13 File Banding

13.1 Introduction

We saw in Chapter 11 that the Field Values Table (or Tables, plural) will always be in memory at run time, at least to a first approximation. And we saw in Chapter 12 how to use file factoring to reduce the space requirements for the Record Reconstruction Table(s) as well; basically, what we do is decompose the original file—at least conceptually—so that we wind up with one "large" Record Reconstruction Table and several "small" ones. And the small Record Reconstruction Tables too will always be in memory at run time (again to a first approximation). So we're left with the large Record Reconstruction Table on disk. That large table can't be compressed any further, more or less by definition; in other words, if we regard its contents just as simple bit strings, then those bit strings are essentially random sequences of zeros and ones.[1] The techniques discussed in this chapter and the next are specifically aimed at getting the best possible performance out of that large table, despite the fact that it's necessarily disk-resident.

Before I go any further, I should make it clear that, although I call it "large," the table we're dealing with—in fact, the entire TR data representation, including the Field Values Table(s) and *all* of the corresponding Record Reconstruction Tables—is still likely to be far smaller than a conventional direct-image representation. Actual experiments have shown that a reduction of five to one is quite typical (if anything, that estimate is probably on the low side). And that's just for the raw data; when indexes and other auxiliary structures are taken into account, the direct-image space requirement can increase by another five-to-one ratio, possibly even more.[2] However, when I need to appeal to such matters later in this chapter, I'll stick, conservatively, to the five-to-one figure.

Anyway, to remind you from Chapter 11, the problem with the large table is that the zigzags in that table, even if their starting points are physically contiguous (as indeed they are, because the table is stored column-wise), quickly splay out to essentially random positions "all over the disk," with the consequence that we might have to do a separate seek for every point after the starting point every time we chase such a zigzag. File banding, or just *banding* for short, is a technique for addressing this problem. Here in outline is how it works:

- Starting with a given user-level relation and hence a corresponding file, we sort that file into order based on values of some **characteristic field** (or field combination; for simplicity, I'll assume throughout what follows that we're always dealing with a single characteristic field, barring explicit statements to the contrary).

- Next, we decompose that sorted file horizontally[3] into two or more subfiles of approximately equal size. Each subfile is smaller than the original file in the sense that it has fewer records (usually far fewer) than the original file did. *Note:* The official term for "horizontal subfiles" is *bands,* and I'll favor this latter term in subsequent sections. You can think of those bands or horizontal subfiles as *partitions,* if you like; note, however, that specifics of the partitioning in question are determined primarily by physical space requirements and only secondarily by values of the characteristic field. We'll see some implications of this state of affairs in the next section (in particular, we'll see that a given characteristic field value might appear in more than one band, something that couldn't happen if the partitioning were done purely on the basis of values of that field).

- We then represent each of those subfiles or bands by its own Field Values Table and its own Record Reconstruction Table. Because the bands are smaller than the original file, those Field Values and Record Reconstruction Tables too are smaller than their counterparts would have been for the original file. In fact, we choose the band size such that any given band can fit into memory in its entirety at run time, and we lay the bands out on the disk in such a way as to allow any given band to be streamed into memory as and when it's needed. (When I say the entire band fits into memory, what I'm mainly talking about is *the Record Reconstruction Table* for the given band, of course. If the Record Reconstruction Table for a given band can be entirely contained in memory, then all of the zigzags within that table will also be entirely contained in memory a fortiori, and—insofar as that particular table is concerned, at least—the splay problem thus won't arise.)

Note: Please understand that the foregoing account is deliberately somewhat simplified; I'll come back and explain later (in Section 13.4) how banding is *really* done. However, the foregoing explanation is accurate enough to serve as a basis for discussions prior to that section.

The structure of the chapter is as follows. Following this introductory section, I'll explain the basic idea of banding by means of a simple example in Section 13.2; then I'll elaborate on and generalize from that example in Section 13.3, and introduce the important idea of controlled redundancy. As already mentioned, in Section 13.4 I'll build on the ideas of previous sections to show how banding is really done. Finally, in Section 13.5, I'll discuss the concept of controlled redundancy in more detail.

