12 File Factoring

12.1 Introduction

We saw in Chapter 11 that it would be good to reduce the size of the Record Reconstruction Table. File factoring, or just *factoring* for short, is a technique for achieving this goal. (As mentioned in the previous chapter, it can have the effect of reducing the size of the Field Values Table as well; however, it's the effect on the Record Reconstruction Table that's the real point.) Here in outline is how it works:

- Starting with a given user-level relation, and hence a corresponding file, we decompose that file "vertically" into two or more **subfiles** (the official term is *factors,* but there's a good reason, which I'll explain in Section 12.3, for preferring the term *subfiles*). Each subfile is smaller than the original file, in the sense that it has fewer fields, and possibly fewer records, than the original file. *Note:* The term *vertical decomposition* refers to the fact that the decomposition is done "between fields," as it were.

- We then map each of those subfiles into its own Field Values Table and Record Reconstruction Table. Because the subfiles are smaller than the original file, those Field Values and Record Reconstruction Tables are smaller than their counterparts would have been for the original file. In particular, the Record Reconstruction Tables involve fewer pointers than their original counterpart would have done, a fact that can have dramatic effects on overall space requirements, as we'll see.

In effect, therefore, factoring replaces large tables by smaller ones, such that the total space required for the smaller ones is less—usually *much* less, in practice—than that required for the large ones. As I claimed in the previous chapter, it can thus be seen as a logical compression technique: logical, because the compression in question is performed (conceptually, at least) at the file level, before we even begin to think about the question of mapping the data to disk. The net effect is:

a) To make a larger portion of the data—in particular, a larger portion of the Record Reconstruction Table—permanently memory-resident, and
b) To pack more useful data into each page on the disk, thus providing "more bang for the buck" on each I/O operation.

The structure of this chapter is as follows. Following this introductory section, I'll explain the basic idea of factoring by means of a simple example in Section 12.2; then I'll elaborate on and generalize from that example in Sections 12.3 and 12.4. In Section 12.5, I'll explain what's involved in doing record reconstruction with factored files. Finally, in Section 12.6, I'll point out some additional benefits of factoring, over and above the overriding one of reducing Record Reconstruction Table space requirements.

12.2 A Simple Example

I have a problem. By definition, the techniques to be discussed in this chapter (also in the next two) are intended for dealing with very large data sets, with raw data space requirements measured in the billions of bytes or even more. (Actually the same was true throughout Part II of the book, but it's even more true here.) For obvious reasons, however, I can't show examples that involve such very large data sets. In what follows, therefore, you'll have to exercise your imagination a little; to be specific, you'll have to extrapolate from very small examples to the very large databases that actually exist in the real world.

I'll build on the parts example from Chapter 8 (see Fig. 8.1 in that chapter). Just to remind you, the relation P in that example originally had five attributes, as follows:

Part number:	`P#`
Part name:	`PNAME`
Color:	`COLOR`
Weight:	`WEIGHT`
Location:	`CITY`

Now let's extend it to include some additional ones—let's say as follows:

State:	`STATE`
Zip code:	`ZIP`
Phone number:	`PHONE#`

In practice there might well be other attributes too—for example, part description, street address, and so on—but the eight listed above are sufficient for our purposes.

Perhaps I should explain the semantics a little, in order to make the example a little more intuitively acceptable. Essentially, I'm taking the combination of CITY, STATE, and ZIP to be an elaboration of the old CITY attribute; I'm assuming that this combination of attributes identifies the location of the (sole) warehouse where parts of the indicated kind are kept. PHONE# gives the (sole) phone number for that warehouse. Also, I'll assume for simplicity that STATE always identifies a state in the U.S., and ZIP is thus always a U.S. zip code—and I'll stick to five-digit zip codes, again for simplicity. (By the way, did you know that *zip* is an acronym? It stands for *zoning improvement plan*.)

Let's assume further that there are ten million different parts, and hence ten million tuples in the parts relation and ten million records in the corresponding parts file. *Note:* I'll stick to this particular assumption throughout this chapter, and indeed throughout the next two as well.

