
S3	P3	J1	200
S3	P3	J2	200

Fig. 10.1: The suppliers and shipments relations S and SPJ

	1	2	3	4	5	6	7
	S#	SNAME	STATUS	CITY	P#	J#	QTY
1	S1[1:2]	Adams[1:1]	10[1:1]	Athens[1:1]	P1[1:4]	J1[1:4]	100[1:2]
2	S2[3:5]	Blake[2:2]	20[2:3]	London[2:3]	P2[5:6]	J2[5:9]	200[3:6]
3	S3[6:9]	Clark[3:3]	30[4:5]	Paris [4:5]	P3[7:9]		500[7:9]
4	S4[:]	Jones[4:4]					
5	S5[:]	Smith[5:5]					

Fig. 10.2: Merged Field Values Table for suppliers and shipments

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	5	1•5	1•4	1•5
2	4	2•4	2•2	2•1
3	2	3•3	2•3	2•4
4	3	4•1	3•5	3•2
5	1	5•2	3•1	3•3

Fig. 10.3: Record Reconstruction Table for suppliers

	1	5	6	7
	S#	P#	J#	QTY
1	1•2	1•1	1•2	1•2
2	1•8	1•2	1•3	1•6
3	2•3	1•3	1•4	2•1
4	2•4	1•7	1•5	2•3
5	2•5	2•8	2•1	2•8
6	3•1	2•9	2•6	2•9
7	3•6	3•4	2•7	3•4
8	3•7	3•5	2•8	3•5
9	3•9	3•6	2•9	3•7

Fig. 10.4: Record Reconstruction Table for shipments

One last preliminary point: I won't bother to include any discussion of ORDER BY operations in my examples, because (a) I think they've been adequately discussed in earlier chapters already, and in any case (b) ORDER BY isn't really a relational operator as such, inasmuch as it doesn't produce a relation as its result (see Chapter 2, especially Sections 2.1 and 2.2).[1]

10.2 Restrict

Consider the following simple SQL query, which asks for a restriction of the shipments relation to just those tuples in which the shipment quantity is 200 (an *equality* restriction):

```
SELECT SPJ.S#, SPJ.P#, SPJ.J#, SPJ.QTY
FROM SPJ
WHERE SPJ.QTY = 200 ;
```

To implement this query, we[2] can start by doing a binary search on column QTY of the Field Values Table (Fig. 10.2), looking for a cell containing the value 200 (note that such a cell must be unique if it exists at all, because the column is condensed). If the search fails, we know immediately that the result of the query is an empty relation (one with no tuples). In the case at hand, however, the search succeeds; cell [*2,7*] of the Field Values Table is the one we want, and it contains, in addition to the specified QTY value, the row range [*3*:*6*]. It follows immediately that cells [*3,7*], [*4,7*], [*5,7*], and [*6,7*] of the shipments Record Reconstruction Table:

a) Contain row numbers for the cell in the merged Field Values Table that contains the QTY value 200 (and indeed they do all include the row number *2*), and

b) Contain row numbers for the "next" cell in the shipments Record Reconstruction Table.

Zigzags can therefore be constructed by following the appropriate pointer rings in the shipments Record Reconstruction Table. In the example, those zigzags look like this:

- *[3,7], [1,1], [2,5], [2,6]*
- *[4,7], [3,1], [3,5], [3,6]*
- *[5,7], [8,1], [7,5], [4,6]*
- *[6,7], [9,1], [9,5], [6,6]*

Following these zigzags through the shipments Record Reconstruction Table and accessing the merged Field Values Table accordingly, we obtain the desired result:

S#	P#	J#	QTY
S1	P1	J1	200
S2	P1	J1	200
S3	P3	J1	200
S3	P3	J2	200

For a second example, let's modify the query so that it involves a "less-than" comparison instead of an "equals" one:

```
SELECT SPJ.S#, SPJ.P#, SPJ.J#, SPJ.QTY
FROM SPJ
WHERE SPJ.QTY < 150 ;
```

It should be clear that this query too is easily handled, this time by:

a) Doing a *sequential* search (instead of a binary one) on column QTY of the Field Values Table;

b) Reconstructing all corresponding records, and hence user-level tuples, for each cell encountered during that search; and

c) Stopping as soon as we find a cell in column QTY of the Field Values Table that contains a QTY value of 150 or greater.

Here's the result:

S#	P#	J#	QTY
S1	P3	J2	100
S3	P1	J1	100

Now consider this query:

```
SELECT SPJ.S#, SPJ.P#, SPJ.J#, SPJ.QTY
FROM SPJ
WHERE SPJ.S# = S#('S3') AND SPJ.QTY = 100 ;
```

Here the WHERE clause involves two separate equality comparisons ANDed together. By means of searches on the S# and QTY columns of the Field Values Table, however, we can easily discover, from the applicable row ranges, that there are four shipments with supplier number S3 but only two with quantity 100. The best strategy is therefore to use the zigzags associated with quantity 100 and check during record reconstruction to see whether the supplier number is S3, stopping reconstruction of the record in question if it isn't.[3] Here's the result:

S#	P#	J#	QTY
S3	P1	J1	100

Finally, let's consider the effect of replacing the AND by an OR:

```
SELECT SPJ.S#, SPJ.P#, SPJ.J#, SPJ.QTY
FROM SPJ
WHERE SPJ.S# = S#('S3') OR SPJ.QTY = 100 ;
```

We can implement this query by, first, finding all tuples for supplier S3, and then finding all tuples not already found in the first step that have QTY value 100 (or the other way around). Assuming, reasonably enough, that the two steps are executed in such a manner that the two results produced are ordered in the same way (in ascending S# order, say), then they can simply be merged to produce the desired overall result. That result looks like this:

S#	P#	J#	QTY
S1	P3	J2	100
S3	P1	J1	100
S3	P2	J2	500
S3	P3	J1	200
S3	P3	J2	200

One happy—but novel—result of the foregoing is that, loosely speaking, OR and UNION have the same performance characteristics. That is, the following logically equivalent SQL query should be implemented in exactly the same way (and therefore exhibit exactly the same performance) as the one shown above:

```
SELECT SPJ.S#, SPJ.P#, SPJ.J#, SPJ.QTY
FROM SPJ
WHERE SPJ.S# = S#('S3')
UNION
SELECT SPJ.S#, SPJ.P#, SPJ.J#, SPJ.QTY
FROM SPJ
WHERE SPJ.QTY = 100 ;
```

Let me close the present section by contrasting the implementation approaches sketched above with what direct-image systems typically have to do. In general, such systems don't have the same kind of exact cardinality information that TR does;[4] to be specific, they typically don't know exactly how many tuples have a given value for a given attribute at a given time. Instead, they have to execute some kind of *statistics utility* every so often in order to compute those cardinalities, and then store them away somewhere. For example, in IBM's DB2 product [45], the utility in question is called RUNSTATS, and the computed statistics—cardinalities and other similar information—are stored in the DB2 catalog. Typically, the database administrator will ask for RUNSTATS to be executed whenever the database is reorganized or whenever it's been heavily updated. Quite apart from the overhead involved in actually running the utility, the fact is that computed values will naturally be out of date and inaccurate much of the time, and the optimizer might thus fail to choose the best strategy for implementing the query.

Note: You might reasonably object that the statistics will be out of date with TR too, if the implementation compiles user requests ahead of time (as DB2 and certain other SQL systems in fact do), instead of when those requests are actually executed. The point is, however, that the access path selection process is so simple and straightforward in TR that there's very little point in compiling requests ahead of time—not to mention the fact that TR will almost certainly select the access path that genuinely is optimal. This state of affairs is in strong contrast to "prior art," where the optimizer has to do a great deal of computation and yet still fails, frequently, to come up with the overall best access path.

10.3 Project

Here's an SQL example of a query involving projection ("Project the shipments relation over attributes S#, P#, and J#"):

```
SELECT SPJ.S#, SPJ.P#, SPJ.J#
FROM SPJ ;
```

Implementing this query is straightforward; essentially, we just go through the usual file reconstruction process for shipments, but skip the reconstruction step for attribute QTY in each record. Here's the result:

S#	P#	J#
S1	P1	J1
S1	P3	J2
S2	P1	J1
S2	P1	J2
S2	P2	J2
S3	P1	J1
S3	P2	J2
S3	P3	J1
S3	P3	J2

However, you might have noticed that I was cheating a little in this example. Since the attributes over which the projection is taken—that is, the attributes that aren't "projected away"—include all of the attributes of the sole key {S#,P#,J#} for relation SPJ, we know ahead of time that the query can't possibly produce any duplicate tuples. But suppose I change the query slightly, thus:

```
SELECT SPJ.S#, SPJ.P#
FROM SPJ ;
```

If you examine the previous result, you'll see that:

a) There are two tuples that both contain supplier number S2 and part number P1, and

b) There are two tuples that both contain supplier number S3 and part number P3,

and so it looks as if this query ought to produce a result that looks like this:

S#	P#	
S1	P1	
S1	P3	
S2	P1	*
S2	P1	*
S2	P2	
S3	P1	
S3	P2	
S3	P3	*
S3	P3	*

As a matter of fact, this *is* the result that SQL would give. However, that result is not a relation—it includes duplicate tuples (flagged above with asterisks). In particular, it has no candidate key, and a fortiori no primary key (notice that I haven't shown any attributes with double underlining).

Of course, TR can certainly produce this nonrelational result if desired—I mean, it can be used to implement SQL systems as well as relational ones, as already mentioned in Chapter 3—but I'm interested here in implementing relational operations specifically. In order to request the true relational projection operation (to obtain the true relational result) in an SQL system, we would have to amend the query to include the specification DISTINCT, as follows:[5]

```
SELECT DISTINCT SPJ.S#, SPJ.P#
FROM SPJ ;
```

The implementation of this revised query is essentially the same as before, except that the system should if possible process the Record Reconstruction Table for shipments in a sequence that will deliver tuples according to the major-to-minor ordering S#-then-P# (or P#-then-S#). In the example, this ordering is obtained by processing the Record Reconstruction Table (Fig. 10.4) in sequence by the S# column. Duplicates will be adjacent in this ordering and thus can easily be eliminated. The final result is:

S#	P#
S1	P1
S1	P3
S2	P1
S2	P2
S3	P1
S3	P2
S3	P3

Actually, this result can be obtained more directly from the Record Reconstruction Table for shipments (that is, without first constructing and then explicitly eliminating duplicates). Here are the first two columns of that table, extracted from Fig. 10.4:

	1	5
	S#	P#
1	1•2	1•1
2	1•8	1•2
3	2•3	1•3
4	2•4	1•7
5	2•5	2•8
6	3•1	2•9
7	3•6	3•4
8	3•7	3•5
9	3•9	3•6

Now consider (by way of example) supplier S2. From the row range [*3*:*5*] for this supplier in the Field Values Table (Fig 10.2), we know among other things that the rows of the shipments Record Reconstruction Table that apply to this supplier are rows *3, 4,* and *5*—that is, the applicable *cells* of that table are [*3,1*], [*4,1*], and [*5,1*], respectively. These cells happen to contain "next cell" row numbers *3, 4,* and *5,* respectively (see Fig. 10.4), and so the "next" cells in the Record Reconstruction Table, according to the usual zigzags, are cells [*3,5*], [*4,5*], and [*5,5*], respectively. And these latter cells contain pointers to the Field Values Table rows *1, 1,* and *2,* respectively. It's thus immediately clear that there are only two *distinct* part numbers corresponding to supplier S2—the one in the Field Values Table cell [*1,5*], which is P1, and the one in the Field Values Table cell [*2,5*], which is P2.

I'll close this section by pointing out explicitly that the example we've been discussing illustrates another important application of the major-to-minor orderings discussed in detail in Chapter 7. To be specific, such orderings can be very helpful in implementing the internal-level operation of eliminating duplicates. In general, duplicate elimination is required in connection with projection operations (as we've just seen), also with union operations (see Section 10.7) and certain aggregation operations (see Section 10.5).

10.4 Extend

You might possibly not be familiar with the relational *extend* operator (the term "extend" isn't used in SQL contexts, at least not with the meaning intended here, though SQL does provide the desired functionality). Basically, the extend operator takes a relation and returns a relation containing an extended form of each tuple from the given relation, where the extension in each case consists of an additional attribute value that's computed in accordance with some specified computational expression. Here's an—admittedly rather contrived—SQL example:

```
SELECT DISTINCT SPJ.S#, SPJ.P#, SPJ.J#, SPJ.QTY,
                                    ( ( 2 * SPJ.QTY ) - 150 ) AS XXX
FROM SPJ ;
```

Result:

S#	P#	J#	QTY	XXX
S1	P1	J1	200	250
S1	P3	J2	100	50
S2	P1	J1	200	250
S2	P1	J2	500	850
S2	P2	J2	500	850
S3	P1	J1	100	50
S3	P2	J2	500	850
S3	P3	J1	200	250
S3	P3	J2	200	250

The only point I want to make in connection with this example is that if we know this query is going to be executed fairly frequently, then we can treat the "computed" attribute XXX just like the regular ("base") attributes S#, P#, and so on; to be specific, we can map it to a column of its own in the Field Values Table. That column can then be sorted and condensed (possibly even merged), just like other such columns, and analogous benefits—fast binary search, use in major-to-minor orderings, and so on—will then immediately accrue.

How then can we know whether a given query will be frequently executed? Well, one possibility is to let the database administrator tell us, of course. Another is to *guess* ... If the foregoing SQL query is specified as the defining expression for a **view,** as here—

```
CREATE VIEW XSPJ
     AS SELECT DISTINCT SPJ.S#, SPJ.P#, SPJ.J#, SPJ.QTY,
                                        ( ( 2 * SPJ.QTY ) - 150 ) AS XXX
           FROM SPJ ;
```

—then it's a pretty safe bet that the query is indeed going to be executed fairly frequently.

10.5 Summarize

Summarize is the relational operator that underpins SQL's aggregation and GROUP BY operations. Here's an SQL example:

```
SELECT DISTINCT SPJ.S#, COUNT(*) AS SHIP_COUNT
FROM SPJ
GROUP BY SPJ.S# ;
```

The effect of this query is to "summarize" the shipments relation in a certain way. To be specific, it returns a relation that contains a tuple for each distinct supplier number in SPJ, giving a count (SHIP_COUNT) of the number of shipments the supplier in question is involved in. The result looks like this:

S#	SHIP_COUNT
S1	2
S2	3
S3	4

Observe now that this result is directly obtainable from the S# column of the Field Values Table. Here is that column, extracted from Fig. 10.2:

S#
S1 [*1*:*2*]
S2 [*3*:*5*]
S3 [*6*:*9*]
S4 [:]
S5 [:]

Recall that the row ranges indicate (among other things) which rows of the uncondensed Field Values Table for shipments the corresponding supplier number would appear in, if such a table were actually to be built. Thus we can see immediately that there are two shipments for supplier S1, three for supplier S2, four for supplier S3, and none at all for suppliers S4 and S5.[6] Note, however, that the result doesn't include tuples for suppliers S4 and S5 (with zero counts), because the SQL query specified "FROM SPJ," and suppliers S4 and S5 don't appear in relation SPJ at all. A relational query using SUMMARIZE that does include suppliers S4 and S5 in the result can easily be formulated (and easily implemented in TR). An SQL query to do the same thing can be formulated too, but the specifics are rather more complicated, and the details are beyond the scope of this book; for more discussion, see reference [32].

By the way, it would make no difference to either the meaning or the result of the foregoing SQL query if we were to replace the COUNT argument "*" by SPJ.P#, or SPJ.J#, or SPJ.QTY, or SPJ.QTY + 1, or indeed by just about any other syntactically valid expression you can think of—*unless* the expression in question is preceded by the specification DISTINCT, as here:

```
SELECT DISTINCT SPJ.S#, COUNT ( DISTINCT SPJ.P# ) AS PART_COUNT
FROM SPJ
GROUP BY SPJ.S# ;
```

The effect of this revised query is to return a relation that contains a tuple for each distinct supplier number in SPJ, giving a count NP of the number of *distinct* parts the supplier in question is shipping, thus:

S#	PART_COUNT
S1	2
S2	2
S3	3

This revised query requires a revised implementation, too: Basically, the system now needs to use the Record Reconstruction Table for shipments, processing it in a sequence that will deliver tuples according to the major-to-minor ordering S#-then-P#. Duplicate part numbers for a given supplier will be adjacent in this ordering and thus can easily be eliminated from the corresponding count. (As in the case of projection—see Section 10.3—it shouldn't be necessary actually to materialize the duplicates before eliminating them; the necessary information can in fact be obtained directly from the Record Reconstruction Table.)

Let's consider some of the other aggregate operators. MAX and MIN are easy enough. For example, consider the SQL query:

```
SELECT DISTINCT SPJ.S#, MIN ( SPJ.QTY ) AS MNQ
FROM SPJ
GROUP BY SPJ.S# ;
```

Here's the result:

S#	MNQ
S1	100
S2	200
S3	100

To see how this query is implemented, consider supplier S2 once again. As we already know (see the discussion of projection in Section 10.3), the cells in the shipments Record Reconstruction Table that correspond to this supplier number are [*3,1*], [*4,1*], and [*5,1*], respectively. Following the zigzags to the corresponding QTY cells in that table, we find that those cells contain pointers to the Field Values Table rows *2, 3,* and *3,* respectively. Since the QTY column (like all columns) in that table is kept in ascending order, it's immediately clear that the minimum QTY value for supplier S2 is the one in row *2* of the Field Values Table—namely, the QTY value 200.

Note: It should be obvious that it makes no difference in the case of MAX and MIN whether or not the argument to the aggregate operator includes a DISTINCT specification.

Other aggregate operators for which TR technology is particularly suited include MEDIAN and MODE. In case you're unfamiliar with these operators, let me explain them briefly here. Suppose we're given a collection of values, possibly including duplicates. Then the *median* of that collection is the value that appears in the middle position when the values are sorted, while the *mode* is the value that appears the most frequently. (Of course, these definitions require certain refinements, beyond the scope of this book, in order to take care of the question of ties and the like, but you get the general idea.) I'll leave it to you to figure out the corresponding TR implementation in each case.

10.6 Join

The join operation is often regarded as the sine qua non of relational systems.[7] Certainly it's extremely important; some might even say that relational systems stand or fall on the basis of how well—how effectively, how efficiently—they implement joins. What's more, there's a widespread perception that joins must perform poorly, almost by definition. Here's a typical quote (from an article critizing relational systems in general and the proposals of reference [40] in particular): "Database application developers ... have been baffled by the intolerable performance [incurred] ... by performing joins" [54]. And reference [63] has this to say:

> In prior art database systems, joins tend to be extremely costly in storage space and/or processing time, requiring either preindexed data to maintain sortedness or a time-intensive search involving multiple passes over the entirety of each attribute that is being joined.
>
> *—from the Initial Patent*

Let's take a closer look. Reference [32] describes a variety of techniques for implementing joins, the following among them:

- Brute force
- Index lookup
- Hash lookup
- Merge
- Hash
- Various combinations of the foregoing

Let me focus first on the *brute force* technique. Let *r* and *s* be the relations to be joined; let *r* and *s* have *M* tuples and *N* tuples, respectively, and let them have just one common attribute, *A*.[8] Let *R* and *S* be direct-image stored files corresponding to *r* and *s,* respectively, with stored records, in sequence, *R*[1], *R*[2], ..., *R*[*M*] and *S*[1], *S*[2], ..., *S*[*N*], again respectively. Here then is the brute force algorithm:

```
do i := 1 to M ;
     do j := 1 to N ;
          if R[i].A = S[j].A then
          append joined record R[i] * S[j] to result ;
     end ;
end ;
```

(I'm using the expression *R*[*i*] * *S*[*j*] to denote the joined record that's formed from the records *R*[*i*] and *S*[*j*].)

As you can see, the brute force technique is very simple-minded—basically, it just examines all possible combinations of records, one from *R* and one from *S,* and joins them together if and only if they have the same value for the common attribute *A* (or for the stored field corresponding to the common attribute *A,* rather). *Note:* The brute force algorithm is often referred to as "nested loops," but this name is misleading because nested loops are in fact involved in all of the conventional implementation algorithms.

Now, it should be obvious that the brute force approach involves a total of *M***N* record read operations. It should also be obvious that if we wanted to join *three* relations, *r, s,* and *t,* say, then the brute force approach will involve *M***N***P* record reads (where *P* is the number of tuples in *t*), and so on. In other words, the costs associated with the brute force algorithm are inherently **multiplicative** in nature. For that reason, that algorithm is generally regarded as the worst case, which is precisely why so much energy has been expended over the past 30 years or so on alternative approaches (index lookup, hash lookup, and the rest).

I don't want to go into a lot of detail on those alternative approaches here. Suffice it to say that they're all aimed, in one way or another, toward the goal of never having to read any record twice—or, preferably, toward the more demanding goal of being able to read each stored file in sequence just once (clearly an optimal state of affairs).

- For example, indexes or hashes on *R.A* and *S.A* could certainly mean that no record of either *R* or *S* is ever read twice. However, they probably wouldn't mean that the stored files are read in sequence just once, as I pointed out in Chapter 2. Also, of course, indexes and hashes lead to other problems, again as discussed in Chapter 2.

- Alternatively, we could sort the two stored files appropriately and then do a *merge* join—and merge join does mean that each stored file is read in sequence just once. Thus, a merge join of *r* and *s* will involve *M*+*N* record reads; a merge join of the three relations *r, s,* and *t* will involve *M*+*N*+*P* record reads; and so on. In other words, the costs associated with the merge approach are inherently **additive** (or **linear**), not multiplicative, in nature. (Of course, I'm ignoring the sort costs here, and those costs can be very significant in practice.)

I'd like to emphasize the dramatic difference between linear and multiplicative costs. Suppose for simplicity that every relation has 100,000 tuples (not at all a large number, by the way, in modern databases). Then the following table shows the number of record reads involved in various joins implemented by merge vs. the same joins implemented by brute force (assuming a direct-image style of implementation in both cases, of course):

	merge	*brute force*
2 relations	200,000	10,000,000,000
3 relations	300,000	1,000,000,000,000,000
4 relations	400,000	100,000,000,000,000,000,000
5 relations	500,000	10,000,000,000,000,000,000,000,000

Note in particular that each step (from two relations to three, from three to four, and so on) involves several orders of magnitude performance degradation with the brute force approach. In order to emphasize the point, suppose each record read takes ten microseconds. Then a merge join of the five relations will take just five seconds, while a brute force join of the same five relations will take over three trillion years, or some 200 times the current best estimate of the age of the universe (!). No wonder merge join is a preferred technique ... But the trouble with merge join, of course, is that it requires the stored files to be sorted into appropriate sequence first (that's why the technique is usually called, more specifically, *sort*/merge). And the beauty of the TR approach, as I've shown in earlier chapters, is that the stored files are already in the desired sort order, always. As I put it in Chapter 4, TR lets us do a sort/merge join without having to do the sort (indeed, we saw in Chapter 9 that it might effectively let us do the join without having to do the merge either). Thus, TR always does a merge join. Note the following implications:

- The more relations that need to be joined, the more the gain. In other words, the more complex the query, the more significant the TR advantage over direct-image systems (as already noted in Chapter 5).

- Because all joins are implemented the same way, we don't have to do that complex access path selection process that those direct-image systems do have to do.

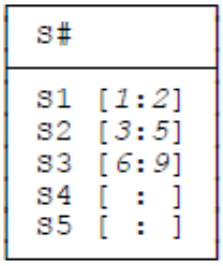
- That access path selection process that direct-image systems have to do is of dubious accuracy anyway, because of the difficulty of estimating intermediate result sizes, among other reasons.

- In fact, as reference [32] shows, there can easily be a huge number of possible strategies for implementing any given query in direct-image systems, precisely because of all the redundancies that indexes and other auxiliary structures introduce. For this reason, those systems typically employ a variety of heuristics for "reducing the search space"—that is, for eliminating certain strategies very early on in the access path selection process (possibly never even considering them at all). Those heuristics in turn (a) make the implementation still more complicated and (b) imply that a good strategy will sometimes be rejected in favor of a bad one.

By way of example, let's consider what's involved in TR in implementing the following SQL query (which asks for suppliers and shipments to be joined on supplier numbers):

```
SELECT DISTINCT S.S#, S.SNAME, S.STATUS, S.CITY, SPJ.P#, SPJ.J#, SPJ.QTY
FROM S, SPJ
WHERE S.S# = SPJ.S# ;
```

Here again is the S# column from the Field Values Table (extracted from Fig. 10.2):



From the information in this column we can see immediately that:

- The first tuple of relation S (for supplier S1) joins to the first and second tuples of relation SPJ.[9] The two joined tuples can be built by starting at cell [*1,1*] of the suppliers Record Reconstruction Table and cells [*1,1*] and [*2,1*] of the shipments Record Reconstruction Table.

- The second tuple of relation S (for supplier S2) joins to the third, fourth, and fifth tuples of relation SPJ. The three joined tuples can be built by starting at cell [*2,1*] of the suppliers Record Reconstruction Table and cells [*3,1*], [*4,1*], and [*5,1*] of the shipments Record Reconstruction Table.

- The third tuple of relation S (for supplier S3) joins to the sixth, seventh, eighth, and ninth tuples of relation SPJ. The four joined tuples can be built by starting at cell [*3,1*] of the suppliers Record Reconstruction Table and cells [*6,1*], [*7,1*], [*8,1*], and [*9,1*] of the shipments Record Reconstruction Table.

Execution of the query is now complete. Note in particular that the fourth and fifth tuples of relation S (for suppliers S4 and S5) don't join to any tuples of relation SPJ at all.

Now, I mentioned earlier in this section (by way of an endnote) that there are other kinds of joins as well as the natural join: equijoins, greater-than joins, and so on. Here's an SQL example of a greater-than join: to be specific, a greater-than join over city names between the suppliers relation S and the parts relation P from Chapter 8. *Note:* "Greater than" here just means—let's assume—"later in alphabetic ordering than" (recall our assumption in Chapter 2 that city names are simple CHAR strings).

```
SELECT DISTINCT S.S#, S.SNAME, S.STATUS, S.CITY AS SCITY
                P.P#, P.PNAME, P.COLOR, P.WEIGHT, P.CITY AS PCITY
FROM S, P
WHERE S.CITY > P.CITY ;
```

Here's the result:

S#	SNAME	STATUS	SCITY	P#	PNAME	COLOR	WEIGHT	PCITY
S2	Jones	10	Paris	P1	Nut	Red	12.0	London
S2	Jones	10	Paris	P4	Screw	Red	14.0	London
S2	Jones	10	Paris	P6	Cog	Red	19.0	London
S2	Jones	10	Paris	P3	Screw	Blue	17.0	Oslo
S3	Blake	30	Paris	P1	Nut	Red	12.0	London
S3	Blake	30	Paris	P4	Screw	Red	14.0	London
S3	Blake	30	Paris	P6	Cog	Red	19.0	London
S3	Blake	30	Paris	P3	Screw	Blue	17.0	Oslo

And here are the CITY columns from the suppliers and parts Field Values Tables (suppliers on the left, parts on the right):

CITY
Athens [*1*:*1*]
London [*2*:*3*]
Paris [*4*:*5*]

CITY
London [*1*:*3*]
Oslo [*4*:*4*]
Paris [*5*:*6*]

For convenience, let's merge these two columns together, as follows[10] (the first row range for each city corresponds to suppliers and the second to parts):

CITY		
Athens	[*1*:*1*]	[:]
London	[*2*:*3*]	[*1*:*3*]
Oslo	[:]	[*4*:*4*]
Paris	[*4*:*5*]	[*5*:*6*]

It's clear from this merged column that the "fourth" and "fifth" supplier tuples both join to each of the "first," "second," "third," and "fourth" part tuples—where "fourth," fifth," etc., are to be interpreted in terms of CITY ordering in both cases—and nothing else joins to anything else. Thus, I think you can see that the desired greater-than join can again be implemented by a kind of merging process, although the details are a little more complicated than they are in the natural join case; in particular, several passes are needed over the row ranges for either parts or suppliers (not both). *Note:* This latter fact might be a good reason for physically storing row ranges in a table of their own, separate from the Field Values Table, as suggested in Chapter 8.

10.7 Union, Intersect, and Difference

The relational operators union, intersect, and difference all require their two input relations to have exactly the same attributes [33]. As a basis for my examples in this section, therefore, I'll consider the *projections* of the suppliers and parts relations on their CITY attributes (since those two projections certainly do have exactly the same attributes). Here then are some SQL examples, with corresponding results:

```
SELECT DISTINCT S.CITY
FROM   S
UNION
SELECT DISTINCT P.CITY
FROM   P ;
```

CITY
Athens
London
Oslo
Paris

```
SELECT DISTINCT S.CITY
FROM   S
INTERSECT
SELECT DISTINCT P.CITY
FROM   P ;
```

CITY
London
Paris

```
SELECT DISTINCT S.CITY
FROM   S
EXCEPT
SELECT DISTINCT P.CITY
FROM   P ;
```

CITY
Athens

```
SELECT DISTINCT P.CITY
FROM   P
EXCEPT
SELECT DISTINCT S.CITY
FROM   S ;
```

CITY
Oslo

Note that SQL uses the keyword EXCEPT to denote the relational difference operator. Note too that UNION, INTERSECT, and EXCEPT—unlike SELECT—all eliminate duplicates by default in SQL (implying that all of the DISTINCT operators shown above are in fact logically unnecessary).

Here now, repeated from the previous section, is a merged Field Values Table CITY column (supplier row ranges on the left, part row ranges on the right):

CITY		
Athens	[1:1]	[:]
London	[2:3]	[1:3]
Oslo	[:]	[4:4]
Paris	[4:5]	[5:6]

The use of this merged column in implementing the foregoing union, intersect, and difference operations should be obvious. In essence:

- *Union:* A given city name appears in the result if and only if it has a nonempty row range for suppliers or parts or both. In other words, the union is just the set of all city names in the merged column.

- *Intersect:* A given city name appears in the result if and only if it has a nonempty row range for both suppliers and parts.

- *Difference:* For the difference between supplier cities and part cities, in that order, a given city name appears in the result if and only if it has a nonempty row range for suppliers and an empty one for parts. Similarly, for the difference between part cities and supplier cities, in *that* order, a given city name appears in the result if and only if it has a nonempty row range for parts and an empty one for suppliers.

All of these operations can clearly be implemented in a single pass over the merged Field Values Table CITY column.

Incidentally, if that CITY column contains a large number of entries, the performance of intersect and difference operations, at least, might be improved by means of *bitmaps*. In the example, we would have two such bitmaps, one to indicate whether the city name in question appears in the suppliers relation and the other to indicate whether it appears in the parts relation. Here's a modified version of the merged column that includes such bitmaps (1 = yes, 0 = no):[11]

CITY			
Athens [1:1]	1	[:]	0
London [2:3]	1	[1:3]	1
Oslo [:]	0	[4:4]	1
Paris [4:5]	1	[5:6]	1

The city names appearing in the intersection can now be pinpointed by executing a logical AND on the two bitmaps, while those appearing in the difference between supplier cities and part cities, in that order, can be pinpointed by executing a logical AND on the suppliers bitmap and the negation (logical complement) of the parts bitmap. Since logical operations like AND and NOT are usually supported directly in hardware, the implementation of the corresponding relational operations now has the potential to be very fast indeed.

10.8 Materializing Derived Relations

Sometimes it's necessary for a relational implementation to materialize some derived relation—that is, to build a concrete representation in storage of the result of some relational expression. Just why and when such materialization might be necessary is a question I don't particularly want to get into here; rather, what I do want to do is examine the question of what's involved in performing such materialization, when it *is* necessary, in the case of TR specifically.