13.2 A Simple Example

In the interests of "user-friendliness," I'll continue to work with the familiar parts example (or an extended version of that example, rather). Assume again—as in Chapter 12, Section 12.5—that we've factored the parts file into large and small files that look like this:

```
Large file        Small file

P#                CC#
PNAME             COLOR
WEIGHT            CITY
CC#
```

Sample values for these files are shown in Fig. 13.1 (an extended version of Fig. 12.2 from Chapter 12). *Note:* Of course, it's the large file we're interested in here, not the small one. In the figure, of course, that file is hardly very "large" (obviously, since it has just nine records); however, don't lose sight of the fact that if we're really supposed to be building on the example from Chapter 12, then the file is really supposed to have some *ten million* records. What's more, fields in that file—with the obvious exception of the introduced artificial identifier CC#—are supposed to be of high cardinality, meaning each such field has around ten million distinct values as well; in fact, the data in the large file isn't supposed to display any "statistical clumpiness" at all.

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Nut	12.0	*cc1*
2	P2	Bolt	17.0	*cc2*
3	P3	Screw	17.0	*cc3*
4	P4	Screw	14.0	*cc1*
5	P5	Cam	12.0	*cc4*
6	P6	Cog	19.0	*cc1*
7	P7	Nut	19.0	*cc1*
8	P8	Wheel	15.0	*cc5*
9	P9	Hinge	20.0	*cc3*

	1	2	3
	CC#	COLOR	CITY
1	*cc1*	Red	London
2	*cc2*	Green	Paris
3	*cc3*	Blue	Oslo
4	*cc4*	Blue	Paris
5	*cc5*	Black	Rome

Fig. 13.1: Parts factored into large and small files (sample values)

For the purpose of comparison with later figures (Figs. 13.5 and 13.6 in particular), Figs. 13.2 and 13.3 show the Field Values Table and a possible Record Reconstruction Table for the large file (only) of Fig. 13.1. *Note:* For simplicity, throughout the examples in this chapter, I'll omit the direct pointers into the corresponding Field Values Table that the Record Reconstruction Table might contain in practice (or not, as the case may be).

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Bolt [1:1]	12.0 [1:2]	cc1 [1:4]
2	P2	Cam [2:2]	14.0 [3:3]	cc2 [5:5]
3	P3	Cog [3:3]	15.0 [4:4]	cc3 [6:7]
4	P4	Hinge [4:4]	17.0 [5:6]	cc4 [8:8]
5	P5	Nut [5:6]	19.0 [7:8]	cc5 [9:9]
6	P6	Screw [7:8]	20.0 [9:9]	
7	P7	Wheel [9:9]		
8	P8			
9	P9			

Fig. 13.2: Field Values Table for the large file of Fig. 13.1

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	5	5	1	1
2	1	2	8	4
3	8	7	2	6
4	7	9	9	7
5	2	1	5	2
6	3	8	6	3
7	6	3	3	9
8	9	6	4	5
9	4	4	7	8

Fig. 13.3: Record Reconstruction Table for the large file of Fig. 13.1

Now, let's assume page size is one megabyte (a figure that, as we saw in Chapter 11, is quite realistic). For simplicity, let's assume also that band size is the same as page size (though a band might map into any number of consecutive disk pages, in general—it's just a matter of what page size we're working with and how much memory we have available). Without getting into tedious arithmetic details, then, for a file of 10 million records and hence 24-bit pointers in the Record Reconstruction Table, we might get (say) 80,000 rows from that table into each band, for an overall total of 125 bands. *Note:* Actually that figure of 80,000 rows per band is too low (and the figure of 125 bands is accordingly too high), for reasons I'll explain later in this section.

Moving now from reality back to our toy example, let's assume the band size we have to work with corresponds to a maximum of just *four* records from the large file ... That file is (let's assume) sorted on ascending part number—in other words, P# is the characteristic field in this example—and so (referring to Fig. 13.1) band one will correspond to records *1-4,* band two to records *5-8,* and band three to record *9* only. Figs. 13.4, 13.5, and 13.6 show, respectively, the banded version of the file, the Field Values Tables for those bands, and Record Reconstruction Tables for those bands.

