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.

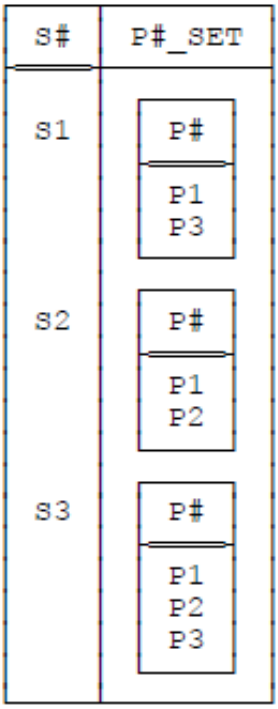

Fig. 15.3: A relation with a relation-valued attribute

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].