Materializing a derived relation in TR means, of course, building an appropriate set of Field Values and Record Reconstruction Table entries for that relation. One obvious point that arises immediately, therefore, is that materialization is likely to be easier in TR than it is in other approaches, because a single Field Values Table can effectively be shared across several different relations, thanks to the merged-columns feature. In other words, it might not be necessary to build a new Field Values Table for the derived relation at all, in which case it could be argued that materialization as such isn't really being done (because it's simply not needed). These remarks apply directly to the monadic case, where the derived relation is obtained by means of some monadic relational operator (restrict, project, extend, summarize); they might possibly also apply to the dyadic case, where the derived relation is obtained by means of some dyadic relational operator (join, union, intersect, difference). *Note:* I'm using the terms *monadic* and *dyadic* here to refer to relational operators that take one relational operand and two relational operands, respectively.

Let's now make the worst-case assumption; that is, let's assume that we do actually have to build a brand new Field Values Table and a brand new Record Reconstruction Table for the derived relation in question. Suppose, for example, that we need to materialize the result of joining suppliers and parts over city names. Well, it's easy to see intuitively that, in general, the biggest overhead in building a Field Values Table and a Record Reconstruction Table is all the sorting of field values that's required. But in the case at hand, most if not all of the sorting has already been done—every column of the suppliers Field Values Table is already in sorted order, and the same is true of every column of the parts Field Values Table as well. Analogous remarks apply to the other relational operators, of course. (As a matter of fact, they even apply to some extent to the pointer values in the corresponding Record Reconstruction Tables also; they too tend to be sorted, at least partially. See, for example, the Record Reconstruction Table shown in Fig. 7.4 in Chapter 7, also the remarks on this topic at the end of Section 7.5 in that same chapter.)

In a nutshell, then, materialization in TR (a) is needed less often than it is in traditional implementations and (b) is more efficient, when it *is* needed, than it is in traditional implementations.

10.9 A Note Regarding Optimization

This brings me to the end of my discussion of how relational operators can be implemented using the TR model. However, there are still a few topics—three of them, to be precise—that I'd like to say something about, briefly, before I close the chapter. The first has to do with the system optimizer.

The optimizer is, of course, that component of the system that decides how to implement any given user request. Now, I've suggested at numerous points in previous discussions, both in this chapter and in several earlier chapters, that TR makes life easier for the optimizer; to be specific, it makes the access path selection process easier (even completely unnecessary, in some cases). However, I don't want to give the impression that the optimizer is no longer necessary. The fact is, there are two broad facets to the optimizer's job, both of them (in general) important, access path selection and **expression transformation** (sometimes called **query rewrite)**. And even if access path selection does become unnecessary (or almost so), query rewrite does not.

Query rewrite is the process of converting a given relational expression into another such expression that (a) is logically equivalent to the original one, in the sense that it's guaranteed to produce the same result when evaluated, but (b) has a good likelihood of being more efficient—that is, performing better—than the original one. I'll give just one simple example (expressed in SQL for reasons of familiarity): The expression

```
SELECT DISTINCT X.CITY
FROM ( SELECT DISTINCT S.S#, S.STATUS, S.CITY
       FROM    S) AS X ;
```

(a projection of a projection) can be "rewritten" as the simpler expression

```
SELECT DISTINCT X.CITY
FROM S AS X ;
```

The rewrite has eliminated one of the projections, and that's why the result is more efficient.

Note: You might be thinking that the foregoing example is somewhat contrived ("no user in his or her right mind would state the query in the first form anyway"—right?). In fact, however, the example is quite realistic. Suppose we have the following view:

```
CREATE VIEW X
     AS SELECT DISTINCT S.S#, S.STATUS, S.CITY
           FROM S ;
```

And suppose the user issues the following query:

```
SELECT DISTINCT X.CITY
FROM X ;
```

Then the first thing the system does in processing this query is (in effect) convert it into the following:

```
SELECT DISTINCT X.CITY
FROM ( SELECT DISTINCT S.S#, S.STATUS, S.CITY
         FROM S ) AS X ;
```

Rewriting this query as previously suggested is thus clearly very desirable.

That said, I should now make it clear that query rewrite is not a TR responsibility as such; rather, it's a task that needs to be performed by code that sits above the TR level. For that reason, I don't want to discuss it any further here.

10.10 A Note Regarding Constraints

The second piece of unfinished business has to do with **integrity constraints**. Such constraints are vitally important, both in theory and in practice (see reference [36]), yet I've said almost nothing about them in this book so far, and it would be very remiss of me to ignore them altogether.

Basically, an integrity constraint is a conditional expression (also known as a boolean, truth-valued, or logical expression) that's required to evaluate to true. Here are a few examples, expressed in natural language for simplicity:

1. Every supplier status value is in the range 1 to 100 inclusive.

2. Every part weight is greater than zero.

3. Every supplier in London has status 20.

4. If there are any parts at all, at least one of them is blue.

5. No two distinct suppliers have the same supplier number.

6. Every shipment involves an existing supplier.

7. No supplier with status less than 20 supplies any part in a quantity greater than 500.

And so on.

Of course, it's the job of the database administrator to state such constraints (using SQL or some other formal language),[12] and it's the job of the DBMS to implement them. But implementing constraints isn't the same thing as implementing the relational operators; in fact, the system component that implements constraints will in all likelihood make use of the relational operators to do so, and therefore will have to invoke the lower-level component that does implement those relational operators. In a TR system, in other words, many constraints—perhaps most—will be implemented by code that sits, not on top of the TR level directly, but on top of the relational operator implementation level that does sit on top of the TR level directly. That's basically why I haven't had much to say about constraints in this book prior to this point.

I must now immediately add that there are likely to be some exceptions—rather important ones—to the foregoing. Consider again the following example:

5. No two distinct suppliers have the same supplier number.

The formal statement of this constraint is, of course, simply a specification to the effect that {S#} is a key—more precisely, a *candidate* key—for the suppliers relation, and the implementation has to guarantee that no two supplier tuples appearing in the suppliers relation at the same time ever have the same supplier number. But as I explained in Chapter 6 (Section 6.5), this guarantee is effectively built into the implementation of the INSERT operator (the UPDATE operator too, as a matter of fact). To repeat the example from that section, suppose we try to insert a supplier tuple for supplier S9. At the TR level, then, the system will have to inspect the supplier number column in the Field Values Table (probably using a binary search), looking for the appropriate insert point for the new supplier number S9; and if it discovers that the supplier number value S9 already exists, then clearly it can reject the INSERT (or UPDATE). In other words, key constraints can and will effectively be implemented directly at the TR level.

The second example I want to discuss is this one:

6. Every shipment involves an existing supplier.

The formal statement of this constraint is a specification to the effect that {S#} in the shipments relation SPJ is a foreign key referencing the candidate key {S#} of the suppliers relation S (every supplier number currently appearing in SPJ must currently appear in S as well). And the point I want to make here is this: If the Field Values Tables for suppliers and shipments are merged on their S# column, as shown in Fig. 10.2, then the mechanism for enforcing this foreign key constraint for shipments is very similar to that discussed above for enforcing the candidate key constraint for suppliers. In other words, foreign key constraints too can, and probably will, effectively be implemented directly at the TR level.

It's appropriate to close this section by mentioning that in direct-image systems, both candidate and foreign key constraints are typically enforced by means of indexes, or sometimes by hashes or other auxiliary structures.

10.11 What's Missing?

The third and last piece of unfinished business has to do with **missing information**. Examples of missing information include such things as "date of birth unknown," "speaker to be announced," "present address not known," and so on. And as you probably know, SQL systems in particular address this issue—or attempt to address it, rather—by means of a construct called a **null**. For example, suppose we know some particular part exists, but we don't know its weight. Then we might say, loosely, that "the weight is null"—meaning, more precisely, that (a) we do know the part has a weight, because all parts have a weight, but (b) to repeat, we don't know what that weight is. So we can't put any sensible value at all in the WEIGHT position within the pertinent tuple; instead, therefore, we *flag* or *mark* that position as "being null."

Now, you've probably noticed that I've said essentially nothing about this topic in this book prior to this point. And the major reason for that omission is that, so far as I'm concerned, *nulls in the foregoing sense have absolutely no place in the relational model*—and, of course, I've been concentrating in this book so far on the application of TR concepts to implementing the relational model specifically. I don't want to get into a lot of detail here as to why I—and indeed most other writers on the relational model, though not all [38]—reject nulls categorically; this book would be the wrong forum for such a discussion. Let me just say, therefore, that:

a) There are very sound reasons, both theoretical and practical,[13] for not including nulls in the relational model itself. See references [18], [32], [40], [43-44], and especially [58] for a discussion of some of the theoretical reasons, and references [18-19] and [22-23] for a discussion of some of the practical ones.

b) There are also very sound reasons for not using nulls, even when they're supported, as they are in SQL. Thus, I recommend strongly that, even if you have to use SQL, you don't try to "take advantage of" the nulls feature of that language. In other words, nulls are contraindicated even when they're supported. See references [17], [32], and [39] for arguments in support of this position.

Given the foregoing state of affairs, I don't propose to discuss the use of TR to implement an SQL-style nulls feature at all. I'll just say that—of course—TR *can* be used to implement such a feature if desired, and that many of the advantages I've been claiming for a TR implementation of the relational model would apply to such an implementation, too. So yes, TR can be used to implement SQL as well as the relational model.[14]

Endnotes

1. If you happen to be familiar with the relational model, you might notice another omission, too: There's no discussion of the relational divide operator. One reason for this omission (not the only one) is that I'll be arguing in Chapter 15 that relational comparisons really ought to be supported. If they are, then the divide operator becomes logically unnecessary [40].
2. I'll use the term "we" throughout this chapter, a trifle sloppily, to mean either the DBMS designers and implementers or the DBMS itself, as the context demands.
3. Checking the supplier number will be quite speedy, too, because column S# and column QTY happen to be logically adjacent within those zigzags. See the remarks on this subject at the very end of Chapter 5.
4. The *cardinality* of a set is the number of elements the set contains.
5. As I've written elsewhere [17], my own recommendation would be that users shouldn't have to waste time thinking about whether a given SQL query can produce duplicates or not but should always specify DISTINCT, and leave it to the system to figure out when such a specification can safely be ignored. Of course, I haven't followed my own advice in this respect in this book so far!—but I'll do so from this point forward (you might like to try the exercise of figuring out in each case whether the DISTINCT can safely be ignored). To quote Hugh Darwen [11]: "If you have to use ... DISTINCT to obtain a true [relational result], do not fail to do so, *but be annoyed about it*" (my italics).
6. Intuitively, the reason COUNT and the other aggregate operators discussed in the present section can be so easily and efficiently implemented in TR is because—as noted in Chapter 8, Section 8.2—the row ranges in the Field Values Table can effectively be regarded as histograms.

7. As you might have already noticed, I use the unqualified term *join* to mean the natural join specifically [33,40]. This practice is both common and convenient in relational contexts. However, other kinds of joins do exist: equijoins, greater-than joins, and so on (see reference [32]). I'll have a little more to say regarding these other kinds of joins toward the end of the present section.
8. I make this assumption for simplicity only. Everything said regarding TR in this section extends gracefully and straightforwardly to the case where there are two or more common attributes (recall from Chapter 9, Section 9.4, that the TR model effectively already includes a means by which two or more attributes can be treated as a single "combined" attribute if desired). Analogous remarks apply to other operators also, including in particular union, intersect, and difference (see Section 10.7).
9. See the remarks at the end of Section 9.2 in Chapter 9 for an explanation of what I mean by expressions like "the first tuple of relation S" and "the first and second tuples of relation SPJ."
10. In fact, the TR join implementation process will do this automatically, if the columns haven't been merged already. Of course, the merging does mean that changes will be required to the corresponding Record Reconstruction Tables, too, but those changes are essentially trivial.
11. The bitmaps are logically redundant, of course. Also, they're nothing to do with bitmap indexing, a topic that was mentioned in passing in Chapter 2 (Section 2.3).
12. All of the examples shown can in fact be formulated in SQL [39]. I omit such formulations for brevity.
13. Actually I believe theoretical reasons *are* practical ones, but that's another big discussion I don't want to get into here.
14. It's appropriate to add that TR is probably much better suited to implementing a truly relational solution—which isn't what the SQL "solution" is!—to the problem of missing information (thanks to Hugh Darwen for this observation). See references [43-44] and [58].

Part III: Disk-Based Implementation

11 General Disk Considerations

11.1 Introduction

So far in this book I've tacitly assumed—for the most part, at any rate—that the entire database is in main memory at run time. Now I need to consider what happens if that assumption is invalid (which will usually be the case in practice, of course). Such is the purpose of this part of the book.

Now, I claimed in Chapter 1 that divide-and-conquer is always a good pedagogical approach, and I appealed to that fact as my justification for largely ignoring disk-specific issues prior to this point. But there's more to it than mere pedagogy; the fact is, divide-and-conquer can be a good approach to design problems as well. The reason is that, in general, making simplifying assumptions and sticking with them for as long as possible can serve to clarify issues that might otherwise remain comparatively opaque. By way of one example, it was the initial assumption of a static, read-only database that led to the highly original TR approach to updating described in Chapter 6. By way of another, it was the initial assumption that everything could be kept in main memory that led to the (again highly original) logical data transforms described throughout the chapters in Part II.

Recall now that in Chapter 8 I characterized the TR data representation as **permutations and histograms;** that is, the logical data transforms just mentioned can be thought of as transforms that map a direct-image version of the data into such permutations and histograms. So when we get to a disk-based implementation, the question becomes: How can we transform those permutations and histograms still further in order to get the best possible representation of them in terms of storage structures on the disk? In other words, what *physical* transforms should we now carry out on the already logically transformed data? Observe how divide-and-conquer comes into play again; we don't even begin to think about looking for a good physical transform until we've carried out a good logical transform first. That's because (as the history of direct-image implementations strongly suggests) it's hard to find an optimized disk representation if we don't have an optimized main-memory representation to start with.

What I want to do, then, in the rest of this chapter and in the next three, is describe a particular set of physical transforms that can be used in TR in order to achieve "main-memory performance off the disk" (to put matters catchily, if not all that precisely). The present chapter considers the problem in general terms; the next three chapters then go on to discuss certain highly TR-specific approaches to that general problem.

Please note that this part of the book, even more than Part II, is not meant to be exhaustive. Rather, it's meant to give you some idea of what's involved in producing a good disk-based implementation of the TR model, without getting too deeply into numerous variations and alternative possibilities. My major aim is to convince you that a good disk-based TR implementation is indeed feasible, and what's more is likely to display some very attractive characteristics (performance characteristics in particular).

I should say too that this part of the book does assume you've read Part II carefully and mastered the key ideas contained therein—probably by doing the exercises as recommended (though you might be glad to hear there aren't any exercises in Part III).

Let me close this section with a couple of points of terminology that I'll be relying on throughout what follows. First, I remind you from Chapter 1 that I use the term *memory,* unqualified, to mean main memory specifically. Second, I'll use the term *memory-resident* to mean that the pertinent data, whatever it might happen to be, has already been brought into memory before we need it at run time.

11.2 What's the Problem?

Clearly, the disk implementation problem in general terms is simply to minimize the time it takes to find the data we want and read it off the disk. So let's briefly review what's involved in that "finding and reading" process. There are two main aspects to consider:

- *Seek time:* This is the time it takes to move the disk read/write head from its current position to the desired block or page. Seek times are measured in milliseconds (msec); they can be anything from 2 to 60 msec, with 6 msec being a good typical figure. By contrast, a typical "seek time" for memory might be 60 *nanoseconds* (nsec); thus, disk access is around 100,000 times, or five orders of magnitude, slower than memory access. The implications are obvious—we clearly want to jump around randomly on disk as little as possible; that is, we want to keep seek activity to a minimum. For otherwise we'll be in a situation in which overall system performance is totally dominated by the time spent doing seeks on the disk.

- *Data rate:* This is the speed at which data can be read off the disk once the read/write head has been positioned to the desired block or page. Data rates are measured in megabytes per second (MB/sec); they can be anything from 4 to 40 MB/sec, with 10 MB/sec being a good typical figure.[1] However, several disk drives can be attached to the same I/O channel, and channel data rates can reach as much as 256 MB/sec; thus, it might be possible by interleaving accesses to different disks to achieve an *effective* data rate across the channel of (say) 160 MB/sec or so. At that rate, if we take the average seek time to be 6 msec as suggested above, then one seek takes about the same amount of time as it takes to read one megabyte off the disk. What's more, it also takes about the same amount of time as it takes to scan one megabyte of data in memory, at least to a first approximation (I'm assuming here, not very realistically, that data is accessed in memory a single byte at a time).

Note: Actually there's a third aspect to the problem of finding and reading disk data, the *latency* or *rotational delay* aspect, which is the time spent waiting for the rotation of the disk to bring the desired block or page under the read/write head. For simplicity I've lumped this aspect in with seek time above.

Let me now elaborate briefly on the implications of considerations such as those above in the case of TR specifically. Observe first that, from the user's perspective, there are basically two general tasks that any DBMS needs to be able to perform (and perform well):

a) Given a particular tuple, find all of its attribute values;

b) Given a particular attribute value, find all of the tuples that contain it.

Now, classical direct-image systems are quite good on the first of these tasks (even on disk), but they're not very good on the second (not even in memory). By contrast, TR is very good on both, at least so long as we limit ourselves to a memory-based implementation:

a) Finding all attribute values for a particular tuple is basically the process of record reconstruction, using the appropriate zigzag in the Record Reconstruction Table;

b) Finding all tuples with a particular attribute value is basically the process of doing a binary search on the appropriate column in the Field Values Table.

But what happens on disk? The algorithms that work so well in memory clearly won't work so well on disk. To be specific, following zigzags and doing binary searches both effectively imply a lot of random jumping around, and disk performance is thus likely to be terrible unless we can come up with some good physical transforms. Such transforms are the subject of the remainder of this chapter.

11.3 Addressing the Problem

In this section I'll offer some general remarks regarding those good physical transforms; in subsequent sections, I'll focus in on some more specific issues and go into more detail. However, I should warn you that those subsequent sections do unavoidably involve a certain amount of cross-referencing among themselves, because the techniques I'll be describing aren't all independent of one another. But first things first.

First of all, then, we'd obviously like to have as much of the database as possible resident in memory at run time. One important technique for achieving this goal is *data compression,* which reduces not only the amount of space the data requires on the disk but also, and more importantly, the amount of space it requires in memory. (Of course, it also reduces the amount of time it takes to find and read the data, and so it's also relevant to the discussion of seek and read times below.) Sections 11.4 and 11.5 discuss specific compression techniques that apply to the Field Values Table and the Record Reconstruction Table, respectively.

Second, when we do have to access the disk because the data we need isn't memory-resident, we'd clearly like to minimize the amount of seeking we have to do. Several techniques are available to help here:

- *Large pages:* Page sizes in today's commercial DBMS products typically range from a minimum of one kilobyte, or even less, to a maximum of perhaps 64 kilobytes (1KB-64KB). As a consequence, the ratio of seek time to read time—"the seek-to-read ratio"—is usually quite high, ranging from around 1,000:1 for 1KB pages to around 16:1 for 64KB pages, if seek time is 6 msec and data rate is 160 MB/sec. In other words, most disk access time in today's systems is typically taken up in seek activity. Using larger pages of (say) one megabyte each will clearly reduce the seek-to-read ratio to something much more reasonable (approximately 1:1 for 1MB pages).

Note: The foregoing analysis does tacitly assume that everything in the page in question is "useful," in the sense that reading the whole page doesn't mean bringing into memory—and taking the time to bring into memory—a lot of data that's irrelevant to the purpose at hand. If we're in a complex-query environment (a data warehouse system, for example), then this assumption isn't too unrealistic. By contrast, if we're in an environment in which the queries are comparatively simple—an OLTP system, for example[2]—then the design tradeoffs might be different (in particular, smaller pages might be desirable). In what follows, I'll tend to assume the complex-query environment, where it makes any difference.

- *Streaming:* Next, we try to arrange matters such that if page *p2* is needed immediately after page *p1* at run time, then page *p2* immediately follows page *p1* on the disk. In this way, moving the read/write head from page *p1* to page *p2* involves little or no seeking, and data can be "streamed" off the disk into memory at a data rate close to the theoretical maximum. *Note:* The next item below, column-wise storage, is highly pertinent to this idea of streaming.

- *Column-wise storage:* Both the Field Values Table and the Record Reconstruction Table are accessed column-wise, at least initially (the Field Values Table when doing binary searches and the Record Reconstruction Table when starting to chase successive zigzags). For this reason, it's a good idea to store both tables column-wise on the disk, so that data items that are logically required together are physically close together on the disk. (In case it isn't clear what I mean when I say the tables are stored column-wise on the disk, let me explain briefly. In essence, what I mean is that column *1* is stored as a set of consecutive pages on the disk, then column *2* is stored as an immediately following set of consecutive pages, and so on.)

Note: The idea of storing the data column-wise is not so important (though certainly not *un*important) in the case of the Field Values Table, because that table will almost certainly be in memory at run time anyway (see Section 11.4). However, it's very important in the case of the Record Reconstruction Table (see Section 11.5), and becomes even more so if the techniques of Chapter 14 are adopted.

- *Banding:* We've seen that zigzags don't work so well if they mean jumping all over the disk. Banding is a solution to this problem; it's discussed briefly in Section 11.6 and in more detail in Chapter 13.

- *Using stars instead of zigzags:* Another solution to the problem of "zigzagging all over the disk" is to replace the zigzags by **stars**. Stars are also discussed briefly in Section 11.6 and in considerably more detail in Chapter 14.

- *Controlled redundancy:* Both banding and stars have the property that they can undermine the objective of symmetric performance (see Chapter 5, Section 5.2). We can address this problem by introducing a degree of controlled redundancy into the storage representations. Controlled redundancy is also discussed briefly in Section 11.6 and in more detail in Chapters 13 and 14.

11.4 Compressing the Field Values Table

Recall from Chapter 4 that the Field Values Table is the only TR table that contains user data as such. Recall too from Chapter 6 that although we refer to it as a table, it isn't physically stored as a table; instead, as noted in the previous section, it's stored column-wise, or in other words as a set of *vectors* (typically), one such vector for each column. And a variety of techniques, some primarily logical in nature and others more physical, are available for compressing those vectors. Let's take a closer look.

Logical Compression

By the term *logical compression,* I mean techniques that transform the data before it even reaches the disk, as it were. All of the techniques discussed in Chapters 8 and 9 fall into this category, including in particular the fundamental ones of condensing and merging columns. A variety of other possibilities also exist, including:

- *Mapping combinations of fields to a single column* (see Chapter 8, Section 8.5). This technique allows two or more vectors to be replaced by one whose length is less—often much less—than the sum of the lengths of the original ones.

- *Breaking fields into subfields,* also known as *subfield encoding* (see below). This technique allows one long vector to be replaced by two or more much shorter ones.

- And several others (again, see Chapter 8, Section 8.5).

Note: File factoring is another logical compression technique that applies to the Field Values Table. However, that technique, though it does indeed have the effect of compressing the Field Values Table, usually has a much more dramatic effect on the Record Reconstruction Table, and it's this latter compression that's the real point. For that reason, I'll defer further discussion of such factoring to the next section.

By the way, regarding column condensing specifically, I'd like to remind you that the amount of compression achievable can be dramatic—recall the example from Chapter 8 of a relation representing drivers' licenses, where the compression ratio was quite literally of the order of a million or so to one. On the other hand, column condensing won't do much for a field whose values are unique or almost unique; for such a field, subfield encoding (see below) or the techniques of Chapter 12 are likely to be more appropriate.

Let me now explain the concept I've mentioned a couple of times already, **subfield encoding**. Subfield encoding represents an additional refinement on the basic idea of condensed columns. The objective is to reduce overall space requirements still further, by breaking a given field into "subfields," each of which has far fewer distinct values than does the original field overall. For example, suppose we have a relation containing one tuple for each phone number in the United States, giving the names of persons or organizations reachable via those phone numbers. Assume for definiteness that there are 200 million tuples in the relation, so the number of distinct values of the PHONE# attribute (equivalently, the number of distinct values of the PHONE# field in the file corresponding to the relation) is 200 million.[3] Assume too for simplicity that the PHONE# field is ten bytes wide, one byte for each digit. Then column PHONE# of the Field Values Table will require 2,000 megabytes, and column PHONE# of the Record Reconstruction Table will require 1,400 megabytes (two pointers per cell, each pointer requiring 28 bits), for a total of 3,400 megabytes. *Note:* I'm relying here and throughout my discussion of this example that pointers are only as big as they logically need to be. This concept is discussed in detail in Section 11.5.

Note, however, that even though there are 200 million different phone numbers, there certainly aren't 200 million different area codes—in fact, there are only a few hundred. For definiteness again, let's assume there are just 250 area codes, with an average of 800,000 phone numbers within each one (250 * 800,000 = 200 million). Let's assume further that there are just 200 different prefixes or "exchanges" within each area code (first three digits of the phone number) and 4,000 different numbers within each area code and prefix (last four digits). So let's break the PHONE# field down into three subfields: AREA_CODE (three bytes), PREFIX (three bytes), and REST (four bytes). Here's what happens:

- Column AREA_CODE requires just 750 bytes (plus space for row ranges) in the Field Values Table, which we can ignore; 200 megabytes in the Record Reconstruction Table for pointers into the Field Values Table (each such pointer will be eight bits); and 700 megabytes in the Record Reconstruction Table for "next cell" pointers.

- Column PREFIX requires just 600 bytes (plus space for row ranges) in the Field Values Table, which we can ignore; 200 megabytes in the Record Reconstruction Table for pointers into the Field Values Table (each such pointer will again be eight bits); and 700 megabytes for "next cell" pointers.

- Column REST requires 16,000 bytes (plus space for row ranges) in the Field Values Table, which once again we can ignore; 300 megabytes in the Record Reconstruction Table for pointers into the Field Values Table (each such pointer will be twelve bits); and 700 megabytes for "next cell" pointers.

The grand total is approximately 2,800 megabytes, or a saving of roughly 17.6 percent compared with the original figure of 3,400 megabytes. What's more, this saving has been achieved even though the relevant portions of the Record Reconstruction Table have actually doubled in size. The point is, the relevant portions of the Field Values Table have effectively been reduced to *zero* size.

Note: Suppose we decide not to include pointers from the Record Reconstruction Table into the Field Values Table (after all, such inclusion was characterized as an "optional extra" in Chapter 8, at the end of Section 8.3). Then the grand totals of 3,400 megabytes and 2,800 megabytes reduce to 2,800 megabytes and 2,100 megabytes, respectively, and the saving becomes 25 percent.

Physical Compression

I'm using the term *physical compression* to mean techniques that effectively treat the output from the logical compressions discussed above—condensing, merging, and so on—simply as a set of very long bit strings and compress those bit strings "mechanically," without paying any attention to what those bit strings might represent. Under this general heading, there's just one point I want to discuss in any detail: namely, the fact that, in TR, such bit strings are always stored **bit-aligned** on the disk, instead of being aligned on (say) a fullword or four-byte boundary. By way of example, suppose we have a field *F* of type INTEGER; assume for the sake of the example that type INTEGER denotes integers in the range -2^{31} to $2^{31}-1$. Suppose, however, that field *F* actually holds values in the range 0 to 99 only. In a conventional system, each *F* value will still require four bytes of storage. In TR, by contrast, it will require only *seven bits*—a saving of over 78 percent (see Fig. 11.1).

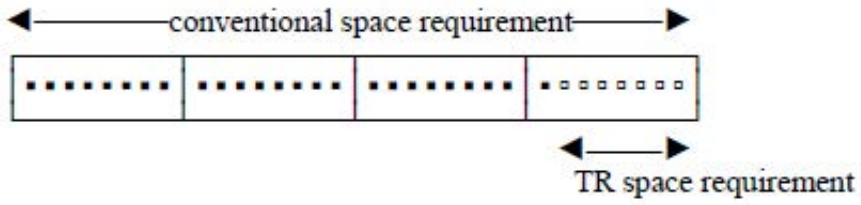


Fig. 11.1: Bit alignment (example)

In addition to the foregoing, conventional physical compression techniques might also be used. *Front compression* is an example (see Chapter 2); since the field values in any given Field Values Table column are sorted into sequence, front compression can be applied directly if desired (though I should point out that such compression will complicate the binary search and record reconstruction tasks somewhat). What's more, the row ranges in any given Field Values Table column are sorted too (more precisely, they are in ascending sequence by either the range begin points or the range end points), and they can therefore be compressed as well.

Net Effect

The net effect of all of the above is that the Field Values Table is always memory-resident, at least to a first approximation. Startling though this claim might appear at first sight, on reflection it should seem plausible enough; after all, how many distinct attribute values do real databases actually contain? (And note that this rhetorical question appeals only to the idea of logical compression. Physical compression can only make the situation better.) *Note:* Even in those rare cases when the Field Values Table is *not* 100 percent memory-resident, there are still efficient ways of accessing it on the disk. Details of what's involved in such cases are beyond the scope of this chapter, however.

So—assuming that the Field Values Table is indeed memory-resident—we've now solved one of our two original problems: All binary searches on columns of that table will be done in memory, not on disk.

There are a few further points I want to make to close this section.

- First, as we'll see in the next section, reducing the size of the Field Values Table reduces the number of bits needed to represent pointers into that table as well. In other words, reducing the size of the Field Values Tables reduces the size of the Record Reconstruction Table as well.

- Second, in computing the size of a given Field Values Table, we ought by rights to take the space required by the row ranges into account as well (even though it's likely that those row ranges will be physically stored separately from the field values per se). Now, if Field Values Table column *C* contains *N* values, then the corresponding row ranges will require a total of *N* log *N* bits—probably much less space than the *N* values themselves require. Perhaps we might say to a first approximation that row ranges cause the overall size of the Field Values Table to double, though I think their effect is likely to be much less than that in practice. But it's simpler—given that I'm usually not trying in this book to do precise analyses—just to assume that if the Field Values Table without row ranges is small and can fit into memory, then the Field Values Table with row ranges is also small and can fit into memory too. In other words, I'm going to ignore the space required for row ranges from this point forward. I don't believe this simplifying assumption has any material effect on any of the arguments to come.

- Third, I've said the Field Values Table is stored column-wise. Now, in Chapter 4 I mentioned the fact that certain other systems, both prototypes and commercial products, "store the data attribute-wise." The two notions aren't directly comparable, though. To be specific, in TR we're not really talking about storing some attribute of some user-level relation at all; rather, we're talking about storing some condensed, merged, and possibly otherwise transformed column of the Field Values Table,[4] and as we've seen there's no direct correlation (in general) between a user-level attribute and a Field Values Table column.

11.5 Compressing the Record Reconstruction Table

The subfield encoding example in the previous section illustrates a point that you might or might not have realized for yourself, but is in any case worth calling out explicitly. To be specific: *In any real database, the amount of space required for the Field Values Table is likely to be negligible compared to the space required for the Record Reconstruction Table*. In other words, the Record Reconstruction Table in any real database is likely to be orders of magnitude bigger than the Field Values Table, and so we'd definitely like to find ways to compress it if we can. The trouble is, the Record Reconstruction Table contains what are in effect *permutations,* and permutational data is notoriously hard to compress. Even so, there are some useful things we can do ... This time I'd like to discuss physical compression techniques first.