		1	2	3	4
		P#	PNAME	WEIGHT	CC#
Band 1	1	P1	Nut	12.0	*cc1*
	2	P2	Bolt	17.0	*cc2*
	3	P3	Screw	17.0	*cc3*
	4	P4	Screw	14.0	*cc1*
Band 2	5	P5	Cam	12.0	*cc4*
	6	P6	Cog	19.0	*cc1*
	7	P7	Nut	19.0	*cc1*
	8	P8	Wheel	15.0	*cc5*
Band 3	9	P9	Hinge	20.0	*cc3*

Fig. 13.4: Large file decomposed into three bands

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Bolt [*1:1*]	12.0 [*1:1*]	*cc1* [*1:2*]
2	P2	Nut [*2:2*]	14.0 [*2:2*]	*cc2* [*3:3*]
3	P3	Screw [*3:4*]	17.0 [*3:4*]	*cc3* [*4:4*]
4	P4			
5	P5	Cam [*5:5*]	12.0 [*5:5*]	*cc1* [*5:6*]
6	P6	Cog [*6:6*]	15.0 [*6:6*]	*cc4* [*7:7*]
7	P7	Nut [*7:7*]	19.0 [*7:8*]	*cc5* [*8:8*]
8	P8	Wheel [*8:8*]		
9	P9	Hinge [*9:9*]	20.0 [*9:9*]	cc3 [*9:9*]

Fig. 13.5: Field Values Tables for the bands of Fig. 13.4

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	2	3	1	1
2	1	1	2	4
3	4	2	3	2
4	3	4	4	3
5	5	5	7	6
6	6	7	8	7
7	7	8	5	5
8	8	6	6	8
9	9	9	9	9

Fig. 13.6: Record Reconstruction Tables for the bands of Fig. 13.4

Several points arise from the foregoing simple example:

- First of all, I need to own up to a slight terminological inexactitude on my part. When I first mentioned banding in the introduction to this chapter (also in Chapter 11), I said we decomposed the file into horizontal subfiles called bands, and so "band" was a concept at the file level. As you might have noticed, however, I subsequently started talking about bands fitting into memory at run time, and so "band" became a concept at the TR level (or possibly at some lower and more physical level still). However, it's very convenient to be able to use the same term *band* at these different levels of abstraction within the overall system, and—trusting that the practice won't cause confusion—I intend to continue doing the same thing throughout the remainder of this chapter.

- Next, observe that the Record Reconstruction Table for any given band does indeed involve pointers that are local to the band in question. In the figures I've stressed this fact by using *1-4* as the sole legal pointer values for band one, *5-8* as the sole legal pointer values for band two, and *9* as the sole legal pointer value for band three. In practice, of course, pointer values need only be unique within the relevant band (since the whole object of the exercise is not to have pointers out of one band into another).

- Precisely because pointers need now be unique only within the relevant band instead of within the entire file, they need fewer bits than they did before (before we did the banding, I mean). To revert for a moment to the example of a file with ten million records: Without banding, pointers are 24 bits, as we know; with banding, however, if one band corresponds to 80,000 records as suggested above, then pointers need be only 17 bits instead of 24—another significant space saving (and, be it noted, one that applies to the large Record Reconstruction Table specifically, a most gratifying state of affairs). *Note:* These facts explain why the figure of 80,000 rows per band quoted earlier in this section was too low. A more reasonable figure would be 115,000 or so (making the total number of bands 85 or so instead of 125).

- Note that banding does suffer from the drawback that it potentially introduces a degree—a tiny degree—of redundancy into the stored data. Without banding (but with condensing), no field value ever appears more than once in the Field Values Table. With banding, however, the same field value might simultaneously appear in several distinct bands (though never more than once per band). The WEIGHT value 12.0 is a case in point in Fig. 13.5: It appears in both band one and band two. *Note:* Such redundancy could even apply to the characteristic field, if values of that field aren't unique (clearly this can't happen in the example, though, because {P#} is a key—in fact, the only key—for the parts relation). More important, however, note that this particular drawback (the possibility of a tiny amount of redundancy, that is) disappears anyway if the approach to be described in Section 13.4 is adopted.

- Another drawback, perhaps more serious than the previous one, is the following: Since we no longer have just one Record Reconstruction Table for the entire file, it follows a fortiori that we can't have a "preferred" Record Reconstruction Table for the entire file (as described in Chapter 7) that provides major-to-minor orderings over all of the fields. However, it's at least true that each of the local Record Reconstruction Tables can be a "preferred" one so far as the records that belong to the band in question are concerned. The Record Reconstruction Tables of Fig. 13.6 are preferred ones in this sense.