Now, the basic parts Record Reconstruction Table is isomorphic to the parts file (that is, it has the same number of rows and columns as that file has records and fields, respectively). So that Record Reconstruction Table now has 80 million cells, and hence 80 million pointers. Each pointer in turn is 24 bits, and so the total space requirement is 240 megabytes (240MB). *Note:* If the Record Reconstruction Table were expanded to include direct pointers into the Field Values Table as well, the space requirement would double, to 480MB; for simplicity, for simplicity, however, let's omit these latter pointers. (Actually the space requirements wouldn't *exactly* double, because the Field Values Table would be condensed and the pointers into it would therefore be less than 24 bits. As I say, however, I'm going to ignore those pointers anyway.)

Assume now for the sake of the example that for any given zip code, there's just one city and state; that is, if *z* is a zip code and *c* and *s* are the corresponding city and state, then, whenever a tuple of the original parts relation P has ZIP = *z,* it also has CITY = *c* and STATE = *s*. In other words, there's a *many-to-one relationship* from ZIP to CITY and STATE: Many zip codes can have the same city and state, but no zip code can have more than one city and state (but see the next section). Formally, we say there's a **functional dependency** from ZIP to CITY and STATE, and we express it thus:

```
        { ZIP } → { CITY, STATE }
```

The general form is LHS → RHS; you can read it as "the right-hand side (RHS) is *functionally dependent* on the left-hand side (LHS)" or, more simply, just as "left-hand side arrow right-hand side." By convention, we enclose the left- and right-hand sides in braces because they're both *sets* of attribute names.

Now, it follows from the existence of the functional dependency from ZIP to CITY and STATE that the parts relation P contains a great deal of redundancy. After all, there are ten million distinct tuples, but there certainly aren't ten million distinct zip codes. In fact, I have it on good authority that there are around 38,000 of them (for the whole of the U.S., that is)—but to keep the arithmetic simple, let's round that figure up to 40,000. On average, then, there'll be 250 distinct tuples in the relation for any given zip code, and all 250 of those tuples will contain precisely the same values for ZIP, CITY, and STATE (there's the redundancy).

An obvious factoring thus suggests itself: Starting with the original parts file, let's decompose it vertically into two subfiles, with fields as indicated below:

```
Subfile 1    Subfile 2

P#           ZIP
PNAME        CITY
COLOR        STATE
WEIGHT
ZIP
PHONE#
```

For example, if the original file included a record looking like this—

P#	PNAME	COLOR	WEIGHT	CITY	STATE	ZIP	PHONE#
P8	Wheel	Black	15.0	Rome	GA	*zzz*	*nnnnnn*

(I've shown the ZIP and PHONE# values symbolically for simplicity)——then the two subfiles will include records looking like this:

Subfile 1

Subfile 1

P#	PNAME	COLOR	WEIGHT	ZIP	PHONE#
P8	Wheel	Black	15.0	*zzz*	*nnnnnn*

Subfile 2

ZIP	CITY	STATE
zzz	Rome	GA

There'll be one record in Subfile 1 for each part number (10 million records), and one record in Subfile 2 for each zip code (40,000 records). Of course, the original file can be reconstructed from the two subfiles by "joining" them back together on the ZIP field ("joining" in quotes because, strictly speaking, join is an operation that applies to relations, not to files).

So Subfile 1 has ten million records and six fields, while Subfile 2 has 40,000 records and three fields. Each subfile has its own Record Reconstruction Table. The first has 60 million pointers, still 24 bits each, for a total of 180MB. The second, however, has only 120,000 pointers, and those pointers are only 16 bits each, for a total of only 240KB (*kilobytes,* not megabytes); in fact, the space required for the second Record Reconstruction Table is negligible compared to that required for the first. The net effect is that we've reduced overall space requirements by around 25 percent.

The foregoing example illustrates the basic idea of factoring. Of course, there's still quite a lot more to be said, but first let me call out a few explicit points here:

- For simplicity I'll assume throughout this chapter that factoring always decomposes a given file into exactly two subfiles, as in this first example (barring explicit statements to the contrary, of course).

- I'll also assume that one of those subfiles is the "large" subfile and the other is the "small" subfile, and I'll refer to them as such (or sometimes as simply the large and small *files*, because of course a subfile can be regarded as a file in its own right—that's why I prefer the term *subfile* over the term *factor*). And I'll refer to the corresponding Field Values and Record Reconstruction Tables as "large" and "small" accordingly.

- *Very important:* The small Record Reconstruction Table will usually be *much* smaller than the large one—as indeed it was in our example—and can therefore be memory-resident. This is the basic object of the exercise, of course.

