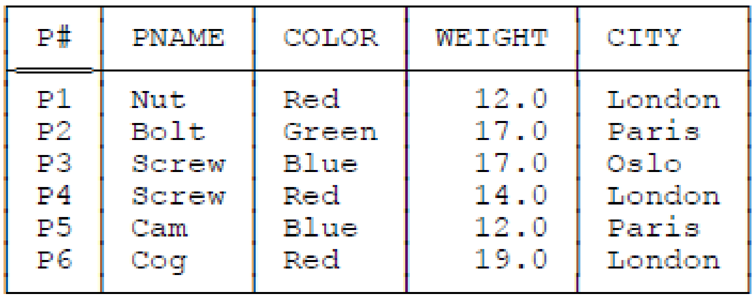
8 Condensed Columns

8.1 Introduction

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

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



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

Fig. 8.1: The parts relation P

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

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

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

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

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

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

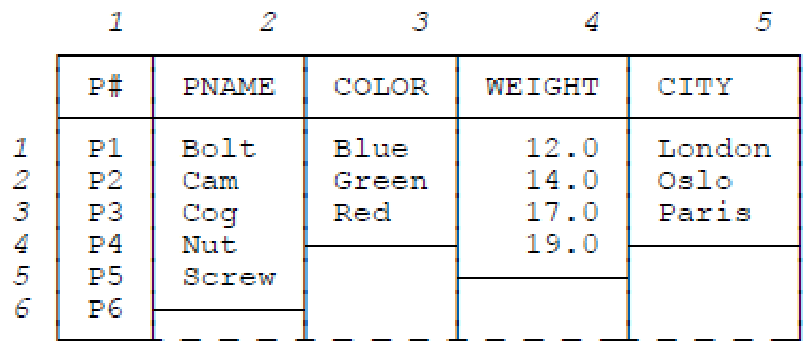
Fig. 8.3: Field Values Table corresponding to the file of Fig. 8.2

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

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

8.2 Condensing the Field Values Table

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



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

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

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

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

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

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

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

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

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

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

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

  a) This trick implies the need for an additional user-level relation to represent the mapping;
  b) It also implies that user-level requests are more complicated, because they require additional joins.

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

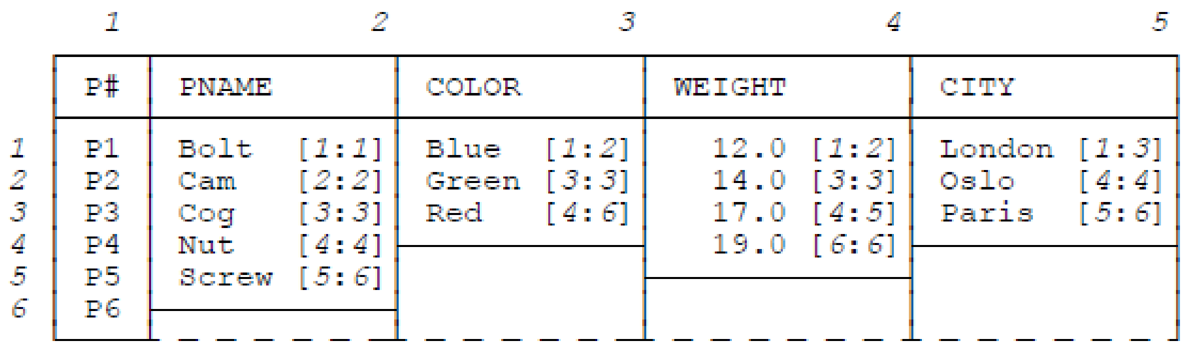
- For completeness, I should note that a column doesn't actually have to be sorted in order to be condensed—the benefits that follow from eliminating redundancy would apply even without sorting. But sorting provides so many additional benefits that it's reasonable to assume that any column that's condensed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

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

Row Ranges

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

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



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

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

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

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

I joined MITAS because
I wanted **real responsibility**
The Graduate Programme
for Engineers and Geoscientists
www.discovermitas.com
Month 16
I was a construction supervisor in the North Sea advising and helping foremen solve problems
Real work
International opportunities
Three work placements
MAERSK

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

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

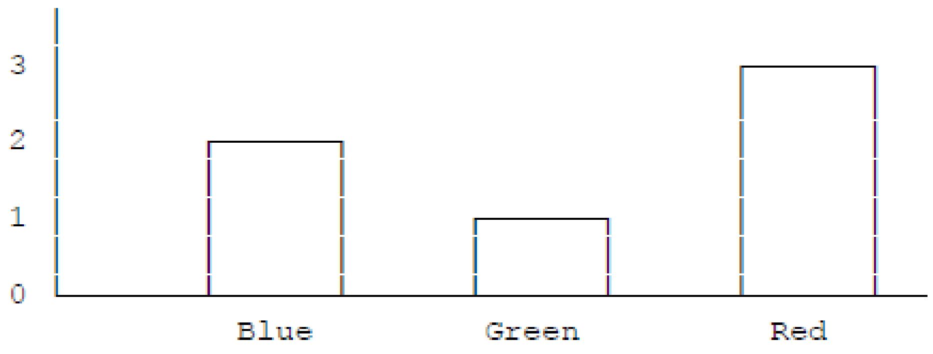


Fig. 8.7: Color histogram (based on Fig. 8.6)

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

8.3 Implications for Record Reconstruction

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

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

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

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

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

8.4 Expanding the Record Reconstruction Table

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.5 Further Space-Saving Techniques

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

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

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

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

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

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

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

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

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

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

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

  This technique is called *straight-line encoding.*

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

Endnotes

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