- Equality or range queries based on the characteristic field can now be handled very efficiently by going directly to the relevant band or bands. (I'm assuming here that the system will keep certain metadata in memory, saying, for example, that band one has information regarding parts with part numbers in the range P1-P4, band two has information regarding parts with part numbers in the range P5-P8, and so on.) Even if several bands are involved, each can be processed independently of the rest (possibly even in parallel). And, of course, once a given band has been streamed into memory, record reconstruction within that band is a purely in-memory operation. Among other things, therefore, banding can provide functionality analogous, somewhat, to that provided by a conventional clustering index on the characteristic field (see Chapter 2 if you need to refresh your memory regarding clustering indexes).

- Note finally that banding does **not** have to be done "by hand"; rather, it can be done automatically during the load process (like factoring in the previous chapter), using built-in heuristics and statistical data analyses that are also done at load time. In other words, the benefits of banding, like those of factoring, can be obtained automatically, without any need for human decisions (except as noted in Section 13.5 below).

A Small Digression

You might have noticed something interesting has happened to band three in the example. Band three corresponds to a single record; it therefore contains a Field Values Table of just a single row and a Record Reconstruction Table of just a single row. Observe now that:

- In the case of the Field Values Table—ignoring the row ranges, which are clearly pretty pointless here anyway—the single row is effectively *a direct-image representation* of the record in question: namely, the "large-file" record for part P9 (see Fig. 13.1).

- In the case of the Record Reconstruction Table, the single row contains a "zigzag" that's in fact a straight line—necessarily so, of course. But a zigzag that's a straight line isn't all that useful, because the pertinent record can easily be "reconstructed" without it. To be specific, the field values of that record are now linked by physical contiguity in the Field Values Table (or something that might be thought of as akin to physical contiguity, at any rate).

It should be clear from the foregoing discussion that if the band size were such that every band corresponded to a single record, then we would be getting rather close to a direct-image representation of the entire file. It's interesting to observe, therefore, that—in a sense—the conventional direct-image style of representation might be regarded as just a highly suboptimal special case of the much more general TR style of representation. I'll leave this observation as something for you to meditate on at your leisure.

13.3 Elaborating on the Example

Let's get back to the main thread of our discussion. We've seen that, with banding, equality and range queries based on the characteristic field—the P# field, in the example—can be handled very efficiently, because the implementation can go directly to the relevant band or bands, stream it or them into memory, and complete processing of the query as a pure in-memory operation. But what about queries based on some other field? For example, consider the following SQL query:

```
SELECT DISTINCT P.P#
FROM P
WHERE P.WEIGHT = 12.0 ;
```

As I pointed out in the previous section, the WEIGHT value 12.0 appears in both band one and band two. In the worst case, of course, the same WEIGHT value could appear in *every* band, precisely because the original file was sorted on P#, not WEIGHT. Now, it might at least be possible for the implementation to know, from the Field Values Table(s), just which bands a given value does in fact appear in; but if it doesn't (and possibly even if it does), a query like the one just shown will effectively require a scan of the entire file, and performance might thus be poor. As noted in Chapter 11, in other words, the symmetric performance property is lost.

One way to address this problem is to band the original file twice, once using P# as the characteristic field and once using WEIGHT. In this way, we can have one set of banded Record Reconstruction Tables corresponding to the P# sort order and another set corresponding to the WEIGHT sort order: a form of **controlled redundancy** (see Section 13.5 for further discussion). Figs. 13.7, 13.8, and 13.9 show, respectively, the banded version of the file, the Field Values Tables for those bands, and Record Reconstruction Tables for those bands, if we sort and band by part weight as suggested (more precisely, by part number within part weight). *Note:* I'm still assuming four records per band, of course.

