Part II: The Transrelational Model

4 Core Concepts

4.1 Introduction

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

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

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

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

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

- The *Field Values Table* contains the field values from a given file, rearranged in a way to be explained in Section 4.3.

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

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

4.2 The Crucial Idea

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

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

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

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

4.3 The Field Values Table

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.4 The Record Reconstruction Table

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

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

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

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

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

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

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

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

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

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

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

	S#	SNAME	STATUS	CITY
4	S1	Smith	20	London

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

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

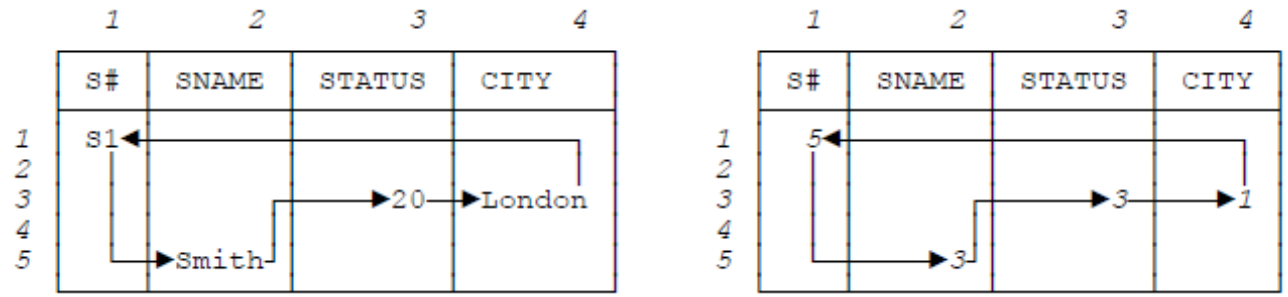


Fig. 4.4: Pointer rings (examples)

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

	S#	SNAME	STATUS	CITY
3	S2	Jones	10	Paris

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

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

Likewise, to implement this SQL query—

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

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

Ordering by Other Attributes

Now consider this SQL query:

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

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

	S#	SNAME	STATUS	CITY
3	S2	Jones	10	Paris

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

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

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

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

Equality Restrictions

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

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

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

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

and

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

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

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

Other User-Level Operations

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

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

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

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

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

4.5 Building the Record Reconstruction Table

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

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

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

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

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

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

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

	S#	SNAME	STATUS	CITY
4	S1	Smith	20	London

In other words, the sequence of Permutation Table cells

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

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

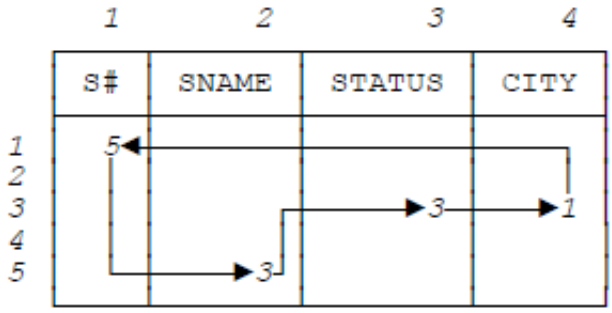
Now, the trouble with the foregoing algorithm—the algorithm, that is, for reconstructing records from the Permutation Table—is that the record numbers are effectively stored in each column of that table in random order. As a consequence, a sequential search is needed to find the desired record number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

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

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

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

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



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

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

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

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

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

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

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

4.6 The Record Reconstruction Table is not Unique

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

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

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

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

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

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

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

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

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

And here's the corresponding Record Reconstruction Table:

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

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

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

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

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

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

	S#	SNAME	STATUS	CITY
1	S4	Clark	20	London

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

Endnotes

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