Physical Compression

My first point has to with **TR pointer size**. As I explained in Chapter 2, the pointers we're talking about, though conceptually addresses, certainly aren't physical addresses, neither on disk nor in memory. In TR, in fact, they aren't even of constant size—they aren't all 32 bits in length, for example. Rather, the pointers within any given Record Reconstruction Table are *just as big as they need to be*. For example, given the Record Reconstruction Table of Fig. 11.2 (a copy of the Record Reconstruction Table from Fig. 4.3 in Chapter 4), it's clear that there are only five different pointer values, and three bits are thus sufficient to represent any of them.

	1	2	3	4	
	S#	SNAME	STATUS	CITY	1
1	5	4	4	5	
2	4	5	2	4	
3	2	2	3	1	
4	3	1	1	2	
5	1	3	5	3	

Fig. 11.2: Record Reconstruction Table for the suppliers file of Fig. 4.1

Recall now that (of course) the Field Values Table is condensed. If we expand the Record Reconstruction Table of Fig. 11.2 to include direct pointers into (say) the CITY column of the condensed Field Values Table, then those pointers will require only two bits, not three, because there are only three distinct CITY values and not five. More realistically, suppose there were 100,000 rows in the uncondensed Field Values Table but only 20 distinct CITY values; then the pointers we're talking about would require only five bits instead of the 17 they would otherwise require (2^{17} = 131,072).[5] In general, the space saving could be considerable (over 70 percent, in this particular example).

Like other data, pointers in TR are bit-aligned on the disk (in particular, therefore, they aren't necessarily even byte-aligned, let alone word-aligned).

While I'm on the subject of pointer size, let me explain something else that might possibly have been bothering you. Suppose we're using an overflow structure to hold newly inserted values as described in Chapter 6 (Section 6.5). Then pointers in that overflow structure don't necessarily have to be the same size as their counterparts in the main database. Suppose, for example, that a given field in the main database contains exactly 128 distinct values, so that associated pointers are just seven bits, and then a new 129th value is inserted (implying that seven bits are no longer adequate). Then pointers in the overflow structure might have to be eight bits—or not, as the case may be—but pointers in the main database won't have to change in size until such time as the merging process is done (that is, until the overflow structure is merged in with the main database, as described in Chapter 6, Section 6.5).

Back to physical compression techniques for the Record Reconstruction Table. Here are some relevant considerations.

- First of all, we don't have to include those direct pointers from the Record Reconstruction Table into the Field Values Table anyway; they're there (as explained in Chapter 8) merely to speed up the record reconstruction process, and they aren't logically necessary. So we could delete them if desired (and we probably would, on the disk).

- Second, even if we do include those direct Field Values Table pointers after all, we can at least apply (for example) front compression to them, since their values within any given Record Reconstruction Table column are at least guaranteed to be in ascending sequence.

- Third, we've seen that compressing the Field Values Table has the desirable side-effect of reducing the size of those direct Field Values Table pointers anyway.

- Fourth, suppose the Record Reconstruction Table is a "cyclic" one (refer to Chapter 7, Section 7.5, for an explanation of this term). Then, within any given column of that table, the zigzag pointers corresponding to a given field value within the Field Values Table are also guaranteed to be in ascending sequence; they can therefore also be compressed.

However, despite all of the above, the fact remains that the Record Reconstruction Table is still likely to be quite large in practice. By way of an example, suppose we start with a user-level relation of ten attributes and 200 million tuples. Suppose we decide not to include direct pointers from the Record Reconstruction Table into the Field Values Table; for simplicity, however, suppose also that the table isn't a cyclic one, and so the compression techniques applicable to such tables aren't available. Then we're going to need a total of two billion pointers of 28 bits each, for a grand total of seven billion bytes. What can we do about this problem?

Logical Compression

Before I attempt to offer an answer to the question just posed, let me first say a little more about the problem of zigzags on the disk.

Now, I've already explained that the Record Reconstruction Table is stored column-wise on the disk. By way of example, consider Figs. 11.3 and 11.4, which show the Field Values Table for the parts relation from Chapter 8 and a corresponding Record Reconstruction Table (the figures are identical to Figs. 8.6 and 8.4, respectively, in Chapter 8). To keep the example simple, I've omitted the direct pointers from the Record Reconstruction Table into the Field Values Table.

	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	P1	Bolt [1:1]	Blue [1:2]	12.0 [1:2]	London [1:3]
2	P2	Cam [2:2]	Green [3:3]	14.0 [3:3]	Oslo [4:4]
3	P3	Cog [3:3]	Red [4:6]	17.0 [4:5]	Paris [5:6]
4	P4	Nut [4:4]		19.0 [6:6]	
5	P5	Screw [5:6]			
6	P6				

Fig. 11.3: Field Values Table for parts

	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	4	3	2	1	1
2	1	1	4	6	4
3	5	6	5	2	6
4	6	4	1	4	3
5	2	2	3	5	2
6	3	5	6	3	5

Fig. 11.4: Record Reconstruction Table for parts

Now consider the query "Get all red parts." In order to implement this query, the system will do an in-memory binary search on the COLOR column of the Field Values Table and will discover that the corresponding row range is [*4:6*]. Then it'll go to the COLOR column of the Record Reconstruction Table and chase three zigzags, beginning at cells [*4,3*], [*5,3*], and [*6,3*], respectively. (Recall that in the subscript expression [*i,j*], *i* is a row number and *j* is a column number.)

From this example, we can see that we certainly want to store cells [*4,3*], [*5,3*], and [*6,3*] contiguously in storage; that is, column-wise storage for column CITY of the Record Reconstruction Table is obviously desirable. And, of course, analogous arguments show that column-wise storage is desirable for every column of that table.

But the problem is, even if (in terms of our example) we *start* chasing the zigzags from contiguous locations, we very quickly find ourselves performing essentially random lookups "all over the disk." Indeed, the three zigzags actually look like this in the example:

- `[4,3], [1,4], [1,5], [1,1], [4,2]`
- `[5,3], [3,4], [2,5], [4,1], [6,2]`
- `[6,3], [6,4], [3,5], [6,1], [3,2]`

In other words, although the starting points are physically contiguous, the zigzags quickly splay out to what are essentially random positions within the Record Reconstruction Table—effectively implying a separate seek and read operation for every point after the starting point in each zigzag, if the zigzag in question isn't in memory at run time.

So reducing the size of the Record Reconstruction Table (so that the zigzags can be in memory at run time after all) is highly desirable. Such is the aim of **file factoring**. File factoring can be regarded as a highly effective logical compression technique—so effective, in fact, that it's likely to mean that large portions, at least, of the Record Reconstruction Table will be memory-resident after all in any real database. And if we can achieve this desirable goal, we'll have solved the other of our two original problems: All zigzagging through that table will be done in memory, not on disk.

File factoring is described in detail in the next chapter.

11.6 Minimizing Seeks

In this section I want to consider, very briefly, what happens if the techniques described in previous sections aren't sufficient to get everything into memory. If that's the case, then we'll still have to perform some degree of disk access at run time, and (as we saw in Section 11.3) we clearly want to keep the amount of seek activity involved in that process to a minimum.

Now, I listed a variety of techniques in Section 11.3 for reducing run-time seeking. Just to remind you, here's that list again:

- Large page sizes
- Streaming data off the disk
- Storing data column-wise
- Banding
- Using stars instead of zigzags
- Controlled redundancy

Of these six items, I've said as much as I'm going to say regarding the first three. The remainder of this section presents a brief overview of the rest.

Banding

For simplicity, I've tended to talk in this book in terms of "the" Field Values Table and "the" Record Reconstruction Table, as if there were just one of each. In practice, of course, there'll be not one but many of each; loosely speaking, there'll be one of each for each user-level relation—though as we already know, in the case of the Field Values Table(s) in particular, the picture is complicated somewhat by the possibility (or likelihood, rather) of column merging and certain other features of the TR model.[6] However, there'll certainly be many Record Reconstruction Tables, in general, and *banding* will lead to more.

Banding is an attack on the problem of zigzags that splay out all over the disk. The basic idea is to split the original file (conceptually) into a set of horizontal *bands,*[7] and then to treat each such band as a file in its own right, with its own TR-level representation. In other words, each band will have its own Field Values Table and its own Record Reconstruction Table—implying in particular that zigzags within any given Record Reconstruction Table will be wholly contained within the relevant band. Band size is chosen such that any given band will fit entirely into memory at run time, and bands are laid out on the disk in such a way as to facilitate streaming data off the disk. See Chapter 13 for further discussion.

Using Stars Instead of Zigzags

Like banding, *stars* too are an attack on the problem of zigzags that splay out all over the disk. Recall from Chapter 5, Section 5.8, that the linkage information that ties together the field values for a given record doesn't have to be implemented as a zigzag specifically—other possibilities exist, and *stars* are one such. Basically, stars are functionally equivalent to zigzags but have different performance characteristics. In particular, they avoid the splay problem and thus reduce the amount of random seeking required. See Chapter 14 for further discussion.

Controlled Redundancy

Banding and stars both have the property that access based on one particular field, the so-called *characteristic* (or *core*) field, will perform better than access based on any other; that is, access via any field other than the characteristic one will involve more seeks than access via the characteristic one. In other words, as noted in Section 11.3, symmetry of performance will be lost (see Chapter 5, Section 5.2, for a discussion of this notion). We can address this problem by introducing a degree of controlled redundancy into the storage structures. See Chapters 13 and 14 for further discussion.

Endnotes

1. The term *megabyte* is sometimes defined to mean exactly one million bytes, sometimes 2^{20} = 1,048,576 bytes. Similarly, the term *kilobyte* is sometimes defined to mean exactly one thousand bytes, sometimes 2^{10} = 1,024 bytes. The differences aren't significant for our purposes.
2. OLTP = online transaction processing.
3. I've no idea how realistic the numbers are that I'm using in this example, but they're good enough to illustrate the point I want to make.
4. Or some column of the Record Reconstruction Table, since that table is also stored column-wise—but here the parallel with conventional attribute-wise storage is even weaker.
5. I note in passing that the pointers we're talking about here (namely, the ones appearing "first" in each Record Reconstruction Table cell) act as surrogates for field values in exactly the manner explained in Chapter 5, Section 5.6. The others (the ones appearing "second" in each such cell) can be regarded as surrogates too, but the decoding mechanism by which the field values are obtained from those surrogates is slightly different in the latter case.
6. In the extreme, in fact, there could be just one Field Values Table after all. I'll discuss this possibility further in Chapter 15 (Section 15.2).
7. "Horizontal" because the splitting occurs "between records," as it were.

12 File Factoring

12.1 Introduction

We saw in Chapter 11 that it would be good to reduce the size of the Record Reconstruction Table. File factoring, or just *factoring* for short, is a technique for achieving this goal. (As mentioned in the previous chapter, it can have the effect of reducing the size of the Field Values Table as well; however, it's the effect on the Record Reconstruction Table that's the real point.) Here in outline is how it works:

- Starting with a given user-level relation, and hence a corresponding file, we decompose that file "vertically" into two or more **subfiles** (the official term is *factors,* but there's a good reason, which I'll explain in Section 12.3, for preferring the term *subfiles*). Each subfile is smaller than the original file, in the sense that it has fewer fields, and possibly fewer records, than the original file. *Note:* The term *vertical decomposition* refers to the fact that the decomposition is done "between fields," as it were.

- We then map each of those subfiles into its own Field Values Table and Record Reconstruction Table. Because the subfiles are smaller than the original file, those Field Values and Record Reconstruction Tables are smaller than their counterparts would have been for the original file. In particular, the Record Reconstruction Tables involve fewer pointers than their original counterpart would have done, a fact that can have dramatic effects on overall space requirements, as we'll see.

In effect, therefore, factoring replaces large tables by smaller ones, such that the total space required for the smaller ones is less—usually *much* less, in practice—than that required for the large ones. As I claimed in the previous chapter, it can thus be seen as a logical compression technique: logical, because the compression in question is performed (conceptually, at least) at the file level, before we even begin to think about the question of mapping the data to disk. The net effect is:

a) To make a larger portion of the data—in particular, a larger portion of the Record Reconstruction Table—permanently memory-resident, and

b) To pack more useful data into each page on the disk, thus providing "more bang for the buck" on each I/O operation.

The structure of this chapter is as follows. Following this introductory section, I'll explain the basic idea of factoring by means of a simple example in Section 12.2; then I'll elaborate on and generalize from that example in Sections 12.3 and 12.4. In Section 12.5, I'll explain what's involved in doing record reconstruction with factored files. Finally, in Section 12.6, I'll point out some additional benefits of factoring, over and above the overriding one of reducing Record Reconstruction Table space requirements.

12.2 A Simple Example

I have a problem. By definition, the techniques to be discussed in this chapter (also in the next two) are intended for dealing with very large data sets, with raw data space requirements measured in the billions of bytes or even more. (Actually the same was true throughout Part II of the book, but it's even more true here.) For obvious reasons, however, I can't show examples that involve such very large data sets. In what follows, therefore, you'll have to exercise your imagination a little; to be specific, you'll have to extrapolate from very small examples to the very large databases that actually exist in the real world.

I'll build on the parts example from Chapter 8 (see Fig. 8.1 in that chapter). Just to remind you, the relation P in that example originally had five attributes, as follows:

Part number:	`P#`
Part name:	`PNAME`
Color:	`COLOR`
Weight:	`WEIGHT`
Location:	`CITY`

Now let's extend it to include some additional ones—let's say as follows:

State:	`STATE`
Zip code:	`ZIP`
Phone number:	`PHONE#`

In practice there might well be other attributes too—for example, part description, street address, and so on—but the eight listed above are sufficient for our purposes.

Perhaps I should explain the semantics a little, in order to make the example a little more intuitively acceptable. Essentially, I'm taking the combination of CITY, STATE, and ZIP to be an elaboration of the old CITY attribute; I'm assuming that this combination of attributes identifies the location of the (sole) warehouse where parts of the indicated kind are kept. PHONE# gives the (sole) phone number for that warehouse. Also, I'll assume for simplicity that STATE always identifies a state in the U.S., and ZIP is thus always a U.S. zip code—and I'll stick to five-digit zip codes, again for simplicity. (By the way, did you know that *zip* is an acronym? It stands for *zoning improvement plan*.)

Let's assume further that there are ten million different parts, and hence ten million tuples in the parts relation and ten million records in the corresponding parts file. *Note:* I'll stick to this particular assumption throughout this chapter, and indeed throughout the next two as well.

Now, the basic parts Record Reconstruction Table is isomorphic to the parts file (that is, it has the same number of rows and columns as that file has records and fields, respectively). So that Record Reconstruction Table now has 80 million cells, and hence 80 million pointers. Each pointer in turn is 24 bits, and so the total space requirement is 240 megabytes (240MB). *Note:* If the Record Reconstruction Table were expanded to include direct pointers into the Field Values Table as well, the space requirement would double, to 480MB; for simplicity, for simplicity, however, let's omit these latter pointers. (Actually the space requirements wouldn't *exactly* double, because the Field Values Table would be condensed and the pointers into it would therefore be less than 24 bits. As I say, however, I'm going to ignore those pointers anyway.)

Assume now for the sake of the example that for any given zip code, there's just one city and state; that is, if *z* is a zip code and *c* and *s* are the corresponding city and state, then, whenever a tuple of the original parts relation P has ZIP = *z,* it also has CITY = *c* and STATE = *s*. In other words, there's a *many-to-one relationship* from ZIP to CITY and STATE: Many zip codes can have the same city and state, but no zip code can have more than one city and state (but see the next section). Formally, we say there's a **functional dependency** from ZIP to CITY and STATE, and we express it thus:

```
{ ZIP } → { CITY, STATE }
```

The general form is LHS → RHS; you can read it as "the right-hand side (RHS) is *functionally dependent* on the left-hand side (LHS)" or, more simply, just as "left-hand side arrow right-hand side." By convention, we enclose the left- and right-hand sides in braces because they're both *sets* of attribute names.

Now, it follows from the existence of the functional dependency from ZIP to CITY and STATE that the parts relation P contains a great deal of redundancy. After all, there are ten million distinct tuples, but there certainly aren't ten million distinct zip codes. In fact, I have it on good authority that there are around 38,000 of them (for the whole of the U.S., that is)—but to keep the arithmetic simple, let's round that figure up to 40,000. On average, then, there'll be 250 distinct tuples in the relation for any given zip code, and all 250 of those tuples will contain precisely the same values for ZIP, CITY, and STATE (there's the redundancy).

An obvious factoring thus suggests itself: Starting with the original parts file, let's decompose it vertically into two subfiles, with fields as indicated below:

```
Subfile 1     Subfile 2

P#            ZIP
PNAME         CITY
COLOR         STATE
WEIGHT
ZIP
PHONE#
```

For example, if the original file included a record looking like this—

P#	PNAME	COLOR	WEIGHT	CITY	STATE	ZIP	PHONE#
P8	Wheel	Black	15.0	Rome	GA	*zzz*	*nnnnnn*

(I've shown the ZIP and PHONE# values symbolically for simplicity)——then the two subfiles will include records looking like this:

Subfile 1

Subfile 1

P#	PNAME	COLOR	WEIGHT	ZIP	PHONE#
P8	Wheel	Black	15.0	*zzz*	*nnnnnn*

Subfile 2

ZIP	CITY	STATE
zzz	Rome	GA

There'll be one record in Subfile 1 for each part number (10 million records), and one record in Subfile 2 for each zip code (40,000 records). Of course, the original file can be reconstructed from the two subfiles by "joining" them back together on the ZIP field ("joining" in quotes because, strictly speaking, join is an operation that applies to relations, not to files).

So Subfile 1 has ten million records and six fields, while Subfile 2 has 40,000 records and three fields. Each subfile has its own Record Reconstruction Table. The first has 60 million pointers, still 24 bits each, for a total of 180MB. The second, however, has only 120,000 pointers, and those pointers are only 16 bits each, for a total of only 240KB (*kilobytes,* not megabytes); in fact, the space required for the second Record Reconstruction Table is negligible compared to that required for the first. The net effect is that we've reduced overall space requirements by around 25 percent.

The foregoing example illustrates the basic idea of factoring. Of course, there's still quite a lot more to be said, but first let me call out a few explicit points here:

- For simplicity I'll assume throughout this chapter that factoring always decomposes a given file into exactly two subfiles, as in this first example (barring explicit statements to the contrary, of course).

- I'll also assume that one of those subfiles is the "large" subfile and the other is the "small" subfile, and I'll refer to them as such (or sometimes as simply the large and small *files*, because of course a subfile can be regarded as a file in its own right—that's why I prefer the term *subfile* over the term *factor*). And I'll refer to the corresponding Field Values and Record Reconstruction Tables as "large" and "small" accordingly.

- *Very important:* The small Record Reconstruction Table will usually be *much* smaller than the large one—as indeed it was in our example—and can therefore be memory-resident. This is the basic object of the exercise, of course.

- *Also very important:* Factoring does **not** have to be done "by hand" (as it were). Rather, it's done automatically during the load process, on the basis of certain heuristics that are built into the load utility and various statistical analyses of the data that are also performed at load time. In other words, the benefits of factoring are obtained automatically, without any need for human decisions (on the part of the database administrator in particular).

Note finally that factoring as described above conceptually leads to two separate Field Values Tables, as well as two separate Record Reconstruction Tables. However, those two Field Values Tables can then be merged back into one as described in Chapter 9. In a sense, file factoring might be thought of as a kind of inverse of column merging as described in Chapter 9; column merging means two or more files map to one Field Values Table, loosely speaking, while file factoring means one file maps to two or more Field Values Tables (but those Field Values Tables are then merged back into one anyway, as we've just seen).

12.3 Elaborating on the Example

In the example in the previous section, we decomposed the original file on the basis of a certain functional dependency (FD for short). For that very reason, however, readers knowledgeable in database matters might have found the example a little unconvincing: Surely the database designer would already have performed the indicated decomposition at the relational level, precisely because of the existence of that FD? Indeed, such "decomposition at the relational level" is exactly what the business of *further normalization* is all about—see, for example, reference [32]. And if the designer had indeed already performed that decomposition at the relational level, then we would have started off with two distinct relations, and hence two distinct files at the file level, and the question of automatic decomposition of a single file into two distinct

subfiles would never have arisen.

There are several possible responses to this objection, however. Four such are explained in the subsections immediately following.

"Denormalize for Performance"

The first point is that, in practice, the database designer might very well *not* have already performed the indicated decomposition at the relational level after all. The reason is that—at least in today's mainstream systems—full normalization is often contraindicated, because the direct-image nature of those systems can give rise to performance problems with fully normalized designs. The usual argument goes something like this [27]:

1. Full normalization means lots of logically separate relations at the relational level.

2. Lots of logically separate relations at the relational level means lots of physically separate stored files at the storage level.

3. Lots of physically separate stored files means lots of I/O.

For example, given our usual suppliers, parts, and shipments relations, a request to find London suppliers who supply red parts will involve two joins: First, join suppliers to shipments (say); second, join the result to parts. (I'm ignoring the two restriction operations for simplicity.) And if the three relations in fact do map to three physically separate stored files as suggested, then those two joins will indeed require lots of I/O and will therefore perform badly. Hence the cry we've doubtless all heard so many times: "Denormalize for performance!"

Note: Just in case you *haven't* heard this cry before, let me elaborate briefly. First, of course, normalization is a logical design discipline for reducing redundancy at the user or relational level. The trouble is, normalization leads to an increase in the number of relations and hence to an increase in the number of joins required in queries; and (to repeat) in a direct-image system, that increased number of joins translates directly into a performance hit. Denormalization is an attempt to fix this latter problem. (Note, however, that denormalization, unlike normalization, can hardly be described as a *discipline,* being in fact totally ad hoc.) Denormalization decreases the need for joins by decreasing the number of relations. Unfortunately, of course, it also increases the degree of redundancy—with negative consequences for updates, and even for some queries. In my opinion, applying a user-level fix (denormalization) to an internal-level problem (performance) cannot, by definition, be the best solution to that problem—but in direct-image systems, it might be the only solution available.

So the foregoing argument—the argument, that is, that the designer might not have already performed the decomposition at the relational level—is valid, more or less, in a direct-image system. However, it certainly isn't valid in a TR system; in a TR system, relations don't map directly to physical files, and joins are cheap. In a TR system, in fact, we really can, and should, go for fully normalized designs at the relational level (I'll come back to this point in Chapter 15). Thus, this first response to the objection that the example of the previous section wasn't very convincing is perhaps not a very strong one, given the TR context. So let's move on quickly to the second response ...

Normalization Is Based on* Relevant *FDs

There's a popular misconception in the IT community at large to the effect that logical database design requires normalization to be performed on the basis of *all* FDs. In fact, of course, such is not the case; rather (as I've written elsewhere [32]), normalization should be performed on the basis of all *relevant* FDs, not on the basis of all FDs that happen to exist. In the case of the parts relation, for example, with its attributes ZIP, CITY, and STATE (among others), the database designer might well decide that the FD

```
{ ZIP } → { CITY, STATE }
```

isn't very relevant to the problem at hand, and hence that decomposition at the relational level on the basis of that particular FD is hardly worthwhile. After all, CITY and STATE are almost invariably required together (think of printing a mailing list, for example); what's more, zip codes don't change very often, and thus there doesn't seem to be much to be gained by conventional normalization on the basis of that FD. Indeed, there might even be something to be lost; certainly conventional normalization will make some queries a bit more complex (from the user's point of view, that is), because they'll involve an additional join.

So we've arrived at the notion that the data might satisfy certain FDs that the database designer didn't use as a basis for normalization and (in all probability) didn't even declare to the DBMS. Conceivably, in fact, the data might satisfy certain FDs that the designer wasn't even aware of—but the load process can still detect such FDs and use them to perform file factoring. Thus, file factoring can apply even when normalization might have been applied in the first place but in fact wasn't, for some reason.

Factoring Based on Approximate FDs

The third response is this (and it's an important one): *File factoring can be based on "approximate FDs" as well as on genuine ones*. For example, I've been assuming up until this point that the FD

```
{ ZIP } → { CITY, STATE }
```

holds true, but in the real world it doesn't—not quite. Let me explain. Recall first that this FD effectively asserts that no zip code ever corresponds to more than one city and state combination, or in other words that distinct city and state combinations always have distinct zip codes. Well, there are exceptions; for example, the cities of Jenner and Fort Ross in California both have zip code 95450. (This kind of thing can happen if a zip code is assigned to some region and then part of that region subsequently incorporates and becomes a separate city in its own right.) Thus, a more accurate statement is that the "FD" (in quotes because it isn't really an FD at all)

```
{ ZIP } → { CITY, STATE }
```

almost holds true ... but that *almost* means we can't use the "FD" as a basis on which to perform normalization. For suppose our original parts relation looked like this:

P#	....	CITY	STATE	ZIP	....
P88	....	Jenner	CA	95450	....
P99	....	Fort Ross	CA	95450	....

Now suppose we decompose it into two projections as follows:[1]

P#	....	ZIP	....
P88	....	95450	....
P99	....	95450	....

ZIP	CITY	STATE
95450	Jenner	CA
95450	Fort Ross	CA

Now we have *ambiguity:* We can't tell which parts are kept in which city (note that if we join the two projections back together, we'll get four tuples, not two). In other words, the decomposition has lost information. (To be valid for normalization, of course, we do require decompositions to be "nonloss" [32].)

However, the fact that we can't do normalization in this example doesn't mean we can't do factoring. In fact, there are at least two ways to do it, and I'll sketch them both briefly here.

The first involves introducing an artificial identifier, ZCS# say, for each distinct zip / city / state combination. Thus, if the original parts file looked like this—

P#	....	CITY	STATE	ZIP	....
P88	....	Jenner	CA	95450	....
P99	....	Fort Ross	CA	95450	....

—we might replace it by the following two subfiles:

P#	....	ZCS#	....
P88	....	*zcs8*	....
P99	....	*zcs9*	....

ZCS#	ZIP	CITY	STATE
zcs8	95450	Jenner	CA
zcs9	95450	Fort Ross	CA

As you can see, the artificial identifier ZCS# plays a role analogous to that played by candidate and foreign keys in the relational model—it's a candidate key for the small subfile and a corresponding foreign key in the large one, loosely speaking. (I say "loosely speaking" because in fact such artificial identifiers *aren't* keys in the relational sense; relational keys apply by definition to relations at the user level, while the artificial identifiers apply to files at the file level. But the parallel is exact.)

So what does the foregoing trick do to our storage requirements? The large Record Reconstruction Table still has 60 million pointers of 24 bits each, for a total of 180MB. However, the small one has only 160,000 pointers of 16 bits each, for a total of only 320KB. (I'm making the reasonable assumption that the small subfile still has around 40,000 records, even though zip codes aren't unique. The point is, they're *almost* unique.) As in Section 12.2, therefore, the space required for the small Record Reconstruction Table is negligible, and the net effect is that we've reduced overall space requirements by around 25 percent.

Note: Values of the artificial identifier ZCS# could even be *direct pointers* into the small Record Reconstruction Table. Certainly they can be just 16 bits, like the pointers in that table. I'll have more to say about this possibility in Section 12.5.

The second approach to factoring using an "approximate FD" is to pretend the FD is genuine, moving the rare exceptions out into a special file of their own. Thus, the vast majority of records in the original parts file can be treated exactly as in Section 12.2. When a situation arises like that with zip code 95450 in our example above, we treat one of the pertinent zip / city / state combinations in the usual way, and move the others out into the special file. The result might look like this in our example:

Large subfile

P#	....	ZIP	....
P88	....	95450	....

Small subfile

ZIP	CITY	STATE
95450	Jenner	CA

Special-case subfile

P#	ZIP	CITY	STATE
P99	95450	Fort Ross	CA

Part numbers are unique in the large subfile; zip codes are unique in the small subfile; and part numbers (again) are unique in the special-case subfile. Moreover, no part number appears in both the large subfile and the special-case subfile. *Note:* It's true that access by part number is now more complicated than it was before, but at least the special-case Record Reconstruction Table will be small enough to be memory-resident, as we'll see in just a moment.

Let's do the storage arithmetic again. Suppose one tenth of one percent of the original ten million part records (in other words, 10,000 part records in total) have to be treated as special cases in the foregoing sense. Then the large Record Reconstruction Table still requires approximately 180MB. The small one still requires approximately 240KB. And the special-case one has 40,000 pointers of 14 bits each, for a total of 70KB. Once again, the large table is the significant one, and once again the net effect is that we've reduced overall space requirements by around 25 percent.

Incidentally, while I'm on the subject of factoring on the basis of approximate FDs, there's one pragmatically important special case to consider, as follows. Let *F2* be a field in some file, and let *F2* be "of low cardinality" (meaning it doesn't contain many distinct values compared to the total number of records in the file overall).[2] Then it's necessarily "almost" true that the FD

```
{ F1 } → { F2 }
```

holds true for all fields *F1* in that same file, and the factoring techniques described in the present subsection are thus directly applicable.

Following on from the previous point, let me now add that there's also one important special case of the general idea of a field being of low cardinality, and that's the case in which most of the values in the field are the same—for example, a numeric field in which most of the values are zero. Again, the factoring techniques described in this subsection are directly applicable.

Factoring Isn't Necessarily Based on FDs

The fourth and last response to the objection that the example of Section 12.2 makes little sense is simply this: While there are certainly parallels between factoring as described so far and conventional normalization—observe in particular that they both involve vertical decomposition—the truth is that factoring is, in a sense, more general than conventional normalization. In particular, factoring doesn't necessarily have to be based on functional dependencies as such[3] (nor even on "approximate" FDs). Rather, it can be based on *absolutely any kind of statistical pattern or "clumpiness"* in the data whatsoever. The section immediately following describes some of these further possibilities.

12.4 Further Possibilities

Suppose the parts file has already been factored as described in the previous section—in the subsection entitled "Factoring Based on Approximate FDs"—to yield subfiles that look like this:

```
Large subfile  Small subfile

P#             ZCS#
PNAME          ZIP
COLOR          CITY
WEIGHT         STATE
ZCS#
PHONE#
```

As we've seen, the Record Reconstruction Table for the large subfile still requires a fairly hefty 180MB. What can we do to reduce this space requirement still further?

Well, suppose now, not at all unrealistically, that there are very few distinct color / weight combinations in the large file (equivalently, in the original unfactored file). For the sake of the example, suppose there are just 500 such combinations, corresponding to (perhaps) ten different colors and 50 different weights. Then the combination of COLOR and WEIGHT behaves like a low-cardinality field, and we can deal with it accordingly. To be specific, we can introduce an artificial identifier, CW# say, for each distinct color / weight combination, and factor the large subfile above into two further subfiles that look like this:

```
Large subfile     Small subfile (second level)
P#                CW#
PNAME             COLOR
CW#               WEIGHT
ZCS#
PHONE#
```

Note: Since the file we're factoring here is itself already a subfile, we might say we're performing *hierarchic* factoring in this particular example. The possibility of hierarchic factoring is, of course, a natural consequence of the fact that a subfile can be regarded as a file in its own right, and it's another reason why I prefer the term *subfile* to the term *factor*. The fact that a subfile is a file is important for exactly the same kinds of reasons that the fact that a subset is a set is important in mathematics.

Anyway, here's the storage arithmetic. The large Record Reconstruction Table now has only 50 million pointers instead of 60 million, reducing the space requirement for that table from 180MB to 150MB. The small one has just 4,500 pointers of just nine bits each, for a total of 4,950 bytes, which we can ignore completely. Once again the large table is the only important one, and now we've reduced overall space requirements by around 37.5 percent (37.5 percent of the original requirement of 240MB, that is, as calculated near the beginning of Section 12.2).