		1	*2*	*3*	*4*
		P#	PNAME	WEIGHT	CC#
Band 1	*1*	P1	Nut	12.0	*cc1*
	2	P5	Cam	12.0	*cc4*
	3	P4	Screw	14.0	*cc1*
	4	P8	Wheel	15.0	*cc5*
Band 2	*5*	P2	Bolt	17.0	*cc3*
	6	P3	Screw	17.0	*cc3*
	7	P6	Cog	19.0	*cc1*
	8	P7	Nut	19.0	*cc1*
Band 3	*9*	P9	Hinge	20.0	*cc3*

Fig. 13.7: Banding parts by weight

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Cam [1:1]	12.0 [1:2]	cc1 [1:2]
2	P4	Nut [2:2]	14.0 [3:3]	cc4 [3:3]
3	P5	Screw [3:3]	15.0 [4:4]	cc5 [4:4]
4	P8	Wheel [4:4]		
5	P2	Bolt [5:5]	17.0 [5:6]	cc1 [5:6]
6	P3	Cog [6:6]	19.0 [7:8]	cc2 [7:7]
7	P6	Nut [7:7]		cc3 [8:8]
8	P7	Screw [8:8]		
9	P9	Hinge [9:9]	20.0 [9:9]	cc3 [9:9]

Fig. 13.8: Field Values Tables for the bands of Fig. 13.7

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	2	2	1	1
2	3	1	3	2
3	1	3	2	3
4	4	4	4	4
5	5	5	7	7
6	8	7	8	8
7	6	8	5	5
8	7	6	6	6
9	9	9	9	9

Fig. 13.9: Record Reconstruction Tables for the bands of Fig. 13.7

The query

```
SELECT DISTINCT P.P#
FROM P
WHERE P.WEIGHT = 12.0 ;
```

can now be implemented by going directly to band one (only) in the foregoing banding, and symmetry of performance is restored.

13.4 How it's *Really* Done

Now I need to clean up my act ... I said in Section 13.1 that we build a separate Field Values Table and Record Reconstruction Table for each band in the banded file. In fact, however, that statement isn't quite accurate. What we really do is this: First, we build a single Field Values Table for the entire file in the usual way; then, for each band, we build a band-local "Field Values Table" (or an analog of such a table, rather) that contains, not field values as such, but rather *pointers into* the overall Field Values Table for the whole file.

Let's see how this works out in our example. First, Fig. 13.10 (a copy of Fig. 13.2) shows the Field Values Table for the entire "large file" from Fig. 13.1:

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Bolt [1:1]	12.0 [1:2]	cc1 [1:4]
2	P2	Cam [2:2]	14.0 [3:3]	cc2 [5:5]
3	P3	Cog [3:3]	15.0 [4:4]	cc3 [6:7]
4	P4	Hinge [4:4]	17.0 [5:6]	cc4 [8:8]
5	P5	Nut [5:6]	19.0 [7:8]	cc5 [9:9]
6	P6	Screw [7:8]	20.0 [9:9]	
7	P7	Wheel [9:9]		
8	P8			
9	P9			

Fig. 13.10: Field Values Table for the large file of Fig. 13.1 (same as Fig. 13.2)

Let's assume once again that we want to sort and band on part number. Here then (repeated from Fig. 13.4) is the first band:

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Nut	12.0	*cc1*
2	P2	Bolt	17.0	*cc2*
3	P3	Screw	17.0	*cc3*
4	P4	Screw	14.0	*cc1*

Here's the Field Values Table for this band as given in Fig. 13.5:

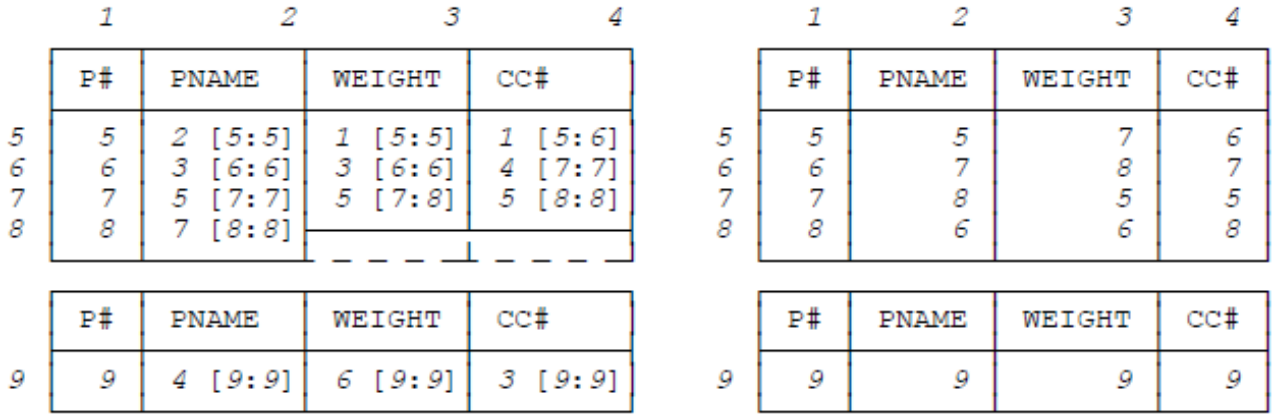
	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Bolt [1:1]	12.0 [1:1]	*cc1* [1:2]
2	P2	Nut [2:2]	14.0 [2:2]	*cc2* [3:3]
3	P3	Screw [3:4]	17.0 [3:4]	*cc3* [4:4]
4	P4			