- *Also very important:* Factoring does **not** have to be done "by hand" (as it were). Rather, it's done automatically during the load process, on the basis of certain heuristics that are built into the load utility and various statistical analyses of the data that are also performed at load time. In other words, the benefits of factoring are obtained automatically, without any need for human decisions (on the part of the database administrator in particular).

Note finally that factoring as described above conceptually leads to two separate Field Values Tables, as well as two separate Record Reconstruction Tables. However, those two Field Values Tables can then be merged back into one as described in Chapter 9. In a sense, file factoring might be thought of as a kind of inverse of column merging as described in Chapter 9; column merging means two or more files map to one Field Values Table, loosely speaking, while file factoring means one file maps to two or more Field Values Tables (but those Field Values Tables are then merged back into one anyway, as we've just seen).

12.3 Elaborating on the Example

In the example in the previous section, we decomposed the original file on the basis of a certain functional dependency (FD for short). For that very reason, however, readers knowledgeable in database matters might have found the example a little unconvincing: Surely the database designer would already have performed the indicated decomposition at the relational level, precisely because of the existence of that FD? Indeed, such "decomposition at the relational level" is exactly what the business of *further normalization* is all about—see, for example, reference [32]. And if the designer had indeed already performed that decomposition at the relational level, then we would have started off with two distinct relations, and hence two distinct files at the file level, and the question of automatic decomposition of a single file into two distinct subfiles would never have arisen.

There are several possible responses to this objection, however. Four such are explained in the subsections immediately following.

"Denormalize for Performance"

The first point is that, in practice, the database designer might very well *not* have already performed the indicated decomposition at the relational level after all. The reason is that—at least in today's mainstream systems—full normalization is often contraindicated, because the direct-image nature of those systems can give rise to performance problems with fully normalized designs. The usual argument goes something like this [27]:

1. Full normalization means lots of logically separate relations at the relational level.
2. Lots of logically separate relations at the relational level means lots of physically separate stored files at the storage level.
3. Lots of physically separate stored files means lots of I/O.

For example, given our usual suppliers, parts, and shipments relations, a request to find London suppliers who supply red parts will involve two joins: First, join suppliers to shipments (say); second, join the result to parts. (I'm ignoring the two restriction operations for simplicity.) And if the three relations in fact do map to three physically separate stored files as suggested, then those two joins will indeed require lots of I/O and will therefore perform badly. Hence the cry we've doubtless all heard so many times: "Denormalize for performance!"

Note: Just in case you *haven't* heard this cry before, let me elaborate briefly. First, of course, normalization is a logical design discipline for reducing redundancy at the user or relational level. The trouble is, normalization leads to an increase in the number of relations and hence to an increase in the number of joins required in queries; and (to repeat) in a direct-image system, that increased number of joins translates directly into a performance hit. Denormalization is an attempt to fix this latter problem. (Note, however, that denormalization, unlike normalization, can hardly be described as a *discipline,* being in fact totally ad hoc.) Denormalization decreases the need for joins by decreasing the number of relations. Unfortunately, of course, it also increases the degree of redundancy—with negative consequences for updates, and even for some queries. In my opinion, applying a user-level fix (denormalization) to an internal-level problem (performance) cannot, by definition, be the best solution to that problem—but in direct-image systems, it might be the only solution available.

So the foregoing argument—the argument, that is, that the designer might not have already performed the decomposition at the relational level—is valid, more or less, in a direct-image system. However, it certainly isn't valid in a TR system; in a TR system, relations don't map directly to physical files, and joins are cheap. In a TR system, in fact, we really can, and should, go for fully normalized designs at the relational level (I'll come back to this point in Chapter 15). Thus, this first response to the objection that the example of the previous section wasn't very convincing is perhaps not a very strong one, given the TR context. So let's move on quickly to the second response ...

Normalization Is Based on **Relevant** *FDs*

There's a popular misconception in the IT community at large to the effect that logical database design requires normalization to be performed on the basis of *all* FDs. In fact, of course, such is not the case; rather (as I've written elsewhere [32]), normalization should be performed on the basis of all *relevant* FDs, not on the basis of all FDs that happen to exist. In the case of the parts relation, for example, with its attributes ZIP, CITY, and STATE (among others), the database designer might well decide that the FD

```
        { ZIP } → { CITY, STATE }
```

isn't very relevant to the problem at hand, and hence that decomposition at the relational level on the basis of that particular FD is hardly worthwhile. After all, CITY and STATE are almost invariably required together (think of printing a mailing list, for example); what's more, zip codes don't change very often, and thus there doesn't seem to be much to be gained by conventional normalization on the basis of that FD. Indeed, there might even be something to be lost; certainly conventional normalization will make some queries a bit more complex (from the user's point of view, that is), because they'll involve an additional join.