We can do better. Instead of factoring out the zip / city / state combination and the color / weight combination separately, why not factor them out *together?* The subfiles might look like this (note the artificial identifier ZCSCW#):

```
Large subfile     Small subfile

P#                ZCSCW#
PNAME             ZIP
ZCSCW#            CITY
PHONE#            STATE
                  COLOR
                  WEIGHT
```

For simplicity, let's abbreviate the expressions zip / city / state and color / weight to ZCS and CW, respectively. By our assumptions, then, there are approximately 40,000 distinct ZCS values and 500 distinct CW values; thus, there are at most 20 million distinct ZCS / CW combinations, and hence at most 20 million records in the small subfile. In practice, of course, it's impossible that all 20 million combinations actually exist (after all, there were only 10 million part records to start with); for the sake of the example, therefore, let's suppose that just one million combinations actually do exist. Now let's do the storage arithmetic again. The large Record Reconstruction Table now has only 40 million pointers, reducing the space requirement for that table still further to 120MB. The small one has six million pointers of 20 bits each, for a total of 15MB. The grand total is thus 135MB, a saving of around 43.75 percent over the original.

Yet another possibility is to break fields up into **subfields** (conceptually speaking) and then to treat those subfields as fully-fledged fields in their own right in the factoring process.[4] In our running example, an obvious candidate for such treatment is the PHONE# field. The original parts file has 10 million phone numbers, but it can't have 10 million area codes; in fact, let's assume, as we did in Chapter 11 (Section 11.4), that there are just 250 distinct area codes. What's more, there's a high correlation between area codes and zip codes; in fact, let's assume, reasonably enough, that most zip codes have just one area code (in other words, there's an "approximate FD" from zip codes to area codes). So it makes sense to split the PHONE# field into AREA_CODE and REST, and then to factor out the AREA_CODE along with the ZCS and CW information like this (note the artificial identifier ZCSCWAC#):

```
Large subfile     Small subfile

P#                ZCSCWAC#
PNAME             ZIP
ZCSCWAC#          CITY
REST              STATE
                  COLOR
                  WEIGHT
                  AREA_CODE
```

Now, this example differs from previous ones in that it has no effect on the large Record Reconstruction Table; rather, its effect is on the large Field Values Table, whose space requirements are reduced by 30MB (ten million three-byte area codes). At the same time, it adds an AREA_CODE column to the small Field Values Table—but that column is condensed and requires only 750 bytes, which we can ignore. More important, it also adds a column of pointers to the small Record Reconstruction Table, for an additional 2.5MB. The net saving is thus 28.5MB. (I can't easily express this saving as a percentage, because now the Field Values Table is involved as well as the Record Reconstruction Table.)

By way of summary, then, the general principle that the foregoing examples illustrate (both in this section and in the previous one) is this:

- Let *F1, F2, ..., Fn* be distinct fields—possibly subfields—within some given file (where *n* is greater than one).
- Let the set of all of those fields be considered as a single combined field *F*.
- Let *F* have cardinality *c*.
- Then, whenever *c* is small compared to the total number of records in the file overall, it's worth factoring *F* out into a "small" subfile, leaving an identifier behind in the "large" subfile to serve as the necessary link to that small file. The identifier might be a user field or it might be an introduced artificial identifier, depending on circumstances.

Note in particular the requirement that *n* be greater than one. Clearly there's no point in factoring out just a single field, because if an identifier has to be left behind in the large file, then that large file will still have just as many fields as it had before.[5] To say it again, the major object of the overall factoring exercise is to reduce the size of the large Record Reconstruction Table, and the way to do that is to reduce the number of fields in the large file.

One last point to close this section: You might possibly be feeling there are some similarities between the notion of file factoring as described in this chapter and the notion of combined columns as discussed in Chapter 9 (Section 9.4)—and indeed there are some similarities. The primary objective in both cases is certainly to save space (and as a matter of fact, the savings obtained are comparable in the two cases). But there are some differences too, of course. In a nutshell:

- Combining columns is a technique for mapping two or more fields to the same column in the Field Values Table, where the fields in question are, in general, of different types (we aren't talking about *merged* columns here). This technique saves space in the Record Reconstruction Table by reducing the number of columns in the Field Values Table. However, it does make it more difficult to search on the basis of columns other than the leading one in any such column combination.
- By contrast, a large part of the point regarding file factoring is precisely that distinct columns in the Field Values Table *aren't* combined into one. Thus, searches on the basis of individual columns are still straightforward. (In fact, as we've seen, we might even want to split one column in the Field Values Table into two or more columns—or, more precisely, we might want to have two or more columns in the Field Values Table for one field in the user-level file.)

12.5 Record Reconstruction

What are the implications of factoring for the record reconstruction process? In order to consider this question, I think it's best to return to a much simpler example. Let's go all the way back to the original parts example from Chapter 8. Fig. 12.1, a copy of Fig. 8.2, shows a file corresponding to the parts relation of Fig. 8.1. *Note*: Of course, this example is really much too simple to illustrate the need for factoring or any of the potential benefits described elsewhere in this chapter. So I'll just have to ask you to suspend disbelief and work through the example with me anyway.

	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	P1	Nut	Red	12.0	London
2	P2	Bolt	Green	17.0	Paris
3	P3	Screw	Blue	17.0	Oslo
4	P4	Screw	Red	14.0	London
5	P5	Cam	Blue	12.0	Paris
6	P6	Cog	Red	19.0	London

Fig. 12.1: File corresponding to the parts relation of Fig. 8.1

Now let's assume we want to factor out the color / city combination. As explained in Section 12.3, we'll have to introduce an artificial identifier, CC# say, to link the resulting subfiles together, logically speaking. Thus, Figs. 12.2, 12.3, and 12.4 show, respectively, the subfiles that result from this factoring, the corresponding Field Values Tables, and the corresponding Record Reconstruction Tables. *Note:* With respect to the Field Values Tables in Fig. 12.3, I think it's clearer not to merge them together, though in practice they probably would be so merged; with respect to the Record Reconstruction Tables in Fig. 12.4, I think it's clearer to omit the direct pointers into the Field Values Tables that, again, they might contain in practice (or not, as the case may be—recall that those pointers are basically just an optional extra anyway).

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Nut	12.0	*cc1*
2	P2	Bolt	17.0	*cc2*
3	P3	Screw	17.0	*cc3*
4	P4	Screw	14.0	*cc1*
5	P5	Cam	12.0	*cc4*
6	P6	Cog	19.0	*cc1*

	1	2	3
	CC#	COLOR	CITY
1	*cc1*	Red	London
2	*cc2*	Green	Paris
3	*cc3*	Blue	Oslo
4	*cc4*	Blue	Paris

Fig. 12.2: Subfiles after factoring out COLOR and CITY

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Bolt [1:1]	12.0 [1:2]	*cc1* [1:3]
2	P2	Cam [2:2]	14.0 [3:3]	*cc2* [4:4]
3	P3	Cog [3:3]	17.0 [4:5]	*cc3* [5:5]
4	P4	Nut [4:4]	19.0 [6:6]	*cc4* [6:6]
5	P5	Screw [5:6]		
6	P6			

	1	2	3
	CC#	COLOR	CITY
1	*cc1*	Blue [1:2]	London [1:1]
2	*cc2*	Green [3:3]	Oslo [2:2]
3	*cc3*	Red [4:4]	Paris [3:4]
4	*cc4*		

Fig. 12.3: Field Values Tales for the subfiles of Fig. 12.2

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	4	4	1	1
2	1	2	6	4
3	6	6	2	6
4	5	1	4	2
5	2	3	5	3
6	3	5	3	5

	1	2	3
	CC#	COLOR	CITY
1	4	2	1
2	3	4	3
3	1	3	2
4	2	1	4

Fig. 12.4: Record Reconstruction Tables for the subfiles of Fig. 12.2

Now let's consider the problem of doing record reconstruction for, say, part records for parts in Oslo (I deliberately choose an example in which there's just one qualifying record, for simplicity). Noting that the CITY field is in the small subfile, we see that the sequence of events must be as follows:

- Search column CITY of the small Field Values Table in Fig. 12.3 for the specified value Oslo. We discover that the sole record we want passes through cell [*2,3*] of the small Record Reconstruction Table (row *2* because the row range for Oslo is [*2:2*], and column *3* because column CITY is indeed the third column of the small tables).

- Follow the zigzag passing through cell [*2,3*] of the small Record Reconstruction Table in Fig. 12.4. That zigzag looks like this:

  ```
  [2,3], [3,1], [1,2]
  ```

  The corresponding field values are:

  ```
  Oslo, cc3, Blue
  ```

- Now search column CC# of the large Field Values Table in Fig. 12.3 for the value *cc3* (in effect, we're using the "candidate key" value in the small Field Values Table to find the rows containing a corresponding "foreign key" value in the large Field Values Table). We discover that the sole record we want passes through cell [*5,4*] of the large Record Reconstruction Table.

- Follow the zigzag passing through cell [*5,4*] of the large Record Reconstruction Table in Fig. 12.4. That zigzag looks like this:

  ```
  [5,4], [3,1], [6,2], [5,3]
  ```

  The corresponding field values are:

  ```
  cc3, P3, Screw, 17.0
  ```

Reconstruction of the desired record is now complete. However, note the fact that we've had to traverse two separate zigzags, "hooking them together" (so to speak) by means of the artificial identifier CC#. Without going into details, it should be clear that we'd have to follow a similar procedure in order to reconstruct, say, part records for parts with weight 19.0, except that this time we'd have to use the "foreign key" value in the large Field Values Table to look up the corresponding "candidate key" value in the small Field Values Table, instead of the other way around (because WEIGHT is a field in the large subfile, not the small one).

One implication of the foregoing is, of course, that factoring can lead to some inefficiencies in the record reconstruction process. However, matters are not as bad as they might seem. As I pointed out in Section 12.3 (in the subsection "Factoring Based on Approximate FDs"), artificial identifier values—CC# values in the example—can be **pointers**. If they are, then instead of the associative lookup we had to do in the "WEIGHT = 19.0" example from the large Field Values Table to the small one, we can now *follow a pointer* directly from that large table to the small *Record Reconstruction* Table (bypassing the small Field Values Table entirely). Reconstruction "from the large to the small" will thus be more efficient.

What about the other direction (reconstructing "from the small to the large," as in the "CITY = Oslo" example)? Well, we can make this process more efficient too if we want, by carrying some additional row ranges in the small Field Values Table (in the CC# column of that table, to be specific). Fig. 12.5 shows what happens to the small Field Values Table in the example if we adopt this approach.

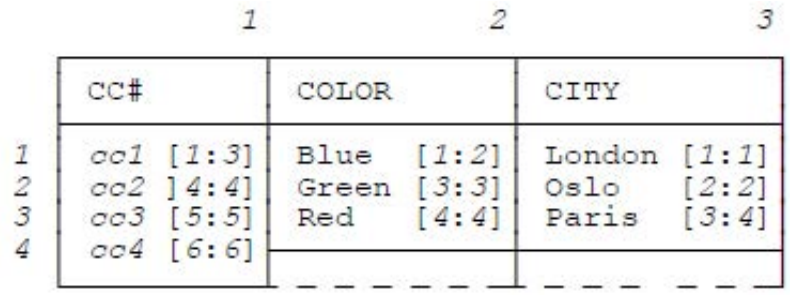


Fig. 12.5: Small Field Values Tables with CC# row ranges for the large Record Reconstruction Table

By way of example, consider cell [*1,1*] of the table in Fig. 12.5. That cell includes the row range [*1*:*3*]. That row range in turn shows that the rows in the large Record Reconstruction Table that correspond to CC# value *cc1* are rows *1, 2,* and *3*. Thus, instead of the associative lookup we previously had to do from the small Field Values Table to the large one, we can now follow pointers directly from that small table to the large Record Reconstruction Table (bypassing the large Field Values Table entirely). Reconstruction "from the small to the large" will thus also be more efficient than it was before.

12.6 Additional Benefits

The foregoing sections should be sufficient to give you some idea of the possibilities inherent in file factoring. For real databases, where relations often start out at the user level with many more than just eight attributes—and sometimes with many more than ten million tuples—the savings achievable are likely to be much greater than those shown in the examples (certainly more than the comparatively miserly percentages we saw in those examples). As I've said, factoring can be based on any kind of "statistical clumpiness" in the data whatsoever. And the fact is, the vast majority of data in the real world exhibits such "clumpiness" in great abundance; thus, the vast majority of data is a candidate for treatment by means of the techniques described in this chapter.[6]

In this final section, I'd like to point out that file factoring has certain additional benefits as well, over and above its primary one of reducing space requirements. To be specific, it offers certain benefits in connection with (a) aggregate queries and (b) update operations. Those benefits are the subject of the next two subsections. First, though, it's only fair to mention one potential drawback too: namely, that replacing one large Record Reconstruction Table by two or more smaller ones means we can't have a "preferred" Record Reconstruction Table (as described in Chapter 7) that provides major-to-minor orderings over all of the fields of the original file. However, the Record Reconstruction Tables for the subfiles—for the "small" subfile in particular—can still be "preferred" (as they are in Fig. 12.4, in fact), and in practice that's likely to be sufficient. Why? Because the fields of the small subfile are likely to be the ones over which orderings will most often be requested, as we'll see in the subsection immediately following.

Aggregate Queries

Aggregate queries—see the discussion of the relational operator SUMMARIZE in Chapter 10, Section 10.5—naturally tend to be framed in terms of fields in the small file, precisely because those fields are the low-cardinality ones. In SQL terms, for example, the following ("Sum weights by city") is certainly a realistic query on the parts relation P—

```
SELECT DISTINCT P.CITY, SUM ( P.WEIGHT ) AS SUMWT
FROM P
GROUP BY P.CITY ;
```

—whereas the following ("Sum weights by name") probably isn't:

```
SELECT DISTINCT P.PNAME, SUM ( P.WEIGHT ) AS SUMWT
FROM P
GROUP BY P.PNAME ;
```

(I'm relying here on the fact that part names are "almost unique.")

Let's assume once again that we're dealing with the original parts relation P with its attributes P#, PNAME, COLOR, WEIGHT, and CITY (only). Let's assume too (as in the previous section) that the file corresponding to that relation P has been factored as follows:

Large subfile	*Small subfile*
P#	CC#
PNAME	COLOR
WEIGHT	CITY
CC#	

If the parts file is as originally shown in Fig. 12.1, then here are the actual values of the two subfiles (repeated from Fig. 12.2):

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Nut	12.0	*cc1*
2	P2	Bolt	17.0	*cc2*
3	P3	Screw	17.0	*cc3*
4	P4	Screw	14.0	*cc1*
5	P5	Cam	12.0	*cc4*
6	P6	Cog	19.0	*cc1*

	1	2	3
	CC#	COLOR	CITY
1	*cc1*	Red	London
2	*cc2*	Green	Paris
3	*cc3*	Blue	Oslo
4	*cc4*	Blue	Paris

Observe now that if we partition the original file by COLOR and CITY, then each row in the small file corresponds to just one partition in the result. So it's not at all unreasonable to **precompute** certain aggregates for those partitions—compute them at load time, that is—and keep the results with the rows in the small file.[7] For example, if we were to treat *sums of weights* in this fashion, the small file might look like this:

	1	2	3	4
	CC#	COLOR	CITY	SW
1	cc1	Red	London	50.0
2	cc2	Green	Paris	17.0
3	cc3	Blue	Oslo	17.0
4	cc4	Blue	Paris	12.0

(I've shown the computed values as an extra field of the file, called SW. Of course, the computations aren't all that interesting in this particular example, but you get the general idea.)

Now suppose the user issues the SQL query from the start of this subsection ("Sum weights by city"):

```
SELECT DISTINCT P.CITY, SUM ( P.WEIGHT ) AS SUMWT
FROM P
GROUP BY P.CITY ;
```

At run time, then, instead of having to partition the large original file by CITY and do a series of possibly lengthy summations, the system can simply partition the *small* file by CITY and do a series of much shorter summations instead. *Note:* The savings can be particularly dramatic if there's a HAVING clause to eliminate certain of the partitions before the summations are done.

Analogous remarks apply to many other aggregate operators as well.[8] Note in particular that the technique could help in the case where the partitioning is done on the basis of a *subfield*. The parts example doesn't illustrate this point, but "Count subscribers in the 415 area code" might be an example of a query that could benefit from treatment similar to that described above.

Finally, ORDER BY requests too naturally tend to be framed in terms of fields in the small file, again because those fields are the low-cardinality ones. In SQL terms, for example, the following is certainly a realistic query on the parts relation P—

```
SELECT ...
FROM P
ORDER BY CITY, COLOR ;
```

—whereas the following probably isn't:

```
SELECT ...
FROM P
ORDER BY CITY, PNAME ;
```

The relevance of factoring here should be obvious; in fact, I explained it earlier, when I said that the small Record Reconstruction Table, at least, could still be a "preferred" one.

Update Operations

The point here is essentially a simple one: When a new record is inserted, it's likely that values in low-cardinality fields within that record will already exist in the small file, precisely because those fields *are* low-cardinality. For example, let *p* be a new part record. Then it's quite likely that *p* will involve a color and a city—and even a color / city combination, and hence, implicitly, a CC# value too—that already exists in the small file. (By contrast, of course, *p* will definitely involve a new part number, and possibly a new name and/or weight as well.) In general, in other words, insert operations will typically "touch" the large file only. Analogous remarks apply to delete operations also.

Endnotes

1. Note that the normalization process is basically a process of taking projections; in other words, the decomposition operator is projection (the *re*composition operator, by contrast, is join).
2. Recall from Chapter 10 that the cardinality of a set is the number of elements in that set. Thus, when we say some field is of such and such cardinality, what we mean is that the set of values in that field is of that cardinality; in other words, we're referring to the number of distinct values in that field, not the number counting duplicates if any.
3. In the interests of accuracy, I should point out that conventional normalization isn't entirely based on functional dependencies either. FDs take us only as far as *Boyce/Codd normal form* (BCNF). *Fourth normal form* (4NF) relies on a generalization of functional dependencies called *multivalued dependencies* (MVDs). Likewise, *fifth normal form* (5NF) relies in turn on a generalization of MVDs called *join dependencies*. And I've recently been involved myself in the definition of a new *sixth normal form* (6NF), which relies on a generalization of JDs (see reference [42]).
4. Factoring on the basis of subfields is not the same as subfield encoding (see the previous chapter), though the concepts are related. By the way, you might have noticed that the first two examples in this section, though advertised as "further possibilities," were really, like the examples in the previous section, based on the idea of approximate FDs—and the same is at least arguably true of the subfield example I'm about to consider. Factoring techniques do exist that genuinely aren't based in any way on approximate FDs, but further details are beyond the scope of this book.
5. Note the implication that not all FDs, genuine or approximate, are useful for factoring. To be specific, the FD (or "FD" in quotes, possibly) LHS → RHS is useful for factoring only if the right-hand side involves at least two attributes.
6. An interesting idea for consideration is the following: Instead of starting with exactly the relations specified by the database designer, it might be nice to begin by denormalizing the database entirely, joining all of the relations together—conceptually, at any rate—into what's sometimes called a "universal relation," and then to go on and use file factoring repeatedly (hierarchically, in fact) to achieve a good disk representation. In this way, attributes that started out in different user-level relations might even find themselves mapped to fields in the same file (or subfile, rather) internally.
7. Conceptually, that is. In practice, the results will be kept in the small Field Values Table (after all, the files per se are just abstractions and don't physically exist).
8. In particular, the row ranges in column CC# in Fig. 12.5 provide analogous support for the COUNT operator.

13 File Banding

13.1 Introduction

We saw in Chapter 11 that the Field Values Table (or Tables, plural) will always be in memory at run time, at least to a first approximation. And we saw in Chapter 12 how to use file factoring to reduce the space requirements for the Record Reconstruction Table(s) as well; basically, what we do is decompose the original file—at least conceptually—so that we wind up with one "large" Record Reconstruction Table and several "small" ones. And the small Record Reconstruction Tables too will always be in memory at run time (again to a first approximation). So we're left with the large Record Reconstruction Table on disk. That large table can't be compressed any further, more or less by definition; in other words, if we regard its contents just as simple bit strings, then those bit strings are essentially random sequences of zeros and ones.[1] The techniques discussed in this chapter and the next are specifically aimed at getting the best possible performance out of that large table, despite the fact that it's necessarily disk-resident.

Before I go any further, I should make it clear that, although I call it "large," the table we're dealing with—in fact, the entire TR data representation, including the Field Values Table(s) and *all* of the corresponding Record Reconstruction Tables—is still likely to be far smaller than a conventional direct-image representation. Actual experiments have shown that a reduction of five to one is quite typical (if anything, that estimate is probably on the low side). And that's just for the raw data; when indexes and other auxiliary structures are taken into account, the direct-image space requirement can increase by another five-to-one ratio, possibly even more.[2] However, when I need to appeal to such matters later in this chapter, I'll stick, conservatively, to the five-to-one figure.

Anyway, to remind you from Chapter 11, the problem with the large table is that the zigzags in that table, even if their starting points are physically contiguous (as indeed they are, because the table is stored column-wise), quickly splay out to essentially random positions "all over the disk," with the consequence that we might have to do a separate seek for every point after the starting point every time we chase such a zigzag. File banding, or just *banding* for short, is a technique for addressing this problem. Here in outline is how it works:

- Starting with a given user-level relation and hence a corresponding file, we sort that file into order based on values of some **characteristic field** (or field combination; for simplicity, I'll assume throughout what follows that we're always dealing with a single characteristic field, barring explicit statements to the contrary).

- Next, we decompose that sorted file horizontally[3] into two or more subfiles of approximately equal size. Each subfile is smaller than the original file in the sense that it has fewer records (usually far fewer) than the original file did. *Note:* The official term for "horizontal subfiles" is *bands,* and I'll favor this latter term in subsequent sections. You can think of those bands or horizontal subfiles as *partitions,* if you like; note, however, that specifics of the partitioning in question are determined primarily by physical space requirements and only secondarily by values of the characteristic field. We'll see some implications of this state of affairs in the next section (in particular, we'll see that a given characteristic field value might appear in more than one band, something that couldn't happen if the partitioning were done purely on the basis of values of that field).

- We then represent each of those subfiles or bands by its own Field Values Table and its own Record Reconstruction Table. Because the bands are smaller than the original file, those Field Values and Record Reconstruction Tables too are smaller than their counterparts would have been for the original file. In fact, we choose the band size such that any given band can fit into memory in its entirety at run time, and we lay the bands out on the disk in such a way as to allow any given band to be streamed into memory as and when it's needed. (When I say the entire band fits into memory, what I'm mainly talking about is *the Record Reconstruction Table* for the given band, of course. If the Record Reconstruction Table for a given band can be entirely contained in memory, then all of the zigzags within that table will also be entirely contained in memory a fortiori, and—insofar as that particular table is concerned, at least—the splay problem thus won't arise.)

Note: Please understand that the foregoing account is deliberately somewhat simplified; I'll come back and explain later (in Section 13.4) how banding is *really* done. However, the foregoing explanation is accurate enough to serve as a basis for discussions prior to that section.

The structure of the chapter is as follows. Following this introductory section, I'll explain the basic idea of banding by means of a simple example in Section 13.2; then I'll elaborate on and generalize from that example in Section 13.3, and introduce the important idea of controlled redundancy. As already mentioned, in Section 13.4 I'll build on the ideas of previous sections to show how banding is really done. Finally, in Section 13.5, I'll discuss the concept of controlled redundancy in more detail.

13.2 A Simple Example

In the interests of "user-friendliness," I'll continue to work with the familiar parts example (or an extended version of that example, rather). Assume again—as in Chapter 12, Section 12.5—that we've factored the parts file into large and small files that look like this:

```
Large file        Small file

P#                CC#
PNAME             COLOR
WEIGHT            CITY
CC#
```

Sample values for these files are shown in Fig. 13.1 (an extended version of Fig. 12.2 from Chapter 12). *Note:* Of course, it's the large file we're interested in here, not the small one. In the figure, of course, that file is hardly very "large" (obviously, since it has just nine records); however, don't lose sight of the fact that if we're really supposed to be building on the example from Chapter 12, then the file is really supposed to have some *ten million* records. What's more, fields in that file—with the obvious exception of the introduced artificial identifier CC#—are supposed to be of high cardinality, meaning each such field has around ten million distinct values as well; in fact, the data in the large file isn't supposed to display any "statistical clumpiness" at all.

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Nut	12.0	*cc1*
2	P2	Bolt	17.0	*cc2*
3	P3	Screw	17.0	*cc3*
4	P4	Screw	14.0	*cc1*
5	P5	Cam	12.0	*cc4*
6	P6	Cog	19.0	*cc1*
7	P7	Nut	19.0	*cc1*
8	P8	Wheel	15.0	*cc5*
9	P9	Hinge	20.0	*cc3*

	1	2	3
	CC#	COLOR	CITY
1	*cc1*	Red	London
2	*cc2*	Green	Paris
3	*cc3*	Blue	Oslo
4	*cc4*	Blue	Paris
5	*cc5*	Black	Rome

Fig. 13.1: Parts factored into large and small files (sample values)

For the purpose of comparison with later figures (Figs. 13.5 and 13.6 in particular), Figs. 13.2 and 13.3 show the Field Values Table and a possible Record Reconstruction Table for the large file (only) of Fig. 13.1. *Note:* For simplicity, throughout the examples in this chapter, I'll omit the direct pointers into the corresponding Field Values Table that the Record Reconstruction Table might contain in practice (or not, as the case may be).

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Bolt [1:1]	12.0 [1:2]	cc1 [1:4]
2	P2	Cam [2:2]	14.0 [3:3]	cc2 [5:5]
3	P3	Cog [3:3]	15.0 [4:4]	cc3 [6:7]
4	P4	Hinge [4:4]	17.0 [5:6]	cc4 [8:8]
5	P5	Nut [5:6]	19.0 [7:8]	cc5 [9:9]
6	P6	Screw [7:8]	20.0 [9:9]	
7	P7	Wheel [9:9]		
8	P8			
9	P9			

Fig. 13.2: Field Values Table for the large file of Fig. 13.1

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	5	5	1	1
2	1	2	8	4
3	8	7	2	6
4	7	9	9	7
5	2	1	5	2
6	3	8	6	3
7	6	3	3	9
8	9	6	4	5
9	4	4	7	8

Fig. 13.3: Record Reconstruction Table for the large file of Fig. 13.1

Now, let's assume page size is one megabyte (a figure that, as we saw in Chapter 11, is quite realistic). For simplicity, let's assume also that band size is the same as page size (though a band might map into any number of consecutive disk pages, in general—it's just a matter of what page size we're working with and how much memory we have available). Without getting into tedious arithmetic details, then, for a file of 10 million records and hence 24-bit pointers in the Record Reconstruction Table, we might get (say) 80,000 rows from that table into each band, for an overall total of 125 bands. *Note:* Actually that figure of 80,000 rows per band is too low (and the figure of 125 bands is accordingly too high), for reasons I'll explain later in this section.

Moving now from reality back to our toy example, let's assume the band size we have to work with corresponds to a maximum of just *four* records from the large file ... That file is (let's assume) sorted on ascending part number—in other words, P# is the characteristic field in this example—and so (referring to Fig. 13.1) band one will correspond to records *1-4,* band two to records *5-8,* and band three to record *9* only. Figs. 13.4, 13.5, and 13.6 show, respectively, the banded version of the file, the Field Values Tables for those bands, and Record Reconstruction Tables for those bands.

		1	2	3	4
		P#	PNAME	WEIGHT	CC#
Band 1	1	P1	Nut	12.0	*cc1*
	2	P2	Bolt	17.0	*cc2*
	3	P3	Screw	17.0	*cc3*
	4	P4	Screw	14.0	*cc1*
Band 2	5	P5	Cam	12.0	*cc4*
	6	P6	Cog	19.0	*cc1*
	7	P7	Nut	19.0	*cc1*
	8	P8	Wheel	15.0	*cc5*
Band 3	9	P9	Hinge	20.0	*cc3*

Fig. 13.4: Large file decomposed into three bands

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Bolt [1:1]	12.0 [1:1]	*cc1* [1:2]
2	P2	Nut [2:2]	14.0 [2:2]	*cc2* [3:3]
3	P3	Screw [3:4]	17.0 [3:4]	*cc3* [4:4]
4	P4			
5	P5	Cam [5:5]	12.0 [5:5]	*cc1* [5:6]
6	P6	Cog [6:6]	15.0 [6:6]	*cc4* [7:7]
7	P7	Nut [7:7]	19.0 [7:8]	*cc5* [8:8]
8	P8	Wheel [8:8]		
9	P9	Hinge [9:9]	20.0 [9:9]	cc3 [9:9]

Fig. 13.5: Field Values Tables for the bands of Fig. 13.4

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	2	3	1	1
2	1	1	2	4
3	4	2	3	2
4	3	4	4	3
5	5	5	7	6
6	6	7	8	7
7	7	8	5	5
8	8	6	6	8
9	9	9	9	9

Fig. 13.6: Record Reconstruction Tables for the bands of Fig. 13.4

Several points arise from the foregoing simple example:

- First of all, I need to own up to a slight terminological inexactitude on my part. When I first mentioned banding in the introduction to this chapter (also in Chapter 11), I said we decomposed the file into horizontal subfiles called bands, and so "band" was a concept at the file level. As you might have noticed, however, I subsequently started talking about bands fitting into memory at run time, and so "band" became a concept at the TR level (or possibly at some lower and more physical level still). However, it's very convenient to be able to use the same term *band* at these different levels of abstraction within the overall system, and—trusting that the practice won't cause confusion—I intend to continue doing the same thing throughout the remainder of this chapter.

- Next, observe that the Record Reconstruction Table for any given band does indeed involve pointers that are local to the band in question. In the figures I've stressed this fact by using *1-4* as the sole legal pointer values for band one, *5-8* as the sole legal pointer values for band two, and *9* as the sole legal pointer value for band three. In practice, of course, pointer values need only be unique within the relevant band (since the whole object of the exercise is not to have pointers out of one band into another).

- Precisely because pointers need now be unique only within the relevant band instead of within the entire file, they need fewer bits than they did before (before we did the banding, I mean). To revert for a moment to the example of a file with ten million records: Without banding, pointers are 24 bits, as we know; with banding, however, if one band corresponds to 80,000 records as suggested above, then pointers need be only 17 bits instead of 24—another significant space saving (and, be it noted, one that applies to the large Record Reconstruction Table specifically, a most gratifying state of affairs). *Note:* These facts explain why the figure of 80,000 rows per band quoted earlier in this section was too low. A more reasonable figure would be 115,000 or so (making the total number of bands 85 or so instead of 125).

- Note that banding does suffer from the drawback that it potentially introduces a degree—a tiny degree—of redundancy into the stored data. Without banding (but with condensing), no field value ever appears more than once in the Field Values Table. With banding, however, the same field value might simultaneously appear in several distinct bands (though never more than once per band). The WEIGHT value 12.0 is a case in point in Fig. 13.5: It appears in both band one and band two. *Note:* Such redundancy could even apply to the characteristic field, if values of that field aren't unique (clearly this can't happen in the example, though, because {P#} is a key—in fact, the only key—for the parts relation). More important, however, note that this particular drawback (the possibility of a tiny amount of redundancy, that is) disappears anyway if the approach to be described in Section 13.4 is adopted.

- Another drawback, perhaps more serious than the previous one, is the following: Since we no longer have just one Record Reconstruction Table for the entire file, it follows a fortiori that we can't have a "preferred" Record Reconstruction Table for the entire file (as described in Chapter 7) that provides major-to-minor orderings over all of the fields. However, it's at least true that each of the local Record Reconstruction Tables can be a "preferred" one so far as the records that belong to the band in question are concerned. The Record Reconstruction Tables of Fig. 13.6 are preferred ones in this sense.