And here's the corresponding analog of this Field Values Table with pointers into the main Field Values Table of Fig. 13.10 instead of actual field values (for convenience, I've extracted the corresponding Record Reconstruction Table from Fig. 13.6 and shown it on the right):

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	1	1 [1:1]	1 [1:1]	1 [1:2]
2	2	5 [2:2]	2 [2:2]	2 [3:3]
3	3	6 [3:4]	4 [3:4]	3 [4:4]
4	4			

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	2	3	1	1
2	1	1	2	4
3	4	2	3	2
4	3	4	4	3

Here for completeness are the Field Values Table analogs and Record Reconstruction Tables for the other two bands:



	1	2	3	4
	P#	PNAME	WEIGHT	CC#
5	5	2 [5:5]	1 [5:5]	1 [5:6]
6	6	3 [6:6]	3 [6:6]	4 [7:7]
7	7	5 [7:7]	5 [7:8]	5 [8:8]
8	8	7 [8:8]		

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
9	9	4 [9:9]	6 [9:9]	3 [9:9]

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
5	5	5	7	6
6	6	7	8	7
7	7	8	5	5
8	8	6	6	8

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
9	9	9	9	9

By way of example, let's consider the problem of reconstructing the record for part P6, say. The sequence of events is as follows:

- First we perform an in-memory look-up for part P6 in the Field Values Table of Fig. 13.10, and we discover that the record we want passes through cell [*6,1*] of that table.

- Knowing that part P6 falls into band two, we adjust that [*6,1*] to [*2,1*] to account for the fact that band one contains four parts. *Note:* Actually this step is unnecessary in our example, because I've numbered the rows *1-4* within band one, *5-8* within band two, and *9* within band three, and so we already know—albeit unrealistically—that [*6,1*] refers to a cell within band two. For definiteness and clarity, I'll continue to rely on that unrealistic assumption that row numbers are globally unique as shown in the figures.

- We stream band two into memory if it's not already there.

- Next, we follow the zigzag passing through cell [*6,1*] of the Record Reconstruction Table in band two (an in-memory process). That zigzag looks like this:

  ```
  [6,1], [6,2], [7,3], [5,4]
  ```

  We use in-memory look-ups to determine that the pointers (row numbers) in the corresponding cells of the corresponding Field Values Table analog are:

  ```
  6, 3, 5, 1
  ```

- We therefore go to cells

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

  of the Field Values Table of Fig. 13.10 (another in-memory process). The corresponding values are:

  P6, Cog, 19.0, *cc1*

  Reconstruction of the desired record is now complete.

At this point I'd like to remind you of something. In Chapter 5 (Section 5.6), I pointed out that row numbers can be regarded as *surrogates* for field values. In the record reconstruction example just now, for instance, the sequence of row numbers

6, 3, 5, 1

can be regarded as surrogates for the sequence of field values

P6, Cog, 19.0, *cc1*

Thus, we might reasonably think of the band-local Field Values Table analogs as containing, not actual field values as such (as indeed we now know), but surrogates for such field values instead.

13.5 Controlled Redundancy

In Section 13.3, I said that banding the parts file twice, once on part number and once on part weight, amounted to a form of controlled redundancy. Now, as I'm sure you know, redundancy in what's stored is usually considered to be a bad thing—not least because it can lead to inconsistencies. However, it's only when the redundancy is *uncontrolled* that it's unquestionably bad. *Controlled* redundancy—in other words, redundancy that's deliberately introduced and properly managed—is (or can be) fine; indeed, there are many sound reasons, both business and technical reasons, for storing several copies of the same data. But it does need to be understood that "controlled" here means that

a) The DBMS must be aware of the redundancy if it exists,

and more particularly that

b) The DBMS must take responsibility for "propagating updates" and maintaining data consistency (in other words, the redundancy must effectively be hidden from the user). I'll come back to this point in a few moments.