So we've arrived at the notion that the data might satisfy certain FDs that the database designer didn't use as a basis for normalization and (in all probability) didn't even declare to the DBMS. Conceivably, in fact, the data might satisfy certain FDs that the designer wasn't even aware of—but the load process can still detect such FDs and use them to perform file factoring. Thus, file factoring can apply even when normalization might have been applied in the first place but in fact wasn't, for some reason.

Factoring Based on Approximate FDs

The third response is this (and it's an important one): *File factoring can be based on "approximate FDs" as well as on genuine ones*. For example, I've been assuming up until this point that the FD

```
{ ZIP } → { CITY, STATE }
```

holds true, but in the real world it doesn't—not quite. Let me explain. Recall first that this FD effectively asserts that no zip code ever corresponds to more than one city and state combination, or in other words that distinct city and state combinations always have distinct zip codes. Well, there are exceptions; for example, the cities of Jenner and Fort Ross in California both have zip code 95450. (This kind of thing can happen if a zip code is assigned to some region and then part of that region subsequently incorporates and becomes a separate city in its own right.) Thus, a more accurate statement is that the "FD" (in quotes because it isn't really an FD at all)

```
{ ZIP } → { CITY, STATE }
```

almost holds true ... but that *almost* means we can't use the "FD" as a basis on which to perform normalization. For suppose our original parts relation looked like this:

P#	....	CITY	STATE	ZIP	....
P88	....	Jenner	CA	95450	....
P99	....	Fort Ross	CA	95450	....

Now suppose we decompose it into two projections as follows:[1]

P#	....	ZIP	....
P88	....	95450	....
P99	....	95450	....

ZIP	CITY	STATE
95450	Jenner	CA
95450	Fort Ross	CA

Now we have *ambiguity:* We can't tell which parts are kept in which city (note that if we join the two projections back together, we'll get four tuples, not two). In other words, the decomposition has lost information. (To be valid for normalization, of course, we do require decompositions to be "nonloss" [32].)

However, the fact that we can't do normalization in this example doesn't mean we can't do factoring. In fact, there are at least two ways to do it, and I'll sketch them both briefly here.

The first involves introducing an artificial identifier, ZCS# say, for each distinct zip / city / state combination. Thus, if the original parts file looked like this—

P#	....	CITY	STATE	ZIP	....
P88	....	Jenner	CA	95450	....
P99	....	Fort Ross	CA	95450	....

—we might replace it by the following two subfiles:

P#	....	ZCS#	....
P88	....	*zcs8*	....
P99	....	*zcs9*	....

ZCS#	ZIP	CITY	STATE
zcs8	95450	Jenner	CA
zcs9	95450	Fort Ross	CA

As you can see, the artificial identifier ZCS# plays a role analogous to that played by candidate and foreign keys in the relational model—it's a candidate key for the small subfile and a corresponding foreign key in the large one, loosely speaking. (I say "loosely speaking" because in fact such artificial identifiers *aren't* keys in the relational sense; relational keys apply by definition to relations at the user level, while the artificial identifiers apply to files at the file level. But the parallel is exact.)

So what does the foregoing trick do to our storage requirements? The large Record Reconstruction Table still has 60 million pointers of 24 bits each, for a total of 180MB. However, the small one has only 160,000 pointers of 16 bits each, for a total of only 320KB. (I'm making the reasonable assumption that the small subfile still has around 40,000 records, even though zip codes aren't unique. The point is, they're *almost* unique.) As in Section 12.2, therefore, the space required for the small Record Reconstruction Table is negligible, and the net effect is that we've reduced overall space requirements by around 25 percent.

Note: Values of the artificial identifier ZCS# could even be *direct pointers* into the small Record Reconstruction Table. Certainly they can be just 16 bits, like the pointers in that table. I'll have more to say about this possibility in Section 12.5.