- Equality or range queries based on the characteristic field can now be handled very efficiently by going directly to the relevant band or bands. (I'm assuming here that the system will keep certain metadata in memory, saying, for example, that band one has information regarding parts with part numbers in the range P1-P4, band two has information regarding parts with part numbers in the range P5-P8, and so on.) Even if several bands are involved, each can be processed independently of the rest (possibly even in parallel). And, of course, once a given band has been streamed into memory, record reconstruction within that band is a purely in-memory operation. Among other things, therefore, banding can provide functionality analogous, somewhat, to that provided by a conventional clustering index on the characteristic field (see Chapter 2 if you need to refresh your memory regarding clustering indexes).

- Note finally that banding does **not** have to be done "by hand"; rather, it can be done automatically during the load process (like factoring in the previous chapter), using built-in heuristics and statistical data analyses that are also done at load time. In other words, the benefits of banding, like those of factoring, can be obtained automatically, without any need for human decisions (except as noted in Section 13.5 below).

A Small Digression

You might have noticed something interesting has happened to band three in the example. Band three corresponds to a single record; it therefore contains a Field Values Table of just a single row and a Record Reconstruction Table of just a single row. Observe now that:

- In the case of the Field Values Table—ignoring the row ranges, which are clearly pretty pointless here anyway—the single row is effectively *a direct-image representation* of the record in question: namely, the "large-file" record for part P9 (see Fig. 13.1).

- In the case of the Record Reconstruction Table, the single row contains a "zigzag" that's in fact a straight line—necessarily so, of course. But a zigzag that's a straight line isn't all that useful, because the pertinent record can easily be "reconstructed" without it. To be specific, the field values of that record are now linked by physical contiguity in the Field Values Table (or something that might be thought of as akin to physical contiguity, at any rate).

It should be clear from the foregoing discussion that if the band size were such that every band corresponded to a single record, then we would be getting rather close to a direct-image representation of the entire file. It's interesting to observe, therefore, that—in a sense—the conventional direct-image style of representation might be regarded as just a highly suboptimal special case of the much more general TR style of representation. I'll leave this observation as something for you to meditate on at your leisure.

13.3 Elaborating on the Example

Let's get back to the main thread of our discussion. We've seen that, with banding, equality and range queries based on the characteristic field—the P# field, in the example—can be handled very efficiently, because the implementation can go directly to the relevant band or bands, stream it or them into memory, and complete processing of the query as a pure in-memory operation. But what about queries based on some other field? For example, consider the following SQL query:

```
SELECT DISTINCT P.P#
FROM P
WHERE P.WEIGHT = 12.0 ;
```

As I pointed out in the previous section, the WEIGHT value 12.0 appears in both band one and band two. In the worst case, of course, the same WEIGHT value could appear in *every* band, precisely because the original file was sorted on P#, not WEIGHT. Now, it might at least be possible for the implementation to know, from the Field Values Table(s), just which bands a given value does in fact appear in; but if it doesn't (and possibly even if it does), a query like the one just shown will effectively require a scan of the entire file, and performance might thus be poor. As noted in Chapter 11, in other words, the symmetric performance property is lost.

One way to address this problem is to band the original file twice, once using P# as the characteristic field and once using WEIGHT. In this way, we can have one set of banded Record Reconstruction Tables corresponding to the P# sort order and another set corresponding to the WEIGHT sort order: a form of **controlled redundancy** (see Section 13.5 for further discussion). Figs. 13.7, 13.8, and 13.9 show, respectively, the banded version of the file, the Field Values Tables for those bands, and Record Reconstruction Tables for those bands, if we sort and band by part weight as suggested (more precisely, by part number within part weight). *Note:* I'm still assuming four records per band, of course.

		1	2	3	4
		P#	PNAME	WEIGHT	CC#
Band 1	1	P1	Nut	12.0	*cc1*
	2	P5	Cam	12.0	*cc4*
	3	P4	Screw	14.0	*cc1*
	4	P8	Wheel	15.0	*cc5*
Band 2	5	P2	Bolt	17.0	*cc3*
	6	P3	Screw	17.0	*cc3*
	7	P6	Cog	19.0	*cc1*
	8	P7	Nut	19.0	*cc1*
Band 3	9	P9	Hinge	20.0	*cc3*

Fig. 13.7: Banding parts by weight

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Cam [1:1]	12.0 [1:2]	cc1 [1:2]
2	P4	Nut [2:2]	14.0 [3:3]	cc4 [3:3]
3	P5	Screw [3:3]	15.0 [4:4]	cc5 [4:4]
4	P8	Wheel [4:4]		
5	P2	Bolt [5:5]	17.0 [5:6]	cc1 [5:6]
6	P3	Cog [6:6]	19.0 [7:8]	cc2 [7:7]
7	P6	Nut [7:7]		cc3 [8:8]
8	P7	Screw [8:8]		
9	P9	Hinge [9:9]	20.0 [9:9]	cc3 [9:9]

Fig. 13.8: Field Values Tables for the bands of Fig. 13.7

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	2	2	1	1
2	3	1	3	2
3	1	3	2	3
4	4	4	4	4
5	5	5	7	7
6	8	7	8	8
7	6	8	5	5
8	7	6	6	6
9	9	9	9	9

Fig. 13.9: Record Reconstruction Tables for the bands of Fig. 13.7

The query

```
SELECT DISTINCT P.P#
FROM P
WHERE P.WEIGHT = 12.0 ;
```

can now be implemented by going directly to band one (only) in the foregoing banding, and symmetry of performance is restored.

13.4 How it's *Really* Done

Now I need to clean up my act ... I said in Section 13.1 that we build a separate Field Values Table and Record Reconstruction Table for each band in the banded file. In fact, however, that statement isn't quite accurate. What we really do is this: First, we build a single Field Values Table for the entire file in the usual way; then, for each band, we build a band-local "Field Values Table" (or an analog of such a table, rather) that contains, not field values as such, but rather *pointers into* the overall Field Values Table for the whole file.

Let's see how this works out in our example. First, Fig. 13.10 (a copy of Fig. 13.2) shows the Field Values Table for the entire "large file" from Fig. 13.1:

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Bolt [1:1]	12.0 [1:2]	cc1 [1:4]
2	P2	Cam [2:2]	14.0 [3:3]	cc2 [5:5]
3	P3	Cog [3:3]	15.0 [4:4]	cc3 [6:7]
4	P4	Hinge [4:4]	17.0 [5:6]	cc4 [8:8]
5	P5	Nut [5:6]	19.0 [7:8]	cc5 [9:9]
6	P6	Screw [7:8]	20.0 [9:9]	
7	P7	Wheel [9:9]		
8	P8			
9	P9			

Fig. 13.10: Field Values Table for the large file of Fig. 13.1 (same as Fig. 13.2)

Let's assume once again that we want to sort and band on part number. Here then (repeated from Fig. 13.4) is the first band:

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Nut	12.0	*cc1*
2	P2	Bolt	17.0	*cc2*
3	P3	Screw	17.0	*cc3*
4	P4	Screw	14.0	*cc1*

Here's the Field Values Table for this band as given in Fig. 13.5:

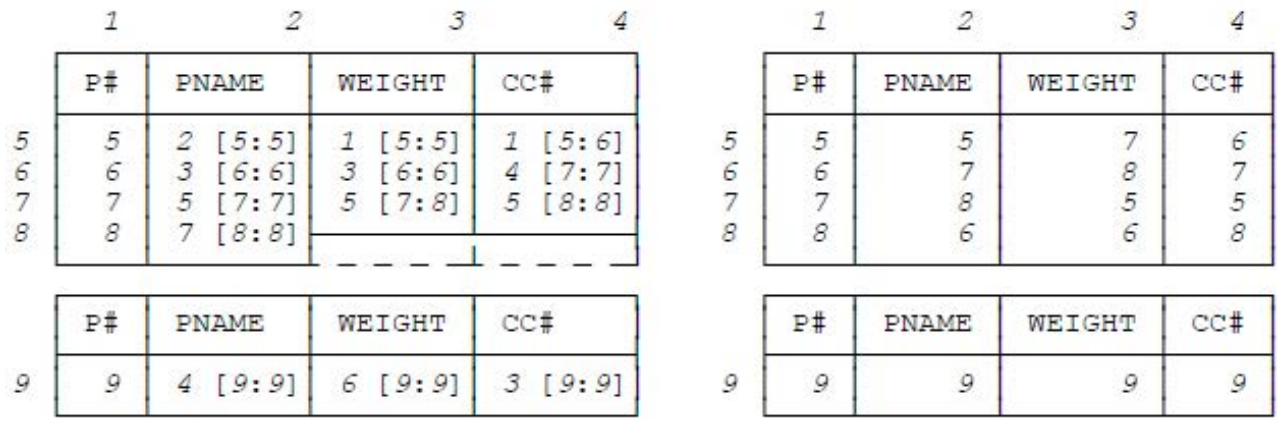
	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Bolt [1:1]	12.0 [1:1]	*cc1* [1:2]
2	P2	Nut [2:2]	14.0 [2:2]	*cc2* [3:3]
3	P3	Screw [3:4]	17.0 [3:4]	*cc3* [4:4]
4	P4			

And here's the corresponding analog of this Field Values Table with pointers into the main Field Values Table of Fig. 13.10 instead of actual field values (for convenience, I've extracted the corresponding Record Reconstruction Table from Fig. 13.6 and shown it on the right):

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	1	1 [1:1]	1 [1:1]	1 [1:2]
2	2	5 [2:2]	2 [2:2]	2 [3:3]
3	3	6 [3:4]	4 [3:4]	3 [4:4]
4	4			

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	2	3	1	1
2	1	1	2	4
3	4	2	3	2
4	3	4	4	3

Here for completeness are the Field Values Table analogs and Record Reconstruction Tables for the other two bands:



	1	2	3	4
	P#	PNAME	WEIGHT	CC#
5	5	2 [5:5]	1 [5:5]	1 [5:6]
6	6	3 [6:6]	3 [6:6]	4 [7:7]
7	7	5 [7:7]	5 [7:8]	5 [8:8]
8	8	7 [8:8]		

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
5	5	5	7	6
6	6	7	8	7
7	7	8	5	5
8	8	6	6	8

	P#	PNAME	WEIGHT	CC#
9	9	4 [9:9]	6 [9:9]	3 [9:9]

	P#	PNAME	WEIGHT	CC#
9	9	9	9	9

By way of example, let's consider the problem of reconstructing the record for part P6, say. The sequence of events is as follows:

- First we perform an in-memory look-up for part P6 in the Field Values Table of Fig. 13.10, and we discover that the record we want passes through cell [*6,1*] of that table.

- Knowing that part P6 falls into band two, we adjust that [*6,1*] to [*2,1*] to account for the fact that band one contains four parts. *Note:* Actually this step is unnecessary in our example, because I've numbered the rows *1-4* within band one, *5-8* within band two, and *9* within band three, and so we already know—albeit unrealistically—that [*6,1*] refers to a cell within band two. For definiteness and clarity, I'll continue to rely on that unrealistic assumption that row numbers are globally unique as shown in the figures.

- We stream band two into memory if it's not already there.

- Next, we follow the zigzag passing through cell [*6,1*] of the Record Reconstruction Table in band two (an in-memory process). That zigzag looks like this:

  ```
  [6,1], [6,2], [7,3], [5,4]
  ```

  We use in-memory look-ups to determine that the pointers (row numbers) in the corresponding cells of the corresponding Field Values Table analog are:

  ```
  6, 3, 5, 1
  ```

- We therefore go to cells

  [6,1], [3,2], [5,3], [1,4]

  of the Field Values Table of Fig. 13.10 (another in-memory process). The corresponding values are:

  P6, Cog, 19.0, *cc1*

  Reconstruction of the desired record is now complete.

At this point I'd like to remind you of something. In Chapter 5 (Section 5.6), I pointed out that row numbers can be regarded as *surrogates* for field values. In the record reconstruction example just now, for instance, the sequence of row numbers

6, 3, 5, 1

can be regarded as surrogates for the sequence of field values

P6, Cog, 19.0, *cc1*

Thus, we might reasonably think of the band-local Field Values Table analogs as containing, not actual field values as such (as indeed we now know), but surrogates for such field values instead.

13.5 Controlled Redundancy

In Section 13.3, I said that banding the parts file twice, once on part number and once on part weight, amounted to a form of controlled redundancy. Now, as I'm sure you know, redundancy in what's stored is usually considered to be a bad thing—not least because it can lead to inconsistencies. However, it's only when the redundancy is *uncontrolled* that it's unquestionably bad. *Controlled* redundancy—in other words, redundancy that's deliberately introduced and properly managed—is (or can be) fine; indeed, there are many sound reasons, both business and technical reasons, for storing several copies of the same data. But it does need to be understood that "controlled" here means that

a) The DBMS must be aware of the redundancy if it exists,

and more particularly that

b) The DBMS must take responsibility for "propagating updates" and maintaining data consistency (in other words, the redundancy must effectively be hidden from the user). I'll come back to this point in a few moments.

Let's return for a moment to the example from Section 13.3. Banding the parts file twice as suggested in that section clearly means we're going to need twice as much storage space. But I remind you that the data is already highly compressed; typically, as I pointed out in the introduction to this chapter, the TR representation requires only some 20 percent of the space required for a direct-image representation of the same data. So we can afford to band and store the original file *five different ways* and still not require any more storage than a conventional system does—and that's before the storage for indexes and other auxiliary structures is taken into account, in the direct-image case. *Note:* The point is worth making that indexes and the like effectively constitute a form of controlled redundancy in conventional systems anyway. And I've already mentioned the amount of storage space that kind of redundancy can involve (as noted in Section 13.1, a further fivefold increase is not at all atypical).

The kind of redundancy we're talking about in TR, then, is (to repeat) only redundancy on top of something that's already highly compressed. What's more, it's only redundancy on top of that portion of the data that can't be handled by the factoring techniques of Chapter 12. And what's more again, *it's the right kind of redundancy*. It's not the field values that are stored redundantly; rather, it's the linkage information. (No field value is ever stored more than once on the disk—assuming, of course, that column condensing and merging is done, as it certainly will be in a disk implementation. Contrast the situation in a direct-image system, with its indexes and other auxiliary structures, where it's virtually guaranteed that the very same field values will be stored many, many times over.) Storing the linkage information in different ways in a disk-based system is precisely what lets us achieve symmetry of performance in such a system.

Note, moreover, that it's a comparatively straightforward matter to decide what redundancies to store (in other words, to decide what sortings and bandings should be done). Detailed knowledge of the internal workings of the system is *not* required; nor is detailed knowledge of exactly the kinds of queries that users will submit. All that's needed is a general sense as to which fields are the pragmatically important ones—and this knowledge could even be obtained by the system itself, by analyzing actual or typical query sequences. Of course, if the system doesn't determine for itself what sortings and bandings are desirable, then the database administrator will have to tell it; in other words, human decisions will be required. But (to repeat) I don't think the decisions in question are very difficult ones.

Now, the obvious drawback to storing data redundantly is the impact it's likely to have on updates: If *N* distinct copies are stored of some given data item *X*, then an update to any one of those copies must be propagated to all the rest. But this is a much more tractable problem in TR than it usually is for at least two reasons:

- First, I've already said that field values as such *aren't* stored redundantly (so that "data item *X*" in my example just now can't be a field value in TR). Note in particular that update propagation can't affect index entries in TR, because there aren't any index entries in TR. Thus, update propagation is primarily a question of maintaining the pointers that are used in record reconstruction.

- Second, updates in the real world typically affect only a tiny portion of the overall database, as explained in Chapter 6. Typically, TR exploits this fact by keeping most of the database static for most of the time and segregating all updates in a much smaller overflow structure of their own (see Chapter 6, Section 6.5). That overflow structure is thus the only portion of the database that needs to be maintained in real time, and hence the only place where anything like update propagation has to be done in real time.

One last point in connection with redundancy in TR: Even if the database as stored does involve redundancy in the form of different bandings, there's no need to copy all of those different bandings to backup storage every time a full database backup is to be taken. All that's necessary is to copy just one of the bandings (the others can be recreated from that one).

Endnotes

1. To a first approximation yet again. In fact, as we saw in Chapter 11 (Section 11.5), there are compression techniques that do still work, even on that large table. For simplicity, however, I won't attempt to incorporate any of those techniques into my examples in this chapter.
2. This is not an overstatement. For example, in a report on the performance of a certain well-known SQL product on the standard TPC-H benchmark, reference [67] shows a raw data set of three terabytes expanding out to occupy nearly 60 terabytes of disk space, a twentyfold increase.
3. Contrast factoring, where we decompose files vertically (see the previous chapter). Indeed, just as there are certain parallels between factoring and conventional projection/join normalization, so there are certain parallels between banding and what might be called *restriction/union* normalization. Restriction/union normalization is a logical design technique, not much researched at the time of writing and certainly not yet much used in practice, in which the decomposition operator is restriction and the recomposition operator is union [32].

14 Stars and Zigzags

14.1 Introduction

This is the last chapter in this part of the book. In it, I want to describe a rather different approach to the problem of implementing the TR model on disk: more specifically, to the problem of minimizing disk seeks. Note immediately, therefore, that the approach in question can be regarded in part as an alternative to file banding as discussed in Chapter 13—but only in part, because in fact file banding can be used in combination with the approach to be described, as we'll see in Section 14.4. Note too that, as with the discussion of file banding in Chapter 13, we're primarily concerned here with how to deal with the "large file" that remains after file factoring has been used to get all of the "small files" into memory. But first things first.

As we know, the basic problem with TR on the disk is that if we're not careful, the zigzags can splay out all over the disk. Well, if the splay problem is caused by the zigzags, then let's get rid of the zigzags! Recall from Chapter 5 (Section 5.8) that the linkage information that lets us reconstruct records doesn't have to be implemented as zigzags specifically—other possibilities exist, with (of course) different performance characteristics. The approach to be described in this chapter exploits this idea; essentially, what it does is replace the zigzags by a different kind of structure called a **star**.

Let me illustrate this idea right away. Fig. 14.1 shows the Field Values Table and corresponding Record Reconstruction Table from Figs. 13.2 and 13.3 in Chapter 13—except that, for pedagogic reasons, I've shown the Field Values Table in uncondensed form. Fig. 14.2 then highlights one particular zigzag from Fig. 14.1 (actually the one for part P7), and Fig. 14.3 shows what happens if we replace that zigzag by a star.

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Bolt	12.0	cc1
2	P2	Cam	12.0	cc1
3	P3	Cog	14.0	cc1
4	P4	Hinge	15.0	cc1
5	P5	Nut	17.0	cc2
6	P6	Nut	17.0	cc3
7	P7	Screw	19.0	cc3
8	P8	Screw	19.0	cc4
9	P9	Wheel	20.0	cc5

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	5	5	1	1
2	1	2	8	4
3	8	7	2	6
4	7	9	9	7
5	2	1	5	2
6	3	8	6	3
7	6	3	3	9
8	9	6	4	5
9	4	4	7	8

Fig. 14.1: Uncondensed Field Values Table and corresponding Record Reconstruction Table

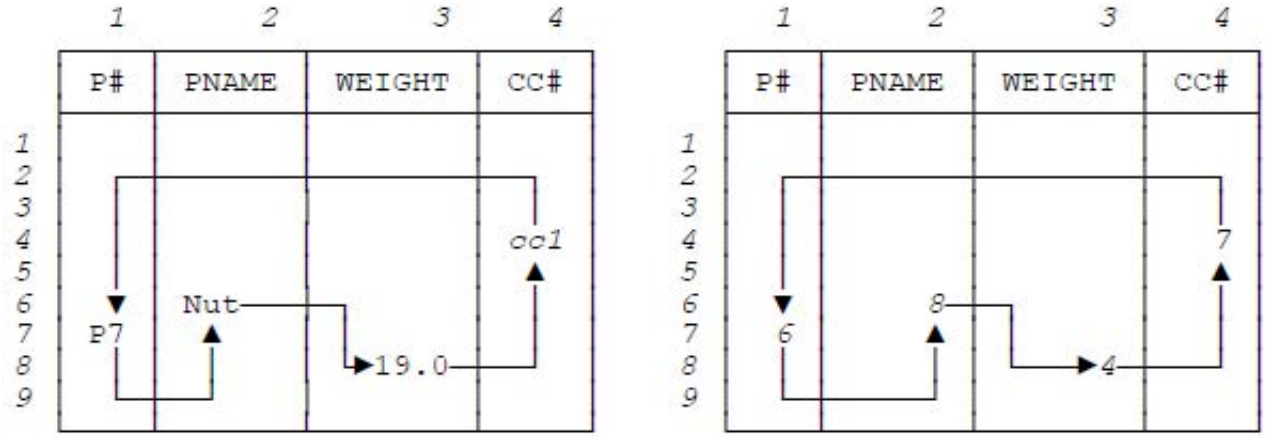


Fig. 14.2: Zigzag for part P7

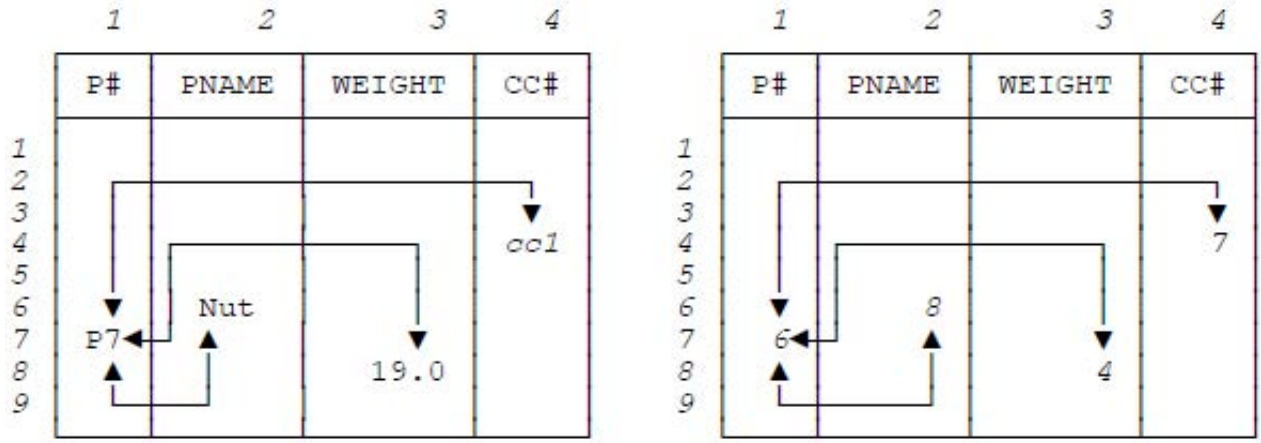


Fig. 14.3: Star for part P7 (with P# the core)

As you can see, where Fig. 14.2 has a *ring* of pointers (implemented within the Record Reconstruction Table and conceptually superimposed on the Field Values Table), Fig. 14.3 has a *star* of pointers instead. Cell [*7,1*], which corresponds to the P# value P7, serves as the center or **core** of that star. Three pointers emanate from that core and point to cells [*6,2*], [*8,3*], and [*4,4*], respectively; those cells correspond to the PNAME value Nut, the WEIGHT value 19.0, and the CC# value *cc1,* respectively. Those three pointers, which (as Fig. 14.3 indicates) are all two-way and can therefore be traversed in either direction, serve as the **spokes** or **rays** of the star.

Now, the star in the figure clearly does support reconstruction of the record for the part in question (part P7). To be specific:

a) If we start at the core, we can simply follow the three spoke pointers outward to obtain the other three field values.

b) If we start at any other point, we can follow the corresponding spoke pointer inward to the core and then proceed as under a) above—with the exception that, if we get to the core by following spoke pointer *sp* inward, then of course there's no need to follow that particular spoke *sp* outward again. *Note:* As a matter of fact, we *never* need to follow a spoke outward from the core within the Record Reconstruction Table as such; we only need to be able to go from the core outward to cells within the Field Values Table.

Now, you might have already realized that, for any given zigzag, there are several distinct but equivalent stars—it just depends on which field we choose as the core. I'll return to this point in Section 14.3. You might also have realized that the record reconstruction algorithm as just outlined displays asymmetric performance—access via the core field will be faster than access via any other field, because stars (unlike zigzags) are an inherently asymmetric structure—and I'll return to *this* point in Section 14.5.

The structure of the chapter is as follows. Following this introductory section, Section 14.2 gives a simple example to illustrate the basic ideas behind star structures. Section 14.3 elaborates on and generalizes that example. Section 14.4 shows how the ideas from the first three sections work on the disk (those previous sections are principally concerned with a memory-based implementation only). Finally, Section 14.5 discusses the use of controlled redundancy in connection with star structures.

14.2 A Simple Example

As in the previous chapter, the basic problem we're trying to deal with is how to get the best possible performance out of the "large" Record Reconstruction Table in a disk-based system. So I'll base my discussions on the same running example as in that previous chapter; to be specific, I'll assume once again that we've factored the parts file into large and small files that look like this:

```
Large file        Small file
P#                CC#
PNAME             COLOR
WEIGHT            CITY
CC#
```

However, we're interested here in the large file exclusively. Fig. 14.4 shows a sample value for that file (extracted from Fig. 13.1 in Chapter 13). And we've already seen a Field Values Table and a zigzag-based Record Reconstruction Table for that file in Fig. 14.1 above. *Note:* While the file shown in Fig. 14.4 is obviously not very large, let me remind you that we're really supposed to be dealing with files of millions or even billions of records, and the data in those files isn't supposed to display any "statistical clumpiness" at all.

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Nut	12.0	cc1
2	P2	Bolt	17.0	cc2
3	P3	Screw	17.0	cc3
4	P4	Screw	14.0	cc1
5	P5	Cam	12.0	cc4
6	P6	Cog	19.0	cc1
7	P7	Nut	19.0	cc1
8	P8	Wheel	15.0	cc5
9	P9	Hinge	20.0	cc3

Fig. 14.4: Sample file

Now, despite the fact that we're really supposed to be talking about a disk implementation, it's convenient to pretend for the time being that everything's in memory, and I'll adopt that pretense until further notice. So how do we proceed? Well, since (as we've already seen) stars are asymmetric, the first thing we have to do is decide what the core's going to be; in other words, we first have to choose a **core field** (much as we had to choose a characteristic field in connection with with banding in the previous chapter).[1] Suppose we choose field P#. Then Fig. 14.5 shows a corresponding star-based Record Reconstruction Table for the file of Fig. 14.4. *Note:* From this point forward, for convenience, I'll abbreviate the term "star-based Record Reconstruction Table" to just **star table,** and similarly for **zigzag table**.

	1	2	3	4
	P# n-w-c	PNAME	WEIGHT	CC#
1	5-1-1	2	1	1
2	1-5-5	5	5	4
3	8-6-6	6	4	6
4	7-3-2	9	8	7
5	2-2-8	1	2	2
6	3-7-3	7	3	3
7	6-8-4	4	6	9
8	9-4-9	3	7	5
9	4-9-7	8	9	8

Fig. 14.5: Star-based Record Reconstruction Table for the file of Fig. 14.4 (with P# the core)

In order to explain the star table of Fig. 14.5, let's go back for a moment to the *zigzag* table of Fig. 14.1. Consider the zigzag for (say) part P7, which—as Fig. 14.2 shows graphically—looks like this:

```
[7,1], [6,2], [8,3], [4,4]
```

In a star analog of this zigzag, therefore, the core cell [*7,1*] will contain the pointer triple *6-8-4* (these are the outward portions of the spokes) and cells [*6,2*], [*8,3*], and [*4,4*] will each contain a *7* (these are the inward portions of the spokes). And similarly, of course, for all of the other stars in the table. *Note:* I've expanded the heading of column P# in Fig. 14.5 to show which pointers are which. To be specific, in the triple *n-w-c, n* is the PNAME (name) pointer, *w* is the WEIGHT pointer, and *c* is the CC# pointer.

Observe, incidentally, that it's a consequence of the way the star table in the example is defined that:

a) The first "subcolumn" within column P# of the star table—the one for PNAME, labeled *n* in the figure—is identical to column P# of the zigzag table (why, exactly?);

b) Column CC# of the star table (the last column) is identical to column CC# of the zigzag table (again, why exactly?).

Now let's consider how the star table of Fig. 14.5 might be used to implement queries. Consider the following simple example:

```
SELECT DISTINCT P.P#, P.WEIGHT
FROM P
ORDER BY P# ;
```

This query refers to relation P, but of course it can be implemented by accessing the large file only—we[2] don't need to touch the small file at all. (This is a generic observation, and I won't bother to repeat it in subsequent examples.) So what we have to do is this:

- Traverse column P# of the star table top to bottom to obtain the result in the desired ordering.

- The first cell encountered, cell [*1,1*], corresponds to cell [*1,1*] in the Field Values Table, which (as Fig. 14.1 shows) contains the part number P1.

- That same first cell in column P# of the star table contains the pointer triple *5-1-1*. The first pointer in this triple corresponds to a part name, the second to a weight, and the third to a CC# value. However, the query isn't interested in part names or CC# values, so we can go just to the WEIGHT cell in the Field Values Table—which is to say cell [*1,3*]—to obtain the desired weight value. The first result record has now been constructed.

- All other result records are constructed analogously.

Note: If the query had specified ORDER BY WEIGHT instead of ORDER BY P#, we would have accessed the star table by column WEIGHT instead of column P#. For each cell encountered, we would have followed the inward pointer to the corresponding core cell and then used that core cell to construct the corresponding result record as above.

One point that emerges right away from the foregoing is that stars might be better than zigzags for implementing projections. To be specific, if the file has *M* fields, then (in general) zigzags require *M* accesses to the Record Reconstruction Table for each result record no matter how many fields are requested, while stars require at most two (and often only one). This fact makes stars attractive, because it's quite rare in practice for a query to request *all* of the fields of the file (or all of the attributes of the relation, rather).

Here's another sample query:

```
SELECT DISTINCT P.P#
FROM P
WHERE P.WEIGHT = 19.0 ;
```

Here we do a binary search on column WEIGHT of the Field Values Table and determine that the records we want pass through cells [*7,3*] and [*8,3*] of the star table. Then we use the stars corresponding to those cells to construct the desired records.

Before closing this section, I should draw your attention to one more point: namely, that a star table will always be bigger than its zigzag analog. Again suppose the file has *M* fields. Then the zigzag table will have *M* pointers per record, but the star table will have 2(*M*-1)—so if *M* is large, the star table will be almost twice the size of the zigzag table. *Note:* I'm assuming here that *M* is greater than one. What happens if that assumption is invalid?

14.3 Elaborating on the Example

Now let's see what happens if we choose a field other than one corresponding to some key as the core field. Let's choose field WEIGHT. Fig. 14.6 shows what happens to the star for part P7 under this assumption; Fig. 14.7 shows the corresponding star table in its entirety. Observe that:

a) The first "subcolumn" within column WEIGHT of the star table of Fig. 14.7—the subcolumn for CC#, labeled *c* in the figure—is identical to column WEIGHT of the zigzag table of Fig. 14.1;

b) Column PNAME of the star table of Fig. 14.7 is identical to column PNAME of the zigzag table of Fig. 14.1.

