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.

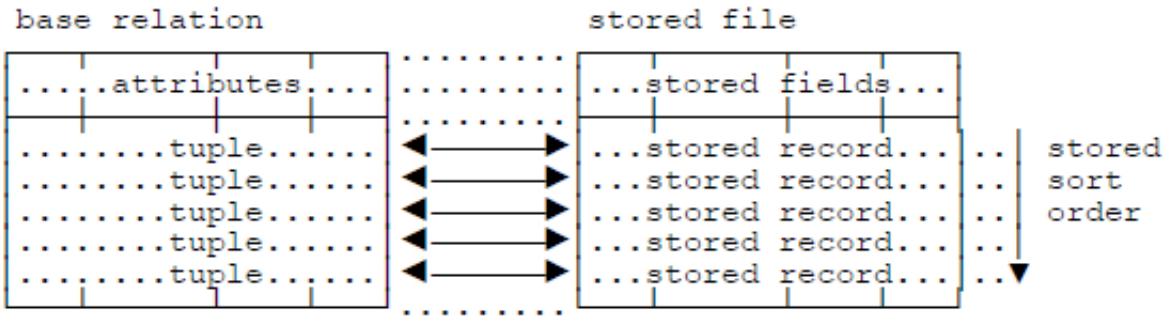

Fig. 1.1: Direct-image implementation

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.