The second approach to factoring using an "approximate FD" is to pretend the FD is genuine, moving the rare exceptions out into a special file of their own. Thus, the vast majority of records in the original parts file can be treated exactly as in Section 12.2. When a situation arises like that with zip code 95450 in our example above, we treat one of the pertinent zip / city / state combinations in the usual way, and move the others out into the special file. The result might look like this in our example:

Large subfile

P#	....	ZIP	....
P88	....	95450	....

Small subfile

ZIP	CITY	STATE
95450	Jenner	CA

Special-case subfile

P#	ZIP	CITY	STATE
P99	95450	Fort Ross	CA

Part numbers are unique in the large subfile; zip codes are unique in the small subfile; and part numbers (again) are unique in the special-case subfile. Moreover, no part number appears in both the large subfile and the special-case subfile. *Note:* It's true that access by part number is now more complicated than it was before, but at least the special-case Record Reconstruction Table will be small enough to be memory-resident, as we'll see in just a moment.

Let's do the storage arithmetic again. Suppose one tenth of one percent of the original ten million part records (in other words, 10,000 part records in total) have to be treated as special cases in the foregoing sense. Then the large Record Reconstruction Table still requires approximately 180MB. The small one still requires approximately 240KB. And the special-case one has 40,000 pointers of 14 bits each, for a total of 70KB. Once again, the large table is the significant one, and once again the net effect is that we've reduced overall space requirements by around 25 percent.

Incidentally, while I'm on the subject of factoring on the basis of approximate FDs, there's one pragmatically important special case to consider, as follows. Let *F2* be a field in some file, and let *F2* be "of low cardinality" (meaning it doesn't contain many distinct values compared to the total number of records in the file overall).[2] Then it's necessarily "almost" true that the FD

```
{ F1 } → { F2 }
```

holds true for all fields *F1* in that same file, and the factoring techniques described in the present subsection are thus directly applicable.

Following on from the previous point, let me now add that there's also one important special case of the general idea of a field being of low cardinality, and that's the case in which most of the values in the field are the same—for example, a numeric field in which most of the values are zero. Again, the factoring techniques described in this subsection are directly applicable.

Factoring Isn't Necessarily Based on FDs

The fourth and last response to the objection that the example of Section 12.2 makes little sense is simply this: While there are certainly parallels between factoring as described so far and conventional normalization—observe in particular that they both involve vertical decomposition—the truth is that factoring is, in a sense, more general than conventional normalization. In particular, factoring doesn't necessarily have to be based on functional dependencies as such[3] (nor even on "approximate" FDs). Rather, it can be based on *absolutely any kind of statistical pattern or "clumpiness"* in the data whatsoever. The section immediately following describes some of these further possibilities.

12.4 Further Possibilities

Suppose the parts file has already been factored as described in the previous section—in the subsection entitled "Factoring Based on Approximate FDs"—to yield subfiles that look like this:

Large subfile	*Small subfile*
P#	ZCS#
PNAME	ZIP
COLOR	CITY
WEIGHT	STATE
ZCS#	
PHONE#	

As we've seen, the Record Reconstruction Table for the large subfile still requires a fairly hefty 180MB. What can we do to reduce this space requirement still further?

Well, suppose now, not at all unrealistically, that there are very few distinct color / weight combinations in the large file (equivalently, in the original unfactored file). For the sake of the example, suppose there are just 500 such combinations, corresponding to (perhaps) ten different colors and 50 different weights. Then the combination of COLOR and WEIGHT behaves like a low-cardinality field, and we can deal with it accordingly. To be specific, we can introduce an artificial identifier, CW# say, for each distinct color / weight combination, and factor the large subfile above into two further subfiles that look like this:

```
Large subfile      Small subfile (second level)

P#                 CW#
PNAME              COLOR
CW#                WEIGHT
ZCS#
PHONE#
```

Note: Since the file we're factoring here is itself already a subfile, we might say we're performing *hierarchic* factoring in this particular example. The possibility of hierarchic factoring is, of course, a natural consequence of the fact that a subfile can be regarded as a file in its own right, and it's another reason why I prefer the term *subfile* to the term *factor*. The fact that a subfile is a file is important for exactly the same kinds of reasons that the fact that a subset is a set is important in mathematics.

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

Anyway, here's the storage arithmetic. The large Record Reconstruction Table now has only 50 million pointers instead of 60 million, reducing the space requirement for that table from 180MB to 150MB. The small one has just 4,500 pointers of just nine bits each, for a total of 4,950 bytes, which we can ignore completely. Once again the large table is the only important one, and now we've reduced overall space requirements by around 37.5 percent (37.5 percent of the original requirement of 240MB, that is, as calculated near the beginning of Section 12.2).