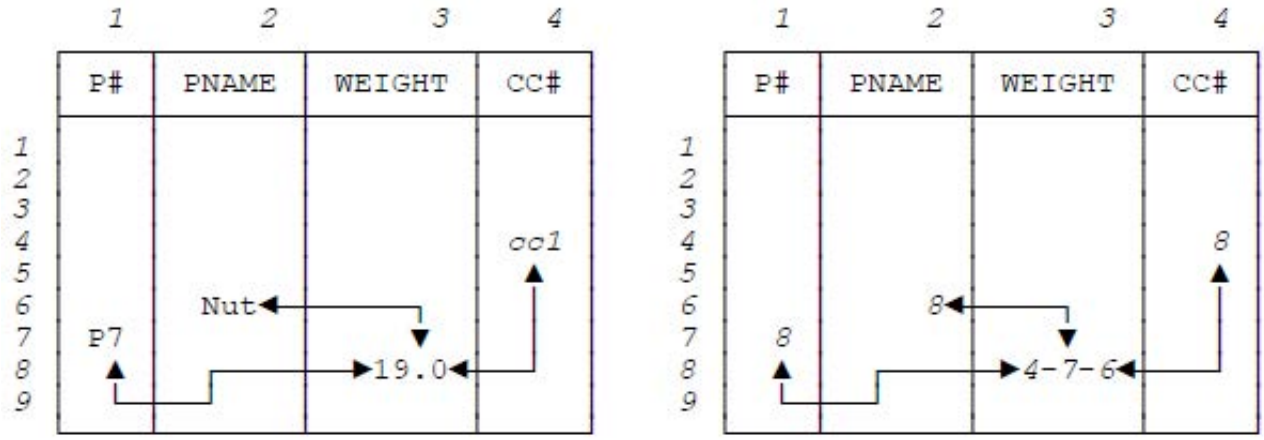


Fig. 14.6: Star for part P7 (with WEIGHT the core)

	1	2	3	4
	P#	PNAME	WEIGHT c-p-n	CC#
1	1	5	1-1-5	1
2	5	2	8-5-2	3
3	6	7	2-4-7	7
4	3	9	9-8-9	8
5	2	1	5-2-1	5
6	7	8	6-3-8	6
7	8	3	3-6-3	9
8	4	6	4-7-6	2
9	9	4	7-9-4	4

Fig. 14.7: Star table for the file of Fig. 14.4 (with WEIGHT the core)

Observe too that (to spell out the obvious) the star tables of Figs. 14.5 and 14.7 are different. Thus, while we might reasonably talk about "the" zigzag table for a given file, we can't sensibly talk about "the" star table for that same file; instead, we have to talk about the star table that corresponds to the given file *together with some given core field*.

Now I want to make another point. Suppose we use the star table of Fig. 14.7 to reconstruct the entire file; suppose for definiteness that we perform this process using column PNAME (that is, we traverse column PNAME of the star table top to bottom). Here's the reconstruction process spelled out in detail:

- From cell [*1,2*] of the star table, follow the spoke inward to the corresponding core cell [*5,3*].

- Go to the Field Values Table and extract the WEIGHT value in cell [*5,3*] (from Fig. 14.1, we see the value in question is 17.0).

- Cell [*5,3*] of the star table contains the pointers *5, 2,* and *1*. Go to the Field Values Table and extract the CC# value in cell [*5,4*], the P# value in cell [*2,1*], and the PNAME value in cell [*1,2*]. Those values are *cc2,* P2, and Bolt, respectively, and we have now constructed the first result record. *Note:* Alternatively, of course, we could have obtained the PNAME value (Bolt) in the first step by going straight to cell [*1,2*] of the Field Values Table (since we started out with cell [*1,2*] of the star table in the first place).

Performing this sequence of steps eight more times but starting successive iterations with cells [*2,2*], [*3,2*], ..., [*9,2*] of the star table (for the second, third, ..., ninth record in the overall file reconstruction process), we obtain the result shown in Fig. 14.8. That result, as you can see by inspection (or by comparison with Fig. 13.1 in Chapter 13), consists of the original nine part records ordered by weight within name (more precisely, ordered by P# within CC# within WEIGHT within PNAME). In other words, the star table of Fig. 14.7 is a "preferred" one in the sense of Chapter 7. (So too is the star table of Fig. 14.5, as a matter of fact.)

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P2	Bolt	17.0	*cc2*
2	P5	Cam	12.0	*cc4*
3	P6	Cog	19.0	*cc1*
4	P9	Hinge	20.0	*cc3*
5	P1	Nut	12.0	*cc1*
6	P7	Nut	19.0	*cc1*
7	P4	Screw	14.0	*cc1*
8	P3	Screw	17.0	*cc3*
9	P8	Wheel	15.0	*cc5*

Fig. 14.8: Part records ordered by WEIGHT within PNAME

One last point to close this section: Consider, by way of example, cell [*8,3*] of the star table in Fig. 14.7. That cell corresponds directly to cell [*8,3*] of the Field Values Table in Fig. 14.1, which contains the weight 19.0. That same cell [*8,3*] in the star table contains the pointers *4, 7,* and *6,* which take us to cells [*4,4*], [*7,1*], and [*6,2*] of the Field Values Table, and those cells in turn contain the CC# value *cc1,* the part number P7, and the name Nut, respectively. In a sense, therefore, that star table cell [*8,3*] can be thought of, all by itself, as a digitized version of the entire record for part P7[3]—its position within the star table effectively specifies one component of that record (the WEIGHT component), and its contents effectively specify the other three components (the CC#, P#, and PNAME components). *Note:* It's relevant to mention once again that—as we first saw in Chapter 5, Section 5.6—pointers to Field Values Table cells can usefully be thought of as *surrogates* for the field values contained within those cells.

14.4 What Happens on Disk

Now let's drop the pretense that everything's in memory and see how the ideas we've been discussing work out in a disk environment—by which I mean, primarily, an environment in which the star table is too big to be memory-resident. *Note*: It's easier to talk in terms of just one star table at a time; in what follows, therefore, I'll pretend there is indeed just one such table (barring explicit statements to the contrary), and I'll refer to it as "the" star table.

The first point is that, in the case of the core column in particular, we're probably going to want to extend the column-wise storage idea to store each subcolumn of that column as an independent column in its own right. The reason is that (as we saw in Section 14.2) we usually don't need to access all of those subcolumns in implementing any given query, and we certainly don't want to read anything into memory that we don't need, if we can help it. Fig. 14.9 shows the star table of Fig. 14.5—the one based on P# as the core field—with the core column divided up into separate columns in this way. *Note:* The term subcolumn, which I've now used several times, is (I hope) self-explanatory. From this point forward, I'll use the term core column to mean the combination of all pertinent subcolumns.

	1n	1w	1c	2	3	4
	core (P#)			PNAME	WEIGHT	CC#
	PNAME	WEIGHT	CC#			
1	5	1	1	2	1	1
2	1	5	5	5	5	4
3	8	6	6	6	4	6
4	7	3	2	9	8	7
5	2	2	8	1	2	2
6	3	7	3	7	3	3
7	6	8	4	4	6	9
8	9	4	9	3	7	5
9	4	9	7	8	9	8

Fig. 14.9: Star table of Fig. 14.5 with core column subdivided

The table of Fig. 14.9 will be stored on disk as six separate columns, and the implementation will thus be able to go directly to the start of any of those stored columns at any time. What's more, those six columns will also be stored in consecutive pages on the disk (the first column in one set of pages, the second in the immediately following set, and so on), with a view to minimizing seek activity and allowing successive columns to be streamed into memory at run time. (Of course, it's highly desirable for the pertinent core subcolumns to be in memory at run time—where by "pertinent" I mean the ones that are needed for any given query—even if the star table overall is too big to fit into memory in its entirety.)

So what's good about this arrangement from a performance point of view? Well, it certainly means that pointers in any given column of the star table will be physically contiguous on the disk, with the implication that traversing such a column will be fast. And since these remarks apply to subcolumns of the core column in particular, it follows that doing record or file reconstruction via the core column will *not* lead to the splay problem. What's more, so long as the core column is in memory, doing reconstruction via any other column will be efficient too; but if the core column is too big to fit into memory, then reconstruction via any other column still has the potential to be extremely inefficient. However, at least we don't have to deal with "splayed zigzags" as such.

So what can we do if the core column is too big to fit into memory? One approach would be to use **banding,** more or less as described in the previous chapter. Banding, as you'll recall, is a divide-and-conquer technique that works by dividing the file up into horizontal subfiles or bands such that (a) each band fits into memory and (b) no pointing occurs between bands. Refer to Chapter 13 for further discussion of this possibility. The section immediately following describes a different approach to the same problem.

14.5 Controlled Redundancy

We've seen that, in order to define a star table, the first thing we have to do is choose a core field.[4] Now, when we were discussing banding in Chapter 13, the choice of characteristic field had major performance implications, because that choice effectively dictated the stored sort order for the file. And the situation is similar (though not identical) with a star table and its core field; again our choice can have major performance implications. To be more specific, if we choose *C* as the core field, then queries that involve access to the file in sequence by values of *C* will be very efficient, but (as explained near the end of the previous section) queries that involve access in any other sequence might not be.

The obvious solution to this problem is to introduce some form of **controlled redundancy** once again. I discussed controlled redundancy at some length in Chapter 13 (Section 13.5); most of the points I made there apply here too, and I won't bother to repeat them all now. Let me just remind you yet again that the TR representation of a given data set typically requires only some 20 percent of the space required for a direct-image representation; as a result, we can afford to store the data up to five different ways without taking up any more space than a conventional system would need (and that's just for storing the raw data alone, in that conventional system).

So let's consider the question of redundancy in the context of star tables specifically. Clearly, the simplest thing to do is to store several different star tables for the same file, each one based on a different core field. For example, we could store both the star table of Fig. 14.5 (based on P#) and the star table of Fig. 14.7 (based on WEIGHT), and thereby achieve symmetric performance for access to parts based on P# and access to parts based on WEIGHT.

An alternative approach would be to store just one star table but to expand that table to include, in addition to what I'll now call the *primary* core column, a set of *secondary* core columns as well. Each such secondary core column will contain essentially the same information as the primary one, but sorted into a sequence that matches the sort sequence of some field that's not the (primary) core field.

To see how this idea works out in practice, let's work through an example. Consider column WEIGHT in the star table of Fig. 14.9. As you can see, that column contains the following pointers (row numbers) in top-to-bottom sequence:

```
1, 5, 4, 8, 2, 3, 6, 7, 9
```

This sequence is in fact the *permutation* that corresponds to the specification ORDER BY WEIGHT, CC#, P#, PNAME; it means, for example, that record *8* of the file—see Fig. 14.4—appears in position *4* in that ordering. It follows that if we were to process the core column of that same star table by taking the first cell first, the fifth cell second, the fourth cell third, and so on, we would reconstruct a version of the file that was in exactly that ordering.

The trouble is, of course, that processing the core column in the way just indicated could lead to a lot of seek activity on the disk. So why not store another copy of that core column that's rearranged into exactly the sequence we want? The result might look like this:

```
5-1-1
2-2-8
7-3-2
9-4-9
1-5-5
8-6-6
3-7-3
6-8-4
4-9-7
```

If such a column is added—redundantly, of course—to the star table of Fig. 14.5, we'll be able to reconstruct the desired file from that table in the desired order without all of that seek activity on the disk, precisely because that new column will be stored as a separate column in its own right on the disk. (Well, actually it'll be stored as three separate columns, one for each subcolumn, but that detail need not concern us here.)

But wait a moment ... Recall that within a given triple of pointers *n-w-c* in the primary core column, *n* is the name pointer, *w* is the weight pointer, and *c* is the CC# pointer. Clearly there's no need to include the *w* pointers in the secondary core column, because the value of the *w* pointer in the *i*th cell will always simply be *i* (as you might have already noticed). Thus, the final version of the star table with both a primary core column and a secondary core column will look as shown in Fig. 14.10 (note the column labels *1n, 1w,* etc.).

	1n	1w	1c	2	3	3n	3c	4
	primary			PNAME	WEIGHT	secondary		CC#
	PNAME	WEIGHT	CC#			PNAME	CC#	
1	5	1	1	2	1	5	1	1
2	1	5	5	5	5	2	8	4
3	8	6	6	6	4	7	2	6
4	7	3	2	9	8	9	9	7
5	2	2	8	1	2	1	5	2
6	3	7	3	7	3	8	6	3
7	6	8	4	4	6	3	3	9
8	9	4	9	3	7	6	4	5
9	4	9	7	8	9	4	7	8

Fig. 14.10: Star table with primary (P#) and secondary (WEIGHT) cores

Now suppose we want to reconstruct the part record passing through some particular cell in the WEIGHT column of the star table of Fig. 14.10—let's say (arbitrarily) the fourth cell, which is still cell [*4,3*] according to the column labeling shown in the figure. The sequence of events is as follows.

- Go to cell [*4,3*] of the Field Values Table of Fig. 14.1 and extract the value stored there (weight 15.0).

- Cell [*4,3*] of the star table contains the row number *8,* so the part number of the record we want is in cell [*8,1*] of the Field Values Table. Go and extract it (part number P8).

- Within the secondary core column, go to the PNAME cell corresponding to WEIGHT cell [*4,3*]. That PNAME cell is cell [*4,3n*], and it contains the row number *9*. Go to cell [*9,2*] of the Field Values Table and extract the name (Wheel).

- Within the secondary core column, go to the CC# cell corresponding to WEIGHT cell [*4,3*]. That CC# cell is cell [*4,3c*], and it also contains the row number *9*. Go to cell [*9,4*] of the Field Values Table and extract the CC# value (*cc5*). Record reconstruction is now complete.

Here's the storage arithmetic again. Suppose once again that the file has *M* fields. Then:

a) A zigzag table will have *M* pointers per record.

b) A star table with a single core column will have 2(*M*-1) pointers per record.

c) A star table with a primary core column and *N* secondary core columns will have 2(*M*-1) + *N*(*M*-2) pointers per record.

Of course, Case b. is just that special case of Case c. in which *N* is zero.

Endnotes

1. In fact the core field is often referred to as a characteristic field. I'll stay with the term *core field* in this chapter.
2. As in Chapter 10, "we" here really means the DBMS.
3. *Chambers Twentieth Century Dictionary* defines *digitize* to mean "to put (data) into digital form for use in a digital computer."
4. It might be possible to automate that choice, but probably not if we introduce redundancy (which I'm about to do); in that case, human decisions are probably going to be needed.

Part IV: Conclusion

15 The Future Looks Bright Ahead

15.1 Introduction

The goal of this book has been to present a tutorial on the TransRelational Model (the TR model, also referred to herein as TR technology, or just TR for short). As explained in the preface, the TR model represents one specific but important application of a more general technology called the Tarin Transform Method; that more general technology is suitable for building data management systems of many different kinds, but I've deliberately concentrated in this book on its suitability for implementing the relational model in particular. Now, in this final chapter, I want to summarize and analyze the main points from what's gone before—especially with respect to the benefits that this exciting new technology can provide—and I also want to speculate a little as to what might lie ahead.

15.2 The TR Model Summarized

Everything I've said about the TR model in this book so far has been based on Required Technologies documentation (the Initial Patent [63] in particular). However, I need to make it clear that I've altered most of the terms, I've simplified many of the concepts (and even omitted a few), and I've imposed my own sequence on the material—always in the hope of making what I think are the really important ideas more readily understandable. Also, the strict stratification into three layers of abstraction described in Chapter 3 (and adhered to throughout the present book) isn't explicitly called out in the Required Technologies documents, and the same is true for some of the techniques sketched in Chapter 10 and elsewhere for implementing the relational operators.

Data Independence

TR is a **transform** technology. The notion of a transform (as that term is used in the TR context) is a logical consequence of the familiar notion of *data independence:* It should be possible to change the way the data is physically stored without having to change the way the data looks to the user. This objective clearly implies the need for at least one transform—more generally, for a set of *N* transforms for some *N* greater than zero—to be performed between the external and internal levels of the system. The trouble is, today's direct-image systems provide only a very weak form of data independence (I'm tempted to say they implement the *identity* transform). As a consequence, we've come to think of data independence as little more than just shielding the user from the bits and bytes on the disk. But there's so much more to it than that! We need to get away from those direct-image transforms. And that's what TR is all about; TR is, I think, unique in the emphasis it places on the crucial concept of data independence. Every part of the TR model as described in earlier chapters fits naturally within, and contributes to, this overall perspective on the database implementation problem.

Let me illustrate the foregoing by briefly reviewing the material from those earlier chapters. I began in Chapter 3 by distinguishing **three levels of abstraction**—the user level, which is relational; the TR level, which is based on TR tables (principally the **Field Values Table** and the **Record Reconstruction Table**); and the file level, which is a level of indirection between the other two. (Thus, we're already talking about at least two transforms, one between relations and files, and one between files and TR tables.) Relations have tuples and attributes; files have records and fields; tables have rows and columns. I stressed the point that TR tables are definitely not the same thing as SQL tables, and I explained at some length why I thought it was better to use relational terminology, not SQL terminology, at the user level. Indeed, I think it's fair to say that one problem with SQL—one of many, unfortunately—is precisely that it muddies the distinction between the relational and file levels; in some ways, in fact, SQL tends to focus on the file level more than it does on the relational level.

Be that as it may, the crucial insight underlying the TR model is this. Let *r* be some record at the file level. Then:

> **The stored form of *r* involves two logically distinct pieces, a set of field values and a set of "linkage" information that ties those field values together, and there's a wide range of possibilities for physically storing each piece.**

In direct-image systems, the two pieces are kept together, and the linkage information is represented by physical contiguity. In TR, by contrast, the two pieces are kept separate; the field values are kept in the Field Values Table, and the linkage information is kept in the Record Reconstruction Table. That separation (which represents a major logical transform right away, of course) effectively allows the very same stored data to be kept sorted in many different ways at the same time. It's rather like having many different pointer chains running through the same set of stored data at the same time; however, the big difference is that, in TR, (a) those pointer chains are separate from the stored data as such, and (b) they effectively connect fields, not records (contrast pointer chains as found in CODASYL systems, as described in Chapter 2). Also, we saw in Chapter 14 that those "chains" of pointers aren't necessarily chains anyway, in TR.

Memory Implementation

In Chapters 4-10, I presented the basic ideas of the TR model while ignoring (for the most part) the special problems of implementing that model on disk, and I'll follow the same pattern in this brief review. First of all, then, let's go over the way the Field Values Table and Record Reconstruction Table might be implemented in memory (for full details, see Chapter 4).

In its simplest form, the Field Values Table has a column for each field in the corresponding file, and the entries in a given column consist of the field values from the corresponding column arranged into sorted order. Let *cfv* and *crr* denote cell [*i,j*] of the Field Values Table and cell [*i,j*] of the Record Reconstruction Table, respectively. Let *r* be that record of the file whose *j*th field value appears in *cfv,* and let the (*j*+1)st field value of *r* appear in cell [*i′,j*+1] of the Field Values Table. Then *crr* contains *i′*. Thus, the Record Reconstruction Table allows any or all of the records in the file to be reconstructed—by means of the **zigzag algorithm**—from the Field Values Table. Moreover, entering the Record Reconstruction Table on any particular column *j* and reconstructing the record corresponding to cell [1,*j*], then the record corresponding to cell [2,*j*], and so on, will eventually reconstruct a version of the file whose records are ordered by values of field *j*.

Let me remind you that the Field Values Table and the Record Reconstruction Table both start out being isomorphic to the corresponding file—that is, they both have the same number of rows and columns as that file has records and fields,

respectively. What's more, the Record Reconstruction Table stays isomorphic in this sense; however, the Field Values Table ceases to do so when the condensed- and merged-column transforms are introduced (see below). Let me also remind you that the Field Values Table is the only TR-level construct that contains user data as such; all the rest—the Record Reconstruction Table, also the Permutation Table and others—contain implementation information (mostly pointers). Finally, let me remind you that those pointers can usefully be thought of **surrogates** for the corresponding field values.

In Chapter 5, I briefly discussed what's involved in inserting new records and in retrieving, deleting, or updating the records that "pass through" some given cell of the Record Reconstruction Table. I explained how DELETE didn't physically remove information from the database but merely flagged it as "logically deleted," and how subsequent INSERTs could then reuse such logically deleted items. I pointed out that finding records and retrieving them (or deleting or updating them) were logically distinct processes, and I explained how TR took advantage of that fact. And I explained how all of these operations effectively took place at the field level rather than the record level. I also discussed "symmetric exploitation" and the possibility of corresponding symmetry of performance (but that's an issue I want to come back to in the next section).

In Chapter 6, I discussed update operations in more depth. In particular, I explained the swap algorithm for implementing INSERT operations; I also sketched an alternative approach based on the use of a separate overflow structure, and pointed out that such an approach enjoyed many advantages (not only in the area of performance but also, and importantly, in the area of backup and recovery). *Note:* Yet again we're talking about some important transforms. I won't keep on saying this.

Next, recall that the Record Reconstruction Table corresponding to a given file (and given Field Values Table) isn't unique, in general, owing to the fact that most fields in most files involve duplicate values. In Chapter 7, I showed how we can take advantage of this fact; to be specific, I showed how certain "preferred" Record Reconstruction Tables could be used to provide several **major-to-minor orderings** simultaneously (as well as, a fortiori, several individual field orderings simultaneously). And I also showed in that chapter (as well as in Chapters 5 and 7) how to use the Permutation and Inverse Permutation Tables as a basis for building any desired Record Reconstruction Table, "preferred" or otherwise. "Preferred" Record Reconstruction Tables constitute one of several important refinements to the basic TR model; although those refinements might be thought of as frills, in a sense, they're so important and useful that (it seems to me) they're virtually certain to be supported in any real implementation.

In Chapter 8, I took a look at another important refinement, **condensed columns**. The idea here is that columns in the Field Values Table can be "condensed" by removing redundant duplicate field values (keeping instead, with each individual field value that remains, a *row range* indicating which rows would have contained that value in the corresponding *un*condensed version of the Field Values Table). As well as representing a possibly dramatic saving in storage space (see the subsection entitled "The Copernican Analogy" in the next section), condensed columns make update operations (and retrieval operations too, quite probably) much more efficient. I remind you that a condensed column can usefully be thought of as a **histogram**.

Of course, condensing the Field Values Table in the foregoing sense does make file and record reconstruction a little more complicated, and possibly a little less efficient. We can fix this problem by expanding the Record Reconstruction Table, such that each cell now includes two pointers (that is, two row numbers) instead of one. One pointer is the same as before—it identifies the appropriate "next" cell in the Record Reconstruction Table—while the other is a direct pointer to the cell of the Field Values Table that contains the corresponding field value.

In Chapter 9, I discussed yet another important refinement, **merged** columns. The idea here is that distinct fields at the file level might map to the same column in the Field Values Table, eliminating further redundancy, and in particular making joins more efficient. What's more, the distinct fields in question don't have to come from the same file—the only requirement is that they must be of the same data type; thus, there's no longer necessarily a one-to-one correspondence between files at the file level and Field Values Tables at the TR level (note the implications here for candidate and foreign keys in particular). In the extreme case, in fact, there could be just a single Field Values Table for the entire database. In relational terms, such an implementation would effectively mean that we were storing *attributes* instead of *tuples* (and those stored attributes would never contain any duplicate values).

Note: In characterizing such an implementation in such a manner, I'm tacitly regarding (for example) attribute S# in the suppliers relation S and attribute S# in the shipments relation SPJ as "the same" attribute. This interpretation is consistent with the formal definition of the term *attribute* as found in, for example, reference [40]. What's more, since it's even possible for that single Field Values Table to include "logically deleted" values that don't currently appear in any user relation at all, such an implementation might reasonably be characterized as one—or as approaching one—that stores **domains** (= types), not just attributes.

Next, in Chapter 10, I indicated what was involved in using TR to implement the relational operators, and gave evidence to support the claim that those implementations should be especially efficient. Joins in particular involve **linear costs** instead of multiplicative ones; in my opinion, this fact by itself—even if it was the *only* advantage provided by TR—would still be more than sufficient to place TR head and shoulders above its competitors. Note in particular that it implies that joins are **scalable;**[1] as a consequence, if some query fundamentally requires *N* joins, then it's all right to go ahead and request those *N* joins, *regardless of the value of N*. By contrast, it's well known that direct-image implementations are effectively incapable of handling values of *N* that are greater than some fairly small lower bound (perhaps seven or eight). Yet a properly designed database could easily have several hundred relations, and realistic queries could easily involve a 20- or 30-way join.

Disk Implementation

In Chapters 11-14, I turned my attention to the question of implementing the TR model on disk. In Chapter 11, I explained the basic problem: *We need to do everything we can to minimize disk seeks*. More specifically, we want as much of the database as possible to be memory-resident at run time, and we want a good data representation on disk to reduce the amount of seeking we have to do when we do have to do it. The overall objective is to try and get "main-memory performance off the disk."

Chapter 11 also described some of the logical and physical **compression techniques** that TR uses to address the foregoing problems. Note in particular that (at least to a first approximation) those techniques have the effect of ensuring that the Field Values Table will always be memory-resident. But the Record Reconstruction Table has the potential to be much larger than the Field Values Table and therefore still presents a problem. Chapter 11 included an overview of certain TR-specific approaches to that problem, while the next three chapters described three of those TR-specific solutions in more detail. Chapter 12 explained the use of **file factoring** to reduce the problem to one of dealing effectively with "large files." Chapters 13 and 14 then addressed this latter problem in detail; Chapter 13 discussed the use of **file banding,** and Chapter 14 examined the possibility of using **stars** instead of zigzags in the Record Reconstruction Table. Chapters 13 and 14 also raised the possibility of judicious use of **controlled redundancy**.

Let me conclude this brief summary by reminding you that, despite its comparatively low-level nature, TR is still an abstract model, and is accordingly capable of many different physical implementations. Several physical implementation alternatives were touched on at various points in previous chapters.

15.3 Analysis

Clearly, TR differs radically from conventional direct-image approaches to implementation; to say it one more time, it's a *transform* technology, not a direct-image one. In this section, I want to describe in outline a variety of ways in which TR's transform technology might reasonably be characterized. The ways in question are all ones that I think can help explain the fundamental significance of the transform idea and can provide some insight, at least by analogy, into what TR is really all about.

The Logarithm Analogy

The first analogy is with logarithms (I mentioned this one briefly in Chapter 1). The idea is that, in a sense, TR's transform technology does for database processing what logarithms do for numeric processing. As I put it in Chapter 1:

> [Logarithms] allow what would otherwise be complicated, tedious, and time-consuming numeric problems to be solved by transforming them into vastly simpler but (in a sense) equivalent problems and solving those simpler problems instead ... [and] TR does the same kind of thing for data management problems.
>
> *—from Chapter 1*

Let's think about logarithms for a moment. We all know the pragmatic difficulties involved in carrying out typical arithmetic operations on large numbers:

> There is nothing more troublesome in mathematics than the multiplications, divisions, square and cubic root extractions of great numbers, which involve a tedious expenditure of time, as well as being subject to "slippery errors."

(These remarks are due to John Napier, the inventor of logarithms. The quote is from Jan Gullberg's book *Mathematics: From the Birth of Numbers,* W. W. Norton and Company, 1977.) Before Napier came along, such "multiplications, divisions, [and] ... root extractions" were, at best, hugely labor-intensive and time-consuming; at worst, they couldn't be done at all, because the amount of time required was prohibitive. (Does this sound familiar?)

Logarithms solved this problem. As already noted in the quote from Chapter 1, they did so by means of certain transforms: They allowed the objects of interest (numbers) to be transformed into a new—and incidentally unfamiliar—representation; that transform then allowed the operators of interest (multiply, divide, etc.) to be transformed into other, more familiar and much simpler, operators (add, subtract, etc.). For example, suppose we need to multiply two large numbers x and y. Then we proceed as follows:

1. First, we transform the numbers x and y into their logarithms x' and y', say. These transforms are done by looking the logarithms up in a precomputed table.

2. Next, we transform the operation of multiplying the two numbers into the much easier one of adding their logarithms $x'$and y', thereby obtaining a result z' say.

3. Finally, we transform z′ into the desired result z by looking up the antilogarithm of z' in another precomputed table.

Not only do the foregoing transforms make the problem much easier to solve, they also drastically reduce the amount of time involved—from multiplicative time to additive or linear time, in fact. (Again, does this sound familiar?)

Now let's get back to TR. TR also transforms the objects of interest—in this case, data files—into a new and unfamiliar representation, the Field Values and Record Reconstruction Tables. And then it transforms the operators of interest (value lookups and sequential searches) into more familiar and much more efficient operators, such as binary search, on those tables. The net effect, as with logarithms, is that:

- Problems that were difficult and excessively time-consuming with the traditional approach become easy and fast with the new approach.

- Problems that were effectively intractable with the traditional approach become feasible with the new approach.

- More generally, problems that required multiplicative time with the traditional approach require only linear time with the new approach, and problems that required linear time with the traditional approach require only logarithmic time with the new approach.

There's one more point to be made. With both logarithms and TR technology, **all of the "heavy lifting" is done just once, in advance**. In the case of logarithms, the lookup tables are precomputed; that is, the work of computing the actual logarithms and antilogarithms is done once, ahead of time, instead of being repeated over and over again every time we want to do some numeric calculation. In the same kind of way, with TR, the Field Values and Record Reconstruction Tables are also precomputed (at load time, in fact); that is, all of the data sorting and merging is done ahead of time, instead of over and over again every time we want to access the database (when executing some query, for example). And in both cases, doing the "heavy lifting" just once in advance translates into *overwhelming cost benefits*.[2]

The Copernican Analogy

There's another analogy that I think is helpful, too, and that's with the Copernican revolution—that is, the conceptual shift from the view in which the sun (and everything else) revolved around the earth to one in which the earth revolved around the sun instead.[3] As we all know, the perception that the sun revolves around the earth makes a kind of intuitive sense (and might even be defended, to some extent, on relativistic grounds), but it's certainly misleading if you want to understand the bigger picture. Well, in the same kind of way, the perception that relations consist primarily of tuples, and that those tuples then only secondarily contain individual data values, also makes sense—indeed, it's logically correct—but, again, it can be misleading if you want to understand the bigger picture.

Part of the problem here lies with books like this one. When such books show relations in pictorial form (that is, as SQL-style tables), for obvious reasons they always use examples that involve very few tuples (see any of the examples in the present book). As a consequence, the pictures in question always look like nice neat little rectangles, and the tuples and the attributes "carry equal weight," as it were. But relations in real databases aren't like that—at least, not usually; more usually, such relations involve comparatively few attributes but several millions or even billions of tuples, and the true picture becomes very long and skinny, almost more like a long thin piece of string than a "nice neat little rectangle." (Even with nice neat little rectangles, in fact, it's often psychologically easier to read down the columns rather than across the rows, a state of affairs that I think lends weight to the present argument.)

If we think of relations in this way, it becomes clear that it's the attributes, not the tuples, that are the real implementation problem; for example, we need to worry much more about how to search down the attributes than we do about how to search across the tuples. In other words, we need to make a conceptual shift from a tuple-oriented to an attribute-oriented point of view. Making that shift is, in a way, what the TR approach does: First, we break the records up into their constituent fields and sort the data by each field individually (of course, now I'm talking about the file analog of the relation in question), and only later do we worry about connecting the field values back together again to form the corresponding records. As we know, this approach is the exact opposite of the traditional direct-image approach, in which the records aren't broken up at all but are kept connected by physical contiguity. Thus, in the direct-image approach, the records are necessarily kept sorted in just one sort order, and redundant auxiliary structures then have to be introduced in order to obtain the effect of sorting the fields individually.

As we also know, it's this shift in perspective that allows us to introduce additional important techniques such as condensed and merged columns. (In this connection, I'd like to remind you in particular of the huge amount of data compression that those techniques make possible—recall the example from Chapter 8 of a relation representing drivers' licenses, where we had 20 million tuples but only ten different hair colors, perhaps.)