Let's return for a moment to the example from Section 13.3. Banding the parts file twice as suggested in that section clearly means we're going to need twice as much storage space. But I remind you that the data is already highly compressed; typically, as I pointed out in the introduction to this chapter, the TR representation requires only some 20 percent of the space required for a direct-image representation of the same data. So we can afford to band and store the original file *five different ways* and still not require any more storage than a conventional system does—and that's before the storage for indexes and other auxiliary structures is taken into account, in the direct-image case. *Note:* The point is worth making that indexes and the like effectively constitute a form of controlled redundancy in conventional systems anyway. And I've already mentioned the amount of storage space that kind of redundancy can involve (as noted in Section 13.1, a further fivefold increase is not at all atypical).

The kind of redundancy we're talking about in TR, then, is (to repeat) only redundancy on top of something that's already highly compressed. What's more, it's only redundancy on top of that portion of the data that can't be handled by the factoring techniques of Chapter 12. And what's more again, *it's the right kind of redundancy*. It's not the field values that are stored redundantly; rather, it's the linkage information. (No field value is ever stored more than once on the disk—assuming, of course, that column condensing and merging is done, as it certainly will be in a disk implementation. Contrast the situation in a direct-image system, with its indexes and other auxiliary structures, where it's virtually guaranteed that the very same field values will be stored many, many times over.) Storing the linkage information in different ways in a disk-based system is precisely what lets us achieve symmetry of performance in such a system.

Note, moreover, that it's a comparatively straightforward matter to decide what redundancies to store (in other words, to decide what sortings and bandings should be done). Detailed knowledge of the internal workings of the system is *not* required; nor is detailed knowledge of exactly the kinds of queries that users will submit. All that's needed is a general sense as to which fields are the pragmatically important ones—and this knowledge could even be obtained by the system itself, by analyzing actual or typical query sequences. Of course, if the system doesn't determine for itself what sortings and bandings are desirable, then the database administrator will have to tell it; in other words, human decisions will be required. But (to repeat) I don't think the decisions in question are very difficult ones.

Now, the obvious drawback to storing data redundantly is the impact it's likely to have on updates: If *N* distinct copies are stored of some given data item *X*, then an update to any one of those copies must be propagated to all the rest. But this is a much more tractable problem in TR than it usually is for at least two reasons:

- First, I've already said that field values as such *aren't* stored redundantly (so that "data item *X*" in my example just now can't be a field value in TR). Note in particular that update propagation can't affect index entries in TR, because there aren't any index entries in TR. Thus, update propagation is primarily a question of maintaining the pointers that are used in record reconstruction.

- Second, updates in the real world typically affect only a tiny portion of the overall database, as explained in Chapter 6. Typically, TR exploits this fact by keeping most of the database static for most of the time and segregating all updates in a much smaller overflow structure of their own (see Chapter 6, Section 6.5). That overflow structure is thus the only portion of the database that needs to be maintained in real time, and hence the only place where anything like update propagation has to be done in real time.

One last point in connection with redundancy in TR: Even if the database as stored does involve redundancy in the form of different bandings, there's no need to copy all of those different bandings to backup storage every time a full database backup is to be taken. All that's necessary is to copy just one of the bandings (the others can be recreated from that one).

Endnotes

1. To a first approximation yet again. In fact, as we saw in Chapter 11 (Section 11.5), there are compression techniques that do still work, even on that large table. For simplicity, however, I won't attempt to incorporate any of those techniques into my examples in this chapter.
2. This is not an overstatement. For example, in a report on the performance of a certain well-known SQL product on the standard TPC-H benchmark, reference [67] shows a raw data set of three terabytes expanding out to occupy nearly 60 terabytes of disk space, a twentyfold increase.
3. Contrast factoring, where we decompose files vertically (see the previous chapter). Indeed, just as there are certain parallels between factoring and conventional projection/join normalization, so there are certain parallels between banding and what might be called *restriction/union* normalization. Restriction/union normalization is a logical design technique, not much researched at the time of writing and certainly not yet much used in practice, in which the decomposition operator is restriction and the recomposition operator is union [32].