In a nutshell, the shift from a tuple- to an attribute-oriented point of view, like the shift afforded by the Copernican revolution, shows how things "really" fit together behind the scenes: In both cases, it's the "right" way to think about the problem, and it's the key to the "right" solution. What's more, the shift has surprisingly deep and powerful implications in both cases, implications that go far beyond the initial simple recognition of the shift as such to a truly fundamental conceptual transformation underneath the surface. In the case of TR in particular, that conceptual transformation seems to me to be the breakthrough that's needed in order to "do relational databases right"; instead of making comparatively small and incremental improvements, which is what database administrators, DBMS implementers, and database researchers have been doing for years, we can take a totally fresh approach to the problem, one that (as we've seen) provides huge performance—and other—benefits.

TR vs. Indexing

Now I want to say more about those redundant auxiliary structures; in particular, I want to say more about indexes. We've seen that TR does away with the need for indexes. Or does it? In what follows, I'd like to examine this question from a slightly different point of view.

Consider Fig. 15.1 (essentially a repeat of Fig. 4.1 from Chapter 4), which shows a possible file for suppliers, and Fig. 15.2, which shows a corresponding Permutation Table. Just to remind you, column S# in this latter table contains "the S# permutation"—that is, it shows that sorting the file of Fig. 15.1 by ascending supplier number returns the records in the sequence *4, 3, 5, 1, 2*—and similarly for the other columns.

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	S4	Clark	20	London
2	S5	Adams	30	Athens
3	S2	Jones	10	Paris
4	S1	Smith	20	London
5	S3	Blake	30	Paris

Fig. 15.1: A suppliers file

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	4	2	3	2
2	3	5	1	1
3	5	1	4	4
4	1	3	2	3
5	2	4	5	5

Fig. 15.2: A Permutation Table corresponding to the file of Fig. 15.1

Observe now that the S# permutation is an index, in a sense!—at least, it does provide the functionality of a conventional index.[4] And, of course, analogous remarks apply to the other permutations, too. Given that the permutation notion plays such a crucial role in TR, therefore, we might say, not that TR *dispenses* with indexes, but rather that indexes are *essential.* In fact, we might quite reasonably say that the TR internal structures—the Field Values Table and the Record Reconstruction Table—are obtained by building indexes on everything, connecting all of those indexes together (but storing the field values and the linkage information separately), and then throwing away the indexed file.

Of course, it's reasonable to talk in the way I've just been doing only if we have a very clear idea of what we really mean. Certainly TR does dispense with indexes as conventionally understood (and so it also dispenses with all of those undesirable consequences of such indexes as described in Chapter 2). After all, TR clearly does away with the notion of the stored file as a direct image of a user-level relation; it therefore also a fortiori does away with the notion of there being a distinction between such a file, on the one hand, and indexes over such a file, on the other. Thus, in the very act of doing away with the direct-image file, TR also does away with the idea of an index that points into such a file, which includes most or all of indexing as conventionally understood—and so I stand by my claim that TR abolishes the need for indexing in the conventional sense. Yet this abolition of indexing in the conventional sense is effectively accomplished by absorbing the functionality of such indexing into TR's own internal structures.

I'd like to expand a little on the foregoing. Conventional DBMSs involve a whole host of extremely difficult performance questions, some of which have to be answered by the database administrator (for example, "Which indexes should I build?") and some by the system optimizer (for example, "Which indexes should I use?"). And how those questions are answered typically has huge implications for system performance—meaning there are huge penalties to pay if the answers are wrong. Now, TR doesn't do away with such questions altogether, but it certainly does do away with many of them. And those questions that remain tend to be much easier to answer, and to have far less drastic performance implications, than their counterparts in conventional systems. In many cases, in fact, the implementation can probably answer the question for itself, or at least provide some sensible default answer; for example, user-level attributes of the same type might automatically cause a corresponding merged column to be built in the Field Values Table at the TR level. Automating decisions in this manner can obviously help to reduce the load on the database administrator still further. However, there will doubtless always be a need for some kind of "manual override" in certain situations.[5]

TR and Hyperplanes

The final characterization of TR that I want to discuss here is one you might find appealing if you happen to be mathematically inclined. Recall these remarks from Chapter 2:

> [A] **relation** can ... be pictured as a *table*. However, a relation **is not** a table. A picture of a thing isn't the same as the thing! In fact, the difference between a thing and a picture of that thing is another of the great logical differences ..
>
> *—from Chapter 2*

Although these remarks are undoubtedly true, it's also true that it can often be very convenient, informally, to think of a relation as a table. Tables are "user-friendly"; the fact that we can often think of relations, informally, as tables—sometimes more explicitly as "flat" or "two-dimensional" tables—makes relational systems intuitively easy to understand and use, and makes it intuitively easy to reason about the way such systems behave. Indeed, it's a very nice property of the relational model that its basic data structure, the relation, has such an intuitively attractive pictorial representation.

Unfortunately, many people have let themselves be blinded by that attractive pictorial representation into thinking that *relations as such* are "flat" or "two-dimensional." Perhaps even more unfortunately, this criticism has historically applied to DBMS implementers in particular—a fact that presumably accounts for the conventional direct-image approach to implementation found in most SQL systems on the market today. Indeed, we might quite reasonably characterize those direct-image implementations as "flat" or "two-dimensional," and we already know from Chapter 2 the problems that such implementations lead to.

But, in general, relations simply *aren't* two-dimensional. Rather, if a given relation has *N* attributes, then *each tuple in that relation represents a point in a certain N-dimensional space*—and the relation as a whole represents a set of such points. In other words, relations are *N*-dimensional, not two-dimensional! As I've written elsewhere (in quite a few places, in fact): **Let's all vow never to say "flat relations" ever again.**

Let's agree to refer to the points in a given *N*-dimensional space as "*N*-points," for brevity. Then the overall database can be regarded as a collection of such *N*-points. Of course, *N* will have different values for different points in the database, in general; and even when two points do have the same value for *N,* the points in question might be based on different dimensions. For example, the suppliers relation S and the shipments relation SPJ both contain tuples representing, specifically, 4-points; however, the underlying dimensions are S#, NAME, INTEGER, and CHAR in the case of suppliers, and S#, P#, J#, and INTEGER in the case of shipments.

Now let's focus for a moment on just one of those 4-points: let's say the 4-point representing the shipment for supplier S1, part P1, and project J1, with quantity 200. Consider some particular attribute value within that shipment tuple, say the supplier number S1. The TR representation of that attribute value involves a cell in the Field Values Table, and of course that cell is directly linked, via an appropriate zigzag or star, to the TR representations of all other attribute values from the same shipment tuple. What's more—thanks to the condensed-columns technique described in Chapter 8—*it's also directly linked, via other zigzags or stars, to the TR representations of all other attribute values in all other shipment tuples with the same supplier number.* In other words, all shipment 4-points with "the same S# coordinate" (if I might be allowed to talk in such terms) are directly linked together at the TR level. And, of course, the same is true for all shipment 4-points with the same P# coordinate, or the same J# coordinate, or the same QTY coordinate. In this sense, the TR representation of any given relation can reasonably be regarded as being **directly *N*-dimensional:** All "points" (that is, all tuples) in that relation that belong to the same "hyperplane" (see the next paragraph but one) are directly connected together at the TR level. By contrast, conventional direct-image implementations—precisely because they *are* direct-image and thus very close to the picture the user sees—can be regarded as being *two*-dimensional; to be specific, distinct points from the same hyperplane in such an implementation are represented independently of one another, and the connections among them therefore have to be explicitly represented by independent auxiliary structures such as indexes.

There's more. Thanks to the merged-columns technique described in Chapter 9, all shipment 4-points with a given S# coordinate can also be directly linked at the TR level to the (unique) supplier 4-point with the same S# coordinate. In fact, if we take the merged-columns idea to its logical conclusion, in which there's just one Field Values Table for the entire database, then we can say that *whenever two tuples are logically connected at the relational level (because they have some attribute value in common), then their internal representations are directly linked at the TR level.* In such a situation, TR can be regarded as **providing a directly *N*-dimensional representation of the entire database**. And, of course, it's that *N*-dimensional representation that (among other things) allows joins to be done in linear time, as we've already seen. It's also what allows both of the following tasks to be carried out efficiently: (a) Given a particular tuple, find all of its attribute values; (b) given a particular attribute value, find all of the tuples that contain it (see Chapter 11, Section 11.2).

Note: In case you're not familiar with the concept, let me explain what I mean by the term "hyperplane." In ordinary three-dimensional space, where points are identified by three coordinates *x, y,* and *z,* the set of all points with the same *x*-coordinate forms a *plane* (and likewise for the set of all points with the same *y*-coordinate or the same *z*-coordinate). More generally, in any given *N*-dimensional space, the set of all points with some given coordinate in common forms a *hyperplane*. Thus, to say that two *N*-points belong to the same hyperplane is just a fancy way of saying they have some common coordinate. Observe that any given *N*-point can be regarded as the intersection of *N* such hyperplanes (and the database as a whole can thus be thought of as a collection of intersections of hyperplanes).

15.4 A Review of the Benefits

In this section, I want to try and bring together in one place a summary of all of the many benefits I believe TR can provide. Some of those benefits have been discussed previously, others are new. *Note:* I should explain right away—as I've done elsewhere, in a somewhat similar context [34]—that the points that follow are all very much interwoven; sometimes they're even the same point in different guises. It's always hard to structure this kind of material completely orthogonally.

Be that as it may, I'd like to begin by quoting some extracts from reference [63] and offering some comments on those extracts. The first is, in part, a repeat of some text I quoted in Chapter 1:

> The present invention provides a new and efficient way of structuring databases [that supports] efficient query and update processing, [reduces] database storage requirements, and [simplifies] database organization and maintenance. Rather than [achieving] orderedness through increasing redundancy (that is, superimposing an ordered data representation on top of the original unordered representation of the same data), the present invention achieves orderedness through eliminating redundancy on a fundamental level.
>
> *—from the Initial Patent*

Comment: If you've managed to read the book this far, you should be in a position to understand exactly what's being claimed here and—I hope—agree with it.

— ◆ ◆ ◆ ◆ ◆ —

> [Conventional implementation approaches] contain key structural weaknesses, including high levels of unorderedness and redundancy, that have traditionally been regarded as unavoidable. For example, [data in such implementations] can be sorted ... on at most one criterion ... This limitation renders essential database functions such as querying ... on all criteria other than this privileged one ... awkward and overly resource-intensive ...[It] obscures natural and exploitable latent data relationships that are revealed by more ordered, condensed, and efficient data arrangements [and] leads to negative characteristics of state-of-the-art DBMSs such as unorderedness, redundancy, cumbersomeness, algorithmic inefficiencies, and performance instabilities.
>
> *—from the Initial Patent*

Comment: The "key structural weakness" of the first sentence here is, of course, the conventional direct-image style of implementation, in which user-level tuples map more or less directly to physically stored records (what I called in the previous section a "flat" or "two-dimensional" representation). As the quoted extract suggests, that direct-image style has simply been taken as a given in most prior work. **The breakthrough represented by the TR approach implies that numerous traditional assumptions underlying prior investigations into physical implementation are no longer valid.** The "more ordered, condensed, and efficient data arrangements" that TR technology makes possible are, of course, the condensed and possibly merged Field Values Tables and the associated Record Reconstruction Tables.

Let me also offer a few comments on those "negative characteristics of state-of-the-art DBMSs":

- *Unorderedness:* This one's obvious—the (unique) physical ordering of a conventional stored file clearly reflects at most one sensible logical ordering, and possibly none at all.

- *Redundancy:* There are at least two points here. First, the auxiliary structures (typically indexes) that are introduced to address the problem of unorderedness involve redundancy by definition. Second, the fact that column condensing and merging can't be used in a direct-image implementation means that the very same individual field values are typically repeated many times (possibly *very* many times) in storage, both within and across distinct stored files.

- *Cumbersomeness:* The vast array of auxiliary structures supported in conventional DBMSs—all of which are ad hoc to a degree—can certainly lead to cumbersome representations, representations that are difficult to design in the first place and can be difficult to change later, too. What's more, the DBMS code itself, which has to deal with all of these different representations, can be cumbersome and difficult to manage as well.

- *Algorithmic inefficiencies:* By way of example here, consider what's involved in implementing joins or aggregations in a TR system vs. what's involved in doing the same thing in a conventional system (see Chapter 10). The TR implementations are clearly much more efficient.

- *Performance instabilities:* And by way of example here, consider the difference in a conventional DBMS between doing a restriction operation when a suitable index exists vs. doing the same thing when it doesn't. Or consider the difference, again in a conventional DBMS, between doing a join when the stored versions of the relations involved are suitably sorted ahead of time vs. doing the same thing when they aren't. Again, the TR implementations are clearly much more efficient, and questions such as "Does a suitable index exist?" and "Is the data suitably sorted?" simply don't arise.

— ◆ ◆ ◆ ◆ ◆ —

> These supplementary structures are inherently, and often massively, redundant ... [and] typically grow to be overly lengthy, convoluted, and ... cumbersome to maintain, optimize, and especially update.
>
> *—from the Initial Patent*

Comment: The "supplementary structures" mentioned here are, of course, the auxiliary structures, typically indexes, introduced as noted previously to overcome the problem of "unorderedness" in conventional DBMSs. "Cumbersome to update": As we saw in Chapter 2, indexes might perhaps speed up queries, but they certainly slow down updates—partly because of the "inherent redundancies" also mentioned in the quote. Updates are faster in TR in part because there simply aren't any auxiliary structures to update.

— ◆ ◆ ◆ ◆ ◆ —

> [Data in TR] is much more easily manipulated than in traditional databases, often requiring only that certain entries in the [Record Reconstruction Table] be changed, with no copying of data.
>
> *—from the Initial Patent*

Comment: This extract refers primarily to the TR mechanism by which stored field values can be shared across stored records (see Chapters 8 and 9). Among other things, that mechanism allows the user to insert new tuples without new attribute values having to be physically inserted, and it allows the user to delete existing tuples without existing attribute values having to be physically deleted. In other words, "data manipulation" or update operations—meaning INSERT, DELETE, and UPDATE operations, as discussed in Chapter 6—can be very efficient, too.

— ◆ ◆ ◆ ◆ ◆ —

> [Certain] operations such as [histogram] analysis, data compression, and [obtaining a variety of distinct] orderings, which are computationally intensive in [conventional DBMSs], are obtainable immediately from the structures described herein. The invention also provides improved processing in parallel computing environments.
>
> *—from the Initial Patent*

Comment: The first sentence here is self-explanatory. Regarding the second sentence, I did mention at the very end of Chapter 3 that the TR tables are suitable for implementation in a multiprocessor environment, if such is available. The details are beyond the scope of this book; however, reference [63] does include several suggestions as to how parallel processing algorithms might be used to improve TR performance—for example, searches on columns of the Field Values Table might well be parallelized, and the same is true for the sorts that are needed to build the Field Values Table in the first place.

— ◆ ◆ ◆ ◆ ◆ —

Now let's revisit some of the problems that I claimed in Chapter 2 come with the use of indexes and other conventional auxiliary structures, and see in each case how TR overcomes those problems:

- *DBMS implementation complexity:* The complexity in question arises from the need for the DBMS to deal with many different auxiliary structures and associated access methods, and in particular from the consequent need for the optimizer to carry out the process of access path selection. The radical new TR internal structures (primarily the Field Values Table and Record Reconstruction Table) address this problem directly by eliminating unnecessary options at the physical level. For example, the optimizer doesn't have to decide whether or not to use an index, because there aren't any indexes, and that's because TR doesn't *need* any indexes (at least, not in the conventional sense—see the subsection entitled "TR vs. Indexing" in the previous section).

- *Stored data redundancy:* See the discussion of redundancy earlier in this section. *Note:* As explained in Chapters 13 and 14, *controlled* redundancy can have its uses. Of course, the kind of redundancy introduced by indexes and other auxiliary structures is controlled too—but it isn't *necessary*.

- *Additional storage space requirements:* Even if we limit our attention to the raw data alone and ignore the additional storage space requirements of auxiliary structures, the TR representation needs far less storage space than conventional structures (an 80 percent reduction is not atypical). In other words, the TR representation—especially when columns are condensed and merged—can be thought of as a highly *compressed* representation. What's more, the compressions in question have the effect of speeding up access as well as drastically reducing storage space, and the compression and decompression algorithms themselves are very fast.

- *Physical database design complications:* The fact that traditional DBMSs offer so many different auxiliary structures and access methods (see *DBMS implementation complexity* above) means that physical database design in such a system can be a very difficult task—especially since there are few solid guidelines for choosing between physical design alternatives. The TR structures directly address this problem, too, again by eliminating unnecessary physical design options.

- *Reorganization and tuning:* Following on from the previous point, traditional DBMSs typically require both (a) periodic physical database reorganization, and (b) constant tuning and retuning, in order to meet a variety of performance goals. The need for such reorganization and tuning is greatly reduced in TR—even eliminated altogether, in many cases.

  Note: In connection with the foregoing, it's worth mentioning that Codd himself is on record as stating (in reference [8]) that one of his objectives in introducing the relational model in the first place was "to simplify the potentially formidable job of the database administrator."[6] And, while it might be argued that the database administrator's job in today's SQL systems is simpler than it was in preSQL systems, I don't think anyone could reasonably claim that those SQL systems make that job *easy*. And it seems to me that the root cause of the problem is the direct-image style of implementation still found in those systems. The relevance of TR to Codd's objective is obvious.

- *Logical database design complications:* As I said in Chapter 2, physical design considerations should in principle have no impact on logical design, but in practice they usually do (once again because of the direct-image style of implementation). As a particularly egregious example, how often have we been told that we must "denormalize for performance"? As I've written elsewhere [27], denormalization (or something akin to denormalization, at any rate), if it *must* be done, should be done at the storage level, not the user level, but the almost one-to-one relationship between those two levels in conventional DBMSs has meant in practice that denormalization is invariably done at the user level too. As noted in Chapter 12, by contrast, in TR there's no need to denormalize at the user level at all, thanks primarily to the fact that joins are so fast. Hence, we can—at last—achieve the benefits of properly normalized designs, without having to pay any associated performance penalty. (As for denormalizing at the storage level, in TR the question doesn't even arise, because TR doesn't physically store relations, as such, at all.)

- *Query inefficiencies and overheads:* As explained in Chapter 2, the inefficiencies and overheads in question both occur because of the access path selection process. Since TR largely eliminates that process, the problem goes away.

- *Update inefficiencies and overheads:* The inefficiencies and overheads that occur with queries because of the access path selection process go away here too, for the same reason. Also, I noted earlier that indexes and other auxiliary structures slow down the update process; since TR has no such structures, that problem goes away too.

- *Data independence:* See the discussion in Section 15.2.

— ◆ ◆ ◆ ◆ ◆ —

Next I'd like to pull together a few miscellaneous points (they're mostly repeats of points I've already made elsewhere, but I'd still like to include them explicitly here):

- *Symmetric performance:* I explained in Chapter 5 that the relational model provided "symmetric exploitation" but that implementations prior to TR didn't provide any comparable symmetry in performance. But TR—even if it doesn't provide symmetry in performance 100 percent—certainly comes much closer to doing so than previous approaches ever did. This is because the TR data representations are themselves very symmetric in nature. To my mind, this fact is a virtue in itself—it adds an element of "rightness," as it were. As George Polya says (admittedly in a rather different context) in his book *How to Solve It* [62]: "Try to treat symmetrically what is symmetrical, and do not destroy wantonly any natural symmetry." I've always found this advice of Polya's a most valuable precept to follow in my own work on the relational model and related matters.

- *High performance:* Of course, TR doesn't just provide symmetric performance, it provides very *good* performance, too. Indeed, I opened Chapter 1 by saying that somebody had at last implemented the **go faster!** command, and we could now build DBMSs that were "blindingly fast." What's more, the performance advantage of TR over traditional systems increases dramatically with the complexity of the query; the more complex the query, the greater the gain (see Chapters 5 and 10). However, I would hope by now that you realize that high performance is only one of the many benefits that TR technology can provide. Certainly it's a critically important benefit, but, to say it again, it's not the only one.

- *Join performance:* In connection with the performance issue, I really have to repeat this particular point, because it's so significant (I'm tempted to say *staggering*): **Joins involve linear instead of multiplicative performance costs** (in other words, joins are *scalable*). As I said before, this fact by itself is sufficient in my opinion to place TR in a class of its own, quite apart from all of its other advantages.

- *Update performance:* We saw in Chapter 6 that the TR transforms don't imply good performance for retrieval at the expense of update; update performance is good, too.

- *Direct end-user access:* If performance is no longer an issue, then (as noted in Chapter 1) there's no need for the IT department to keep end-users shut out from their own data. In other words, end-users should be able to access the database directly for themselves, without having to go through the potential bottleneck of the IT department.

- *Concurrency control:* Now this is a topic I haven't discussed in this book at all, prior to this point; nor do I mean to get into a detailed discussion of it at this late juncture. The fact is, however, the TR internal structures form a good basis on which to implement sophisticated locking techniques, including (though not limited to) techniques that—like the retrieval and update operations discussed in the body of the book—essentially operate at the level of individual fields instead of records. What's more, locks are typically held for a much shorter time, precisely because queries and updates are so fast.

— ◆ ◆ ◆ ◆ ◆ —

Let me conclude this review of TR benefits with one more item from the TR documentation (it's based on some remarks in an internal document, but I've edited those remarks considerably here). I think it pretty much speaks for itself.

> With traditional DBMSs, the database administrator's job typically involves a complicated balancing act among four independent sets of requirements:

- *Query performance:* We want queries to perform well.
- *Update performance:* We want updates to perform well, too.
- *Storage space:* We want to keep the physical size of the database within reasonable bounds.

- *Optimizability:* Given that traditional optimizers are far from perfect, we want to stay within the bounds of what the optimizer can reasonably be expected to handle.

The trouble is, although the requirements are independent, the mechanisms for meeting them in conventional DBMSs typically aren't. *But TR is different*—TR replaces the usual series of vexing tradeoffs with dramatic improvements in all of these areas simultaneously.

15.5 Possible Future Developments

In this section, I'd like to speculate briefly about possible future applications of TR technology as I've described it in previous chapters. However, I must immediately make it clear that everything that follows is my opinion only; in particular, I'm categorically not "preannouncing" any TR products, nor am I disclosing anything from any of the follow-on patents. Rather, I just want to describe what might be thought of as a "future directions wish list" on my own part. What's more, I strongly suspect that some of the items in the list will require certain extensions to the TR model as described in previous chapters.

In a way, just about everything I want to say in what follows can be regarded as part of the same overall point:

Let's implement the relational model!

In other words, it's my belief that if we were to build a true relational DBMS, as Hugh Darwen and I have advocated in *The Third Manifesto* [40]—and I've tried to suggest all through this book that TR technology would be ideally suited to that task—then we would at least have the right framework for implementing all of the other items that I indicate below might be desirable. In fact, I want to go further; I want to suggest that trying to implement those desirable items in any other kind of framework is likely to prove more difficult than doing it right.[7]

Be that as it may, a true relational system would include direct support for all of the relational operators discussed in Chapter 10 and others besides, including at least *attribute rename, semijoin, semidifference, compose,* and *transitive closure* (TCLOSE).[8] It would also include direct, comprehensive, and systematic—that is, not ad hoc—support for *relational comparisons* (for example, the ability to test whether two relations are equal, whether one is a subset of another, and so on), *integrity constraints,* and *view updating*. All of these matters are discussed in detail in one or both of references [32] and [40].

Now, one aspect of the relational model that's very widely misunderstood is the following (and this observation is relevant to just about everything else I want to say in this section):

The relational model has absolutely nothing to say regarding the nature of the types over which relations are defined.

In particular, although people tend to think of those types as being very simple—integers, character strings, and so forth—there's absolutely nothing in the relational model that requires them to be limited to such simple forms. Thus, we might have an "audio recordings" type, a "geographic map" type, a "video recordings" type, an "engineering drawings" type, a "legal documents" type, a "geometric objects" type, and on and on.

Relation types are an extremely important special case of the foregoing. That is, the system should support types whose values are relations, and therefore should also support relations with attributes of such types; in other words, it should support relations with attributes whose values are relations in turn ("relation-valued attributes"). A simple example is given in Fig. 15.3.

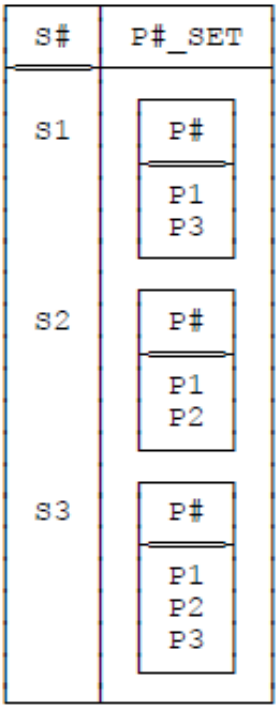


Fig. 15.3: A relation with a relation-valued attribute

Note: You might have encountered claims in the literature to the effect that relation-valued attributes violate the requirements of normalization (indeed, I'm on record as having made such claims myself—in earlier editions of reference [32] in particular). Such claims are incorrect, however. See reference [32] for further explanation.

Support for relation-valued attributes involves among other things support for operators, called *group* and *ungroup* in references [32] and [40], for mapping between relations without such attributes and relations with them. Also, it turns out that relation-valued attributes are important, at least conceptually, in connection with temporal database support (see the paragraphs immediately following).

Interval types are another important special case of types in general. In particular, such types provide the basis for proper *temporal database* support (which is a crucial aspect of data warehouse systems, albeit one that hasn't yet been implemented in existing data warehouse products so far as I know). For example, Fig. 15.4 gives an example of a *temporal relation;* that relation is supposed to show that certain suppliers supplied certain parts during certain intervals of time (you can read *d04, d06,* etc., as "day 4," "day 6," etc.; likewise, you can read [*d04:d06*] as "the interval from day 4 to day 6 inclusive," etc.). DURING in that relation is an example of an *interval-valued attribute*. *Note:* The similarity between those DURING intervals and the row ranges discussed elsewhere in this book isn't entirely coincidental.

S#	P#	DURING
S1	P1	[*d04:d06*]
S1	P1	[*d09:d10*]
S1	P3	[*d05:d10*]
S2	P1	[*d02:d04*]
S2	P1	[*d08:d10*]
S2	P2	[*d03:d03*]
S2	P2	[*d09:d10*]
S3	P1	[*d02:d05*]
S3	P2	[*d08:d10*]
S3	P3	[*d08:d10*]

Fig. 15.4: A relation with an interval-valued attribute

Support for interval-valued attributes (and hence for temporal databases) involves among other things support for generalized versions of the usual relational operators. For reasons that need not concern us here, those generalized operators are referred to in reference [42] as "U_" operators; thus, there's a *U_restrict* operator, a *U_join* operator, a *U_union* operator, and so on. *Note:* Those "U_" operators are all defined in terms of two new relational operators called *pack* and *unpack,* and those latter operators in turn are defined in terms of relation-valued attributes. As already noted, therefore, support for interval-valued attributes relies on support for relation-valued attributes, at least conceptually. Again, see reference [42] for further discussion.

As reference [42] also explains, proper and complete temporal database support additionally requires proper support for *type inheritance*.[9] Thus, I would like to see TR technology used, not just to implement the relational model as such, but also to implement the type system—including the inheritance portions of that system—defined for the relational model in reference [40]; in fact, I would argue that the type system in question *must* be implemented if temporal databases are to be supported properly and completely.

Of course, proper type support certainly includes support for user-defined types (see the earlier remarks regarding an "audio recordings" type, a "geographic map" type, etc). In fact, I've been assuming such support throughout this book—recall the user-defined types S#, NAME, and so on—but I haven't made a big deal of it. So let me do so now:

- The first point is that user-defined type support is the sine qua non—at the user or logical level, in fact, it's the sole distinguishing feature—of the so-called "object/relational" DBMSs (which in my opinion are, or at least ought to be, just relational DBMSs anyway; once again, see reference [40] for further discussion). Thus, if we use TR technology to build a true relational DBMS, we will necessarily have included full user-defined type support (for otherwise the DBMS wouldn't be a true relational DBMS, by definition), and so we will in fact have built an "object/relational" DBMS. Indeed, the term "object/relational" is little more than a marketing term anyway; it's needed only because the term "relational" has, sadly, been usurped (some might say destroyed) by SQL.

Perhaps I should mention one particular challenge that arises in connection with the foregoing. The fact is, some user-defined types have values that are very large and require a lot of storage (think of the type "video recordings," for example). Dealing with such types satisfactorily in a TR environment (or any other environment, come to that) looks like it might be an interesting implementation problem.

- Of course, user-defined type support includes user-defined operator support, too; that is, if we can define our own types, we must be able to define our own operators as well, because types without operators are useless. In particular, we must be able to define our own operators in connection with system- as well as user-defined types. *Note:* Reference [40] in fact insists on the provision of certain operators (with prescribed semantics) in connection with *every* type: "=" (equality comparison), ":=" (assignment), certain "selector" and "THE_" operators, and a few others. But, of course, the user is at liberty to define additional ones as well.

- Not all types—in particular, not all user-defined types—are "ordinal" types; that is, some types have no "<" operator defined for them, and hence have no logical ordering to their values. An example might be the type "geometric points in three-dimensional space"; clearly, it makes no sense to say that some point *p1* is less than (or greater than) some other point *p2*. So what can we do about about columns that correspond to such types in the Field Values Table? (Recall that columns in that table are generally supposed to be kept in sorted order.)

  Well, every type does at least have an "=" operator, even if it has no "<" operator, so at least we can always carry out the column condensing and merging described in Chapters 8 and 9, even if the columns aren't sorted as such. What's more, even for a type with no "<" operator, the implementation is always free to define a "<" operator for the internal (bit-string) *representation* of values of the type in question—so the Field Values Table columns can still be sorted (and binary searches can still be used), even if the ordering in question has no meaning at the user level.

Finally, I note that one type that's currently important (or at least fashionable) in the commercial world is the type *XML document*. And it seems to me that TR technology is particularly well suited to supporting such a type, because:

a) XML documents have a structure that's intrinsically hierarchic in nature;

b) Hence, given that joins are so fast in TR, it might make sense to map different hierarchic levels of a given XML document to different relations under the covers, and then to reconstruct the XML document as seen by the user by means of suitable joins, as and when required.

The TCLOSE operator mentioned earlier might be relevant here; so too might be relation-valued attributes.

Endnotes

1. The term *scalability* isn't very precisely defined in the literature, but to say something is scalable is basically just jargon for saying costs are linear. Here are two examples: (a) If hardware capacity and data volume are both increased by the same factor, then query response times should remain constant; likewise, (b) if hardware capacity and number of users are both increased by the same factor, then again query response times should remain constant.
2. I don't want to give the impression that "doing the heavy lifting at load time" implies that load performance must be bad in TR—it isn't. In fact, TR load times aren't all that different from load times in a conventional system, because it's the data read/write time that tends to dominate the process in both cases.
3. Thanks to Steve Tarin for suggesting this analogy.
4. To be more precise, it provides that functionality when considered in conjunction with the S# column of the Field Values Table, which contains the pertinent data values.
5. We saw a couple of examples (but only a couple!) in Part III of this book: Somebody has to choose characteristic or core fields, and somebody has to specify what if anything is to be redundantly stored. Note, however, that both of these decisions do at least have some potential for automation.
6. A summary and discussion of all of Codd's stated objectives for the relational model can be found in reference [35], Chapter 12.
7. To quote Gregory Chudnovsky, well-known mathematician and a member of the Required Technologies Scientific Advisory Board: "If you do it the stupid way, you will have to do it again" (from an article in *The New York Times*, December 24th, 1997).
8. TR technology should be particularly good for implementing TCLOSE, since (a) that operator consists essentially of an iterated compose, (b) compose in turn consists of a join followed by a projection, and (c) we already know that TR is good at joins and projections.

9) Of course, I'm well aware that type inheritance is supported in several commercial products already. In my opinion, however, most if not all of those implementations are logically flawed! This is not the place to get into details; if you want to know more, see reference [40].

Appendixes

Appendix A Exercises

Exercise 1: Use the following Field Values Table and Record Reconstruction Table to reconstruct the suppliers file:

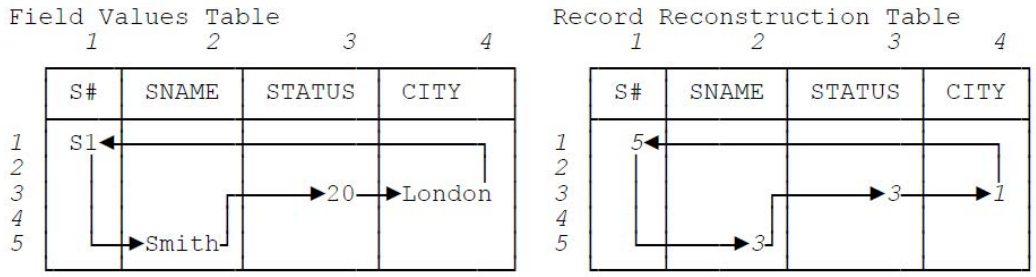
Field Values Table

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	S1	Adams	10	Athens
2	S2	Blake	20	London
3	S3	Clark	20	London
4	S4	Jones	30	Paris
5	S5	Smith	30	Paris

Record Reconstruction Table

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	5	4	4	5
2	4	5	2	4
3	2	2	3	1
4	3	1	1	2
5	1	3	5	3

The following diagram should serve to remind you how the reconstruction algorithm works (it shows the pointer rings for the record obtained by starting at cell [*1,1*] in each of the two tables):



The "first" reconstructed record is thus as shown below. You should be able to fill in the rest (begin with cell [*2,1*] in the Field Values Table, then cell [*3,1*], then cell [*4,1*], and finally cell [*5,1*]—in other words, proceed down column *1*).

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	S1	Smith	20	London
2				
3				
4				
5				

Your answer should look like Fig. 3.2.

Exercise 2: Use the following suppliers file and corresponding Permutation Table to build a Record Reconstruction Table:

File

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	S4	Clark	20	London
2	S5	Adams	30	Athens
3	S2	Jones	10	Paris
4	S1	Smith	20	London
5	S3	Blake	30	Paris

Permutation Table

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	4	2	3	2
2	3	5	1	1
3	5	1	4	4
4	1	3	2	3
5	2	4	5	5

Here's the algorithm:

Step 1: Let *PT* be the Permutation Table. Build a table *RRT* with the same number of rows and columns as *PT* and with all cells empty.

Step 2: For all records in the user file, do *Step 3*.

Step 3: For all columns of *PT,* do *Step 4*.

Step 4: Let the current record of the user file be the *r*th record, and let the current column of *PT* be the *j*th column. Let cell [*i,j*] of *PT* be the cell of column *j* that contains the record number *r*. At cell [*i,j*] of *RRT,* place the value *i'*, where cell [*i'*,*j*+1] of *PT* is the cell of column *j*+1 that contains the record number *r*. If column *j* is the last column, take column *j*+1 as the first column.

After this algorithm has been executed, table *RRT* is the desired Record Reconstruction Table:

	1	2	3	4
	S#	SNAME	STATUS	CITY
1				
2				
3				
4				
5				

Your answer should look like the Record Reconstruction Table shown in Exercise 1.

Exercise 3: Use the following suppliers file to build a corresponding Field Values Table:

File

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	S1	Smith	20	London
2	S2	Jones	10	Paris
3	S3	Blake	30	Paris
4	S4	Clark	20	London
5	S5	Adams	30	Athens

Field Values Table

	1	2	3	4
	S#	SNAME	STATUS	CITY
1				
2				
3				
4				
5				

Your answer should look like the Field Values Table shown in **Exercise 1**. Now construct a corresponding Permutation Table and (using that Permutation Table) a corresponding Record Reconstruction Table:

Permutation Table

	1	2	3	4
	S#	SNAME	STATUS	CITY
1				
2				
3				
4				
5				

Record Reconstruction Table

	1	2	3	4
	S#	SNAME	STATUS	CITY
1				
2				
3				
4				
5				

Does your Record Reconstruction Table look like Fig. 3.5? If not, why not?

Exercise 4: Use the following Permutation Table to build a corresponding Inverse Permutation Table:

Permutation Table

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	4	2	3	2
2	3	5	1	1
3	5	1	4	4
4	1	3	2	3
5	2	4	5	5

Inverse Permutation Table

	1	2	3	4
	S#	SNAME	STATUS	CITY
1				
2				
3				
4				
5				

Recall that if you think of any given permutation as a vector *V,* then the inverse permutation *V′* can be obtained in accordance with the rule that if $V[i] = i'$, then $V'[i'] = i$. Your answer should look like the Inverse Permutation Table shown in **Exercise 5**.

Exercise 5: Use the following Inverse Permutation Table to build a Record Reconstruction Table:

Inverse Permutation Table

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	4	3	2	2
2	5	1	4	1
3	2	4	1	4
4	1	5	3	3
5	3	2	5	5

Record Reconstruction Table

	1	2	3	4
	S#	SNAME	STATUS	CITY
1				
2				
3				
4				
5				

Here's the algorithm:

> Go to cell [*i,1*] of the Inverse Permutation Table. Let that cell contain the value *r;* also, let the next cell to the right, cell [*i,2*], contain the value *r′*. Go to the *r*th row of the Record Reconstruction Table and place the value *r′* in cell [*r,1*].

Executing this algorithm for *i = 1, 2, ..., 5* yields the entire S# column of the Record Reconstruction Table. The other columns are built analogously. Your answer should look like the Record Reconstruction Table shown in **Exercise 1**.

Exercise 6: Given the following suppliers file—

File

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	S4	Clark	20	London
2	S5	Adams	30	Athens
3	S2	Jones	10	Paris
4	S1	Smith	20	London

—check that the following Field Values Table and Record Reconstruction Table are correct:

Field Values Table

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	S1	Adams	10	Athens
2	S2	Clark	20	London
3	S4	Jones	20	London
4	S5	Smith	30	Paris

Record Reconstruction Table

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	4	4	4	4
2	3	2	2	3
3	2	1	3	1
4	1	3	1	2

Exercise 7: Given the following suppliers file—

File

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	S4	Clark	20	London
2	S5	Adams	30	Athens
3	S2	Jones	10	Paris
4	S1	Smith	20	London
5	S3	Blake	30	Paris
6	S6	Brady	30	Athens
7	S7	Patel	40	Haifa

—check that the following Field Values Table and Record Reconstruction Table are correct:

Field Values Table

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	S1	Adams	10	Athens
2	S2	Blake	20	Athens
3	S3	Brady	20	Haifa
4	S4	Clark	30	London
5	S5	Jones	30	London
5	S6	Patel	30	Paris
5	S7	Smith	40	Paris

Record Reconstruction Table

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	7	4	6	5
2	5	6	4	6
3	2	5	5	7
4	4	3	1	1
5	1	1	2	4
5	3	7	7	2
5	6	2	3	3

Exercise 8: Use the following Field Values and Record Reconstruction Tables to reconstruct the shipments file, starting at cell [*1,1*] of each of the two tables for the first record in that reconstruction and continuing down column *1*. Then do the same thing again, but this time going down column 2; and then again, going down column 3; and then again, going down column 4.

Field Values Table

	1	*2*	*3*	*4*
	S#	P#	J#	QTY
1	S1	P1	J1	100
2	S1	P1	J1	100
3	S2	P1	J1	200
4	S2	P1	J1	200
5	S2	P2	J2	200
6	S3	P2	J2	200
7	S3	P3	J2	500
8	S3	P3	J2	500
9	S3	P3	J2	500

Record Reconstruction Table

	1	*2*	*3*	*4*
	S#	P#	J#	QTY
1	*2*	*1*	*2*	*2*
2	*8*	*2*	*3*	*6*
3	*3*	*3*	*4*	*1*
4	*4*	*7*	*5*	*3*
5	*5*	*8*	*1*	*8*
6	*1*	*9*	*6*	*9*
7	*6*	*4*	*7*	*4*
8	*7*	*5*	*8*	*5*
9	*9*	*6*	*9*	*7*

Your answers should be as shown in Fig. 6.5.

Exercise 9: Use the following Inverse Permutation Table to build a "preferred" Record Reconstruction Table for shipments:

Inverse Permutation Table

	1	2	3	4
	S#	P#	J#	QTY
1	1	2	2	3
2	2	8	5	1
3	3	3	3	4
4	4	4	7	7
5	5	5	8	8
6	6	1	1	2
7	7	6	9	9
8	8	7	4	5
9	9	9	6	6

Record Reconstruction Table

	S#	P#	J#	QTY
1				
2				
3				
4				
5				
6				
7				
8				
9				

Your answer should look like Fig. 6.4.

Exercise 10: Given the following shipments file, show a Permutation Table corresponding to the following sort orders:

- For column S# : `S# - P# - J#`
- For column P# : `P# - J# - S#`
- For column J# : `J# - S# - P#`
- For column QTY : `QTY - S# - P# - J#`

File

	1	2	3	4
	S#	P#	J#	QTY
1	S1	P1	J1	200
2	S1	P3	J2	100
3	S2	P1	J1	200
4	S2	P1	J2	500
5	S2	P2	J2	500
6	S3	P1	J1	100
7	S3	P2	J2	500
8	S3	P3	J1	200
9	S3	P3	J2	200

Permutation Table

	1	2	4	5
	S#	P#	J#	QTY
1				
2				
3				
4				
5				
6				
7				
8				
9				

Use this Permutation Table to build a corresponding Inverse Permutation Table and a corresponding Record Reconstruction Table:

Inverse Permutation Table

	1	2	3	4
	S#	P#	J#	QTY
1				
2				
3				
4				
5				
6				
7				
8				
9				

Record Reconstruction Table

	1	2	3	4
	S#	P#	J#	QTY
1				
2				
3				
4				
5				
6				
7				
8				
9				

Your Record Reconstruction Table should look like Fig. 6.6. Check that this Record Reconstruction Table does exhibit the desired behavior regarding major-to-minor orderings.

Exercise 11: Use the following Field Values Table and Record Reconstruction Table to reconstruct the parts file. Start with column *5* in order to obtain the result in ascending city name sequence.

Field Values Table

	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	P1	Bolt [1:1]	Blue [1:2]	12.0 [1:2]	London [1:3]
2	P2	Cam [2:2]	Green [3:3]	14.0 [3:3]	Oslo [4:4]
3	P3	Cog [3:3]	Red [4:6]	17.0 [4:5]	Paris [5:6]
4	P4	Nut [4:4]		19.0 [6:6]	
5	P5	Screw [5:6]			
6	P6				

Record Reconstruction Table

	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	4	3	2	1	1
2	1	1	4	6	4
3	5	6	5	2	6
4	6	4	1	4	3
5	2	2	3	5	2
6	3	5	6	3	5

To remind you, here's the crucial revision to the reconstruction algorithm:

> Consider cell [*i,j*] of the Record Reconstruction Table. Instead of going to cell [*i,j*] of the Field Values Table, go to cell [*i',j*] of that table, where cell [*i',j*] is that unique cell within column *j* of that table that contains a row range that includes row *i*.

Your answer should look like Fig. 7.2, except that the records should appear in ascending city name sequence.

Exercise 12: Use the following Field Values Table and Record Reconstruction Table to reconstruct the parts file. Start with column *5* in order to obtain the result in ascending city name sequence.

Field Values Table

	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	P1	Bolt [1:1]	Blue [1:2]	12.0 [1:2]	London [1:3]
2	P2	Cam [2:2]	Green [3:3]	14.0 [3:3]	Oslo [4:4]
3	P3	Cog [3:3]	Red [4:6]	17.0 [4:5]	Paris [5:6]
4	P4	Nut [4:4]		19.0 [6:6]	
5	P5	Screw [5:6]			
6	P6				

Record Reconstruction Table

	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	4	1▪3	1▪2	1▪1	1▪1
2	1	2▪1	1▪4	1▪6	1▪4
3	5	3▪6	2▪5	2▪2	1▪6
4	6	4▪4	3▪1	3▪4	2▪3
5	2	5▪2	3▪3	3▪5	3▪2
6	3	5▪5	3▪6	4▪3	3▪5

To remind you, in those columns of the Record Reconstruction Table that include two row numbers instead of one, the first is the number of the desired row within the Field Values Table, and the second is the number of the next row to be inspected within the Record Reconstruction Table. As in **Exercise 11,** your answer should look like Fig. 7.2, except that the records should appear in ascending city name sequence.

Exercise 13: Given the following bill-of-materials file—

File

	1	2	3
	MAJOR_P#	MINOR_P#	QTY
1	P3	P4	3
2	P1	P3	4
3	P2	P4	8
4	P1	P4	1
5	P2	P5	6
6	P3	P6	4
7	P1	P2	2
8	P5	P6	3
9	P2	P3	3

—check that the following Field Values Table and Record Reconstruction Table are correct:

Field Values Table

	1	2	3
	MAJOR_P#	MINOR_P#	QTY
1	P1	P2	1
2	P1	P3	2
3	P1	P3	3
4	P2	P4	3
5	P2	P4	3
6	P2	P4	4
7	P3	P5	4
8	P3	P6	6
9	P5	P6	8

Record Reconstruction Table

	1	2	3
	MAJOR_P#	MINOR_P#	QTY
1	1	2	3
2	2	6	1
3	4	3	4
4	3	1	7
5	5	9	9
6	7	4	2
7	6	8	8
8	8	7	6
9	9	5	5

Note: The Record Reconstruction Table is intended to reflect the following sort orders:

- For column MAJOR_P#: `MAJOR_P# - MINOR_P# - QTY`
- For column MINOR_P#: `MINOR_P# - MAJOR_P# - QTY`
- For column QTY: `QTY - MAJOR_P# - MINOR_P#`

Exercise 14: Given the Field Values Table and Record Reconstruction Table from **Exercise 13,** check that the following condensed and expanded versions (respectively) are correct.

Field Values Table

	1	2	3
	MAJOR_P#	MINOR_P#	QTY
1	P1 [1:3]	P2 [1:1]	1 [1:1]
2	P2 [4:6]	P3 [2:3]	2 [2:2]
3	P3 [7:8]	P4 [4:6]	3 [3:5]
4	P5 [9:9]	P5 [7:7]	4 [6:7]
5		P6 [8:9]	6 [8:8]
6			8 [9:9]

Record Reconstruction Table

	1	2	3
	MAJOR_P#	MINOR_P#	QTY
1	1▪1	1▪2	1▪3
2	1▪2	2▪6	2▪1
3	1▪4	2▪3	3▪4
4	2▪3	3▪1	3▪7
5	2▪5	3▪9	3▪9
6	2▪7	3▪4	4▪2
7	3▪6	4▪8	4▪8
8	3▪8	5▪7	5▪6
9	4▪9	5▪5	6▪5

Exercise 15: Use the following Field Values Table and Record Reconstruction Table to reconstruct the bill-of-materials file. Start with column *1* in order to obtain the result in sequence by minor part number within major part number.

Field Values Table

	1 MAJOR_P# + MINOR_P#	*2* QTY
1	P1 [*1*:*3*] [:]	1 [*1*:*1*]
2	P2 [*4*:*6*] [*1*:*1*]	2 [*2*:*2*]
3	P3 [7:*8*] [*2*:*3*]	3 [*3*:*5*]
4	P4 [:] [*4*:*6*]	4 [*6*:7]
3	P5 [*9*:*9*] [7:7]	6 [*8*:*8*]
4	P6 [:] [*8*:*9*]	8 [*9*:*9*]

Record Reconstruction Table

	1 MAJOR_P#	*2* MINOR_P#	*3* QTY
1	*1*▪*1*	*2*▪*2*	*1*▪*3*
2	*1*▪*2*	*3*▪*6*	*2*▪*1*
3	*1*▪*4*	*3*▪*3*	*3*▪*4*
4	*2*▪*3*	*4*▪*1*	*3*▪7
5	*2*▪*5*	*4*▪*9*	*3*▪*9*
6	*2*▪7	*4*▪*4*	*4*▪*2*
7	*3*▪*6*	*5*▪*8*	*4*▪*8*
8	*3*▪*8*	*6*▪7	*5*▪*6*
9	*5*▪*9*	*6*▪*5*	*6*▪*5*

Your answer should be a file that's a direct image of relation MMQ as shown in Fig. 8.1.

Exercise 16: Given the following suppliers file—

File

	1 S#	2 SNAME	3 STATUS	4 CITY
1	S1	Smith	20	London
2	S2	Jones	10	Paris
3	S3	Blake	30	Paris
4	S4	Clark	20	London
5	S5	Adams	30	Athens

—check that the following Field Values Table and Record Reconstruction Table are correct:

Field Values Table

	1 S#	2 SNAME	3 STATUS	4 CITY
1	S1	Adams [1:1]	10 [1:1]	Athens [1:1]
2	S2	Blake [2:2]	20 [2:3]	London [2:3]
3	S3	Clark [3:3]	30 [4:5]	Paris [4:5]
4	S4	Jones [4:4]		
5	S5	Smith [5:5]		

Record Reconstruction Table

	1 S#	2 SNAME	3 STATUS	4 CITY
1	5	1▪5	1▪4	1▪5
2	4	2▪4	2▪2	2▪1
3	2	3▪3	2▪3	2▪4
4	3	4▪1	3▪5	3▪2
5	1	5▪2	3▪1	3▪3

Likewise, given the following shipments file—

File

	1 S#	2 P#	3 J#	4 QTY
1	S1	P1	J1	200
2	S1	P3	J2	100
3	S2	P1	J1	200
4	S2	P1	J2	500
5	S2	P2	J2	500
6	S3	P1	J1	100
7	S3	P2	J2	500
8	S3	P3	J1	200
9	S3	P3	J2	200

—check that the following Field Values Table and Record Reconstruction Table are correct:

Field Values Table

	1 S#	2 P#	3 J#	4 QTY
1	S1 [1:2]	P1 [1:4]	J1 [1:4]	100 [1:2]
2	S2 [3:5]	P2 [5:6]	J2 [5:9]	200 [3:6]
3	S3 [6:9]	P3 [7:9]		500 [7:9]

Record Reconstruction Table

	1 S#	2 P#	3 J#	4 QTY
1	1▪2	1▪1	1▪2	1▪2
2	1▪8	1▪2	1▪3	1▪6
3	2▪3	1▪3	1▪4	2▪1
4	2▪4	1▪7	1▪5	2▪3
5	2▪5	2▪8	2▪1	2▪8
6	3▪1	2▪9	2▪6	2▪9
7	3▪6	3▪4	2▪7	3▪4
8	3▪7	3▪5	2▪8	3▪5
9	3▪9	3▪6	2▪9	3▪7

Finally, check that the following merged Field Values Table is correct:

Field Values Table

	1 S#	2 SNAME	3 STATUS	4 CITY	5 P#	6 J#	7 QTY
1	S1[1:2]	Adams[1:1]	10[1:1]	Athens[1:1]	P1[1:4]	J1[1:4]	100[1:2]
2	S2[3:5]	Blake[2:2]	20[2:3]	London[2:3]	P2[5:6]	J2[5:9]	200[3:6]
3	S3[6:9]	Clark[3:3]	30[4:5]	Paris [4:5]	P3[7:9]		500[7:9]
4	S4[:]	Jones[4:4]					
5	S5[:]	Smith[5:5]					

Appendix B References and Bibliography

What follows is a consolidated list of references for the entire book. Let me immediately apologize for the embarrassingly large number of references to publications for which I'm the author or a coauthor; such a state of affairs is more or less unavoidable, however, given the nature of this book and the history of my own involvement in this field.

1. M. M. Astrahan et al.: "System R: Relational Approach to Database Management," *ACM Transactions on Database Systems 1,* No. 2 (June 1976).
2. E. Babb: "Implementing a Relational Database by Means of Specialized Hardware," *ACM Transactions on Database Systems 4,* No. 1 (March 1979).
3. R. G. G. Cattell: *Object Data Management* (revised edition). Reading, Mass.: Addison-Wesley (1994).
4. R. G. G. Cattell and Douglas K. Barry (eds.): *The Object Database Standard: ODMG 3.0*. San Francisco, Calif.: Morgan Kaufmann (2000).
5. E. F. Codd: "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks," IBM Research Report RJ599 (August 19th, 1969).
6. E. F. Codd: "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM 13,* No. 6 (June 1970). Republished in "Milestones of Research," *Communications of the ACM 26,* No. 1 (January 1982).
7. E. F. Codd: "Relational Completeness of Data Base Sublanguages," in Randall J. Rustin (ed.), *Data Base Systems: Courant Computer Science Symposia Series 6*. Englewood Cliffs, N.J.: Prentice-Hall (1972).
8. E. F. Codd: "Recent Investigations into Relational Data Base Systems," Proc. 1974 Congress, Stockholm, Sweden (1974).
9. E. F. Codd: *The Relational Model for Database Management Version 2*. Reading, Mass.: Addison-Wesley (1990).
10. E. F. Codd and C. J. Date: "Interactive Support for Nonprogrammers: The Relational and Network Approaches," Proc. ACM SIGMOD Workshop on Data Description, Access, and Control (Vol. II), Ann Arbor, Mich. (May 1974). Republished in C. J. Date, *Relational Database: Selected Writings*. Reading, Mass.: Addison-Wesley (1986).
11. Hugh Darwen (writing as Andrew Warden): "The Keys of the Kingdom," in C. J. Date, *Relational Database Writings 1985-1989*. Reading, Mass.: Addison-Wesley (1990).
12. Hugh Darwen: "The Nullologist in Relationland; *or,* Nothing Really Matters," in C. J. Date (with Hugh Darwen), *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).
13. Hugh Darwen: "The Duplicity of Duplicate Rows," in C. J. Date (with Hugh Darwen), *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).
14. Data Base Task Group of CODASYL Programming Language Committee: *Report* (April 1971).
15. C. J. Date: "A Critique of the SQL Database Language," in *Relational Database: Selected Writings*. Reading, Mass.: Addison-Wesley (1986).
16. C. J. Date: "What's Wrong with SQL?", in *Relational Database Writings 1985-1989*. Reading, Mass.: Addison-Wesley (1990).
17. C. J. Date: "SQL Dos and Don'ts," in *Relational Database Writings 1985-1989*. Reading, Mass.: Addison-Wesley (1990).
18. C. J. Date: "NOT Is Not 'Not'! (Notes on Three-Valued Logic and Related Matters)," in *Relational Database Writings 1985-1989*. Reading, Mass.: Addison-Wesley (1990).

19. C. J. Date: "EXISTS Is Not 'Exists'! (Some Logical Flaws in SQL)," in *Relational Database Writings 1985-1989*. Reading, Mass.: Addison-Wesley (1990).
20. C. J. Date: "Why Duplicate Rows Are Prohibited," in *Relational Database Writings 1985-1989*. Reading, Mass.: Addison-Wesley (1990). A considerably updated version of this article is "Double Trouble, Double Trouble" (in two parts), published on the website *www.dbdebunk.com* (April 2002).
21. C. J. Date: "Support for the Conceptual Schema: The Relational and Network Approaches," in *Relational Database Writings 1985-1989.* Reading, Mass.: Addison-Wesley (1990).
22. C. J. Date: "Three-Valued Logic and the Real World," in C. J. Date (with Hugh Darwen), *Relational Database Writings 1989-1991.* Reading, Mass.: Addison-Wesley (1992).
23. C. J. Date: "Oh No Not Nulls Again," in C. J. Date (with Hugh Darwen), *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).
24. C. J. Date: "Tables with No Columns" and "More on DEE and DUM," both in *Relational Database Writings 1991-1994*. Reading, Mass.: Addison-Wesley (1995).
25. C. J. Date: "An Inverted List System: DATACOM/DB," "A Hierarchic System: IMS," and "A Network System: IDMS," all in *Relational Database Writings 1991-1994*. Reading, Mass.: Addison-Wesley (1995).
26. C. J. Date: "Storage Structures and Access Methods," Appendix A of *An Introduction to Database Systems* (6th edition). Reading, Mass.: Addison-Wesley (1995).
27. C. J. Date: "The Normal Is So ... Interesting" (in two parts), *Database Programming & Design 10,* No. 11 (November 1997) and No. 12 (December 1997).
28. C. J. Date: "Why the 'Object Model' Is Not a Data Model," in C. J. Date (with Hugh Darwen and David McGoveran), *Relational Database Writings 1994-1997*. Reading, Mass.: Addison-Wesley (1998).
29. C. J. Date: "Object Identifiers vs. Relational Keys," in C. J. Date (with Hugh Darwen and David McGoveran), *Relational Database Writings 1994-1997*. Reading, Mass.: Addison-Wesley (1998).
30. C. J. Date: "Don't Mix Pointers and Relations!" and "Don't Mix Pointers and Relations—*Please!*", both in C. J. Date (with Hugh Darwen and David McGoveran), *Relational Database Writings 1994-1997*. Reading, Mass.: Addison-Wesley (1998).
31. C. J. Date: "Grievous Bodily Harm" (in two parts), *Database Programming & Design 11,* No. 5 (May 1998) and No. 6 (June 1998).
32. C. J. Date: *An Introduction to Database Systems* (7th edition). Reading, Mass.: Addison-Wesley (2000).
33. C. J. Date: "The Relational Model," Part II of reference [32].
34. C. J. Date: *WHAT Not HOW: The Business Rules Approach to Application Development.* Reading, Mass.: Addison-Wesley (2000).
35. C. J. Date: *The Database Relational Model: A Retrospective Review and Analysis*. Reading, Mass.: Addison-Wesley (2001).
36. C. J. Date: "Constraints and Predicates: A Brief Tutorial" (in three parts), published on the websites *www.dbdebunk.com* (May 2001) and *www.BRCommunity.com* (May/September/November 2001).
37. C. J. Date and E. F. Codd: "The Relational and Network Approaches: Comparison of the Application Programming Interfaces," Proc. ACM SIGMOD Workshop on Data Description, Access, and Control (Vol. II), Ann Arbor, Mich. (May 1974). Republished in C. J. Date, *Relational Database: Selected Writings*. Reading, Mass.: Addison-Wesley (1986).
38. C. J. Date and E. F. Codd: "Much Ado About Nothing," in C. J. Date, *Relational Database Writings 1991-1994*. Reading, Mass.: Addison-Wesley (1995).
39. C. J. Date (with Hugh Darwen): *A Guide to the SQL Standard* (4th edition). Reading, Mass.: Addison-Wesley (1997).

40. C. J. Date and Hugh Darwen: *Foundation for Future Database Systems: The Third Manifesto* (2nd edition). Reading, Mass.: Addison-Wesley (2000).
41. C. J. Date and Hugh Darwen: "Subtyping and Inheritance," Part IV of reference [39].
42. C. J. Date, Hugh Darwen, and Nikos A. Lorentzos: *Temporal Data and the Relational Model*. San Francisco, Calif.: Morgan Kaufmann (2002, to appear).
43. C. J. Date, Hugh Darwen, and David McGoveran: "Nothing to Do with the Case," in *Relational Database Writings 1994-1997*. Reading, Mass.: Addison-Wesley (1998).
44. C. J. Date, Hugh Darwen, and David McGoveran: "Up to a Point, Lord Copper," in *Relational Database Writings 1994-1997*. Reading, Mass.: Addison-Wesley (1998).
45. C. J. Date (with Colin J. White): *A Guide to DB2* (4th edition). Reading, Mass.: Addison-Wesley (1992).
46. Keith Devlin: *The Math Gene*. New York, N.Y.: Basic Books (2000).
47. Andrew Eisenberg and Jim Melton: "SQL:1999, Formerly Known as SQL3," *ACM SIGMOD Record 28,* No. 1 (March 1999).
48. Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong: "Extendible Hashing—A Fast Access Method for Dynamic Files," *ACM Transactions on Database Systems 4,* No. 3 (September 1979).
49. Richard Finkelstein: "Sybase IQ: Expressly for the Warehouse," *Database Programming & Design 9,* No. 12 (December 1996).
50. Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom: *Database System Implementation*. Upper Saddle River, N.J.: Prentice Hall (2000).
51. Anil K. Garg and C. C. Gotlieb: "Order-Preserving Key Transformations," *ACM Transactions on Database Systems 11,* No. 2 (June 1986).
52. Robert C. Goldstein and Alois J. Strnad: "The MacAIMS Data Management System," Proc. 1970 ACM SICFIDET Workshop on Data Description and Access, Houston, Texas (November 1970).
53. International Organization for Standardization (ISO): *Database Language SQL,* Document ISO/IEC 9075:1999. Also available as American National Standards Institute (ANSI) Document ANSI NCITS.135-1999.
54. Won Kim: "On Marrying Relations and Objects: Relation-Centric and Object-Centric Perspectives," *Data Base Newsletter 22,* No. 6 (November/December 1994).
55. Donald E. Knuth: *The Art of Computer Programming. Volume III: Sorting and Searching*. Reading, Mass.: Addison-Wesley (1973).
56. Vincent Y. Lum: "Multi-Attribute Retrieval with Combined Indexes," *Comunications of the ACM 13,* No. 11 (November 1970).
57. William C. McGee: "The IMS/VS System," *IBM Sys. J. 16,* No. 2 (June 1977).
58. David McGoveran: "Nothing from Nothing" (in four parts), in C. J. Date (with Hugh Darwen and David McGoveran), *Relational Database Writings 1994-1997*. Reading, Mass.: Addison-Wesley (1998).
59. T. H. Merrett: "Why Sort/Merge Gives the Best Implementation of the Natural Join," *ACM SIGMOD Record 13,* No. 2 (January 1983).
60. R. Morris: "Scatter Storage Techniques," *Communications of the ACM 11,* No. 1 (January 1968).
61. James K. Mullin: "Retrieval-Update Speed Tradeoffs Using Combined Indices," *Communications of the ACM 14,* No. 12 (December 1971).
62. George Polya: *How To Solve It* (2nd edition). Princeton, N.J.: Princeton University Press (1971).
63. U.S. Patent and Trademark Office: *Value-Instance-Connectivity Computer-Implemented Database*. US Patent No. 6,009,432 (December 28th, 1999).

64. Ben Shneiderman: "Reduced Combined Indexes for Efficient Multiple Attribute Retrieval," *Information Systems 2,* No. 4 (1976).
65. Alois J. Strnad: "The Relational Approach to the Management of Data Bases," Proc. 1971 IFIP Congress, Ljubljana, Yugoslavia (August 1971).
66. Peter J. Titman: "An Experimental Data Base System Using Binary Relations," Proc. IFIP TC-2 Working Conference on Data Base Management Systems (eds., Klimbie & Koffeman), Cargèse, Corsica (April 1974). Amsterdam, Netherlands: North-Holland (1974).
67. Transaction Processing Council (TPC): "TPC-H Result Highlights - HP 9000 Superdome Enterprise Server - Executive Summary," *www.tpc.org/results/individual_results/HP/hp_tpch_sd_750_es.prof.*
68. Patrick Valduriez: "Join Indices," *ACM Transactions on Database Systems 12,* No. 2 (June 1987).
69. Gio Wiederhold: *Database Design* (2nd edition). New York, N.Y.: McGraw-Hill (1983).