We can do better. Instead of factoring out the zip / city / state combination and the color / weight combination separately, why not factor them out *together*? The subfiles might look like this (note the artificial identifier ZCSCW#):

Large subfile	*Small subfile*
P#	ZCSCW#
PNAME	ZIP
ZCSCW#	CITY
PHONE#	STATE
	COLOR
	WEIGHT

For simplicity, let's abbreviate the expressions zip / city / state and color / weight to ZCS and CW, respectively. By our assumptions, then, there are approximately 40,000 distinct ZCS values and 500 distinct CW values; thus, there are at most 20 million distinct ZCS / CW combinations, and hence at most 20 million records in the small subfile. In practice, of course, it's impossible that all 20 million combinations actually exist (after all, there were only 10 million part records to start with); for the sake of the example, therefore, let's suppose that just one million combinations actually do exist. Now let's do the storage arithmetic again. The large Record Reconstruction Table now has only 40 million pointers, reducing the space requirement for that table still further to 120MB. The small one has six million pointers of 20 bits each, for a total of 15MB. The grand total is thus 135MB, a saving of around 43.75 percent over the original.

Yet another possibility is to break fields up into **subfields** (conceptually speaking) and then to treat those subfields as fully-fledged fields in their own right in the factoring process.[4] In our running example, an obvious candidate for such treatment is the PHONE# field. The original parts file has 10 million phone numbers, but it can't have 10 million area codes; in fact, let's assume, as we did in Chapter 11 (Section 11.4), that there are just 250 distinct area codes. What's more, there's a high correlation between area codes and zip codes; in fact, let's assume, reasonably enough, that most zip codes have just one area code (in other words, there's an "approximate FD" from zip codes to area codes). So it makes sense to split the PHONE# field into AREA_CODE and REST, and then to factor out the AREA_CODE along with the ZCS and CW information like this (note the artificial identifier ZCSCWAC#):

Large subfile	*Small subfile*
P#	ZCSCWAC#
PNAME	ZIP
ZCSCWAC#	CITY
REST	STATE
	COLOR
	WEIGHT
	AREA_CODE

Now, this example differs from previous ones in that it has no effect on the large Record Reconstruction Table; rather, its effect is on the large Field Values Table, whose space requirements are reduced by 30MB (ten million three-byte area codes). At the same time, it adds an AREA_CODE column to the small Field Values Table—but that column is condensed and requires only 750 bytes, which we can ignore. More important, it also adds a column of pointers to the small Record Reconstruction Table, for an additional 2.5MB. The net saving is thus 28.5MB. (I can't easily express this saving as a percentage, because now the Field Values Table is involved as well as the Record Reconstruction Table.)

By way of summary, then, the general principle that the foregoing examples illustrate (both in this section and in the previous one) is this:

- Let *F1, F2, ..., Fn* be distinct fields—possibly subfields—within some given file (where *n* is greater than one).
- Let the set of all of those fields be considered as a single combined field *F*.
- Let *F* have cardinality *c*.
- Then, whenever *c* is small compared to the total number of records in the file overall, it's worth factoring *F* out into a "small" subfile, leaving an identifier behind in the "large" subfile to serve as the necessary link to that small file. The identifier might be a user field or it might be an introduced artificial identifier, depending on circumstances.

Note in particular the requirement that *n* be greater than one. Clearly there's no point in factoring out just a single field, because if an identifier has to be left behind in the large file, then that large file will still have just as many fields as it had before.[5] To say it again, the major object of the overall factoring exercise is to reduce the size of the large Record Reconstruction Table, and the way to do that is to reduce the number of fields in the large file.

One last point to close this section: You might possibly be feeling there are some similarities between the notion of file factoring as described in this chapter and the notion of combined columns as discussed in Chapter 9 (Section 9.4)—and indeed there are some similarities. The primary objective in both cases is certainly to save space (and as a matter of fact, the savings obtained are comparable in the two cases). But there are some differences too, of course. In a nutshell:

- Combining columns is a technique for mapping two or more fields to the same column in the Field Values Table, where the fields in question are, in general, of different types (we aren't talking about *merged* columns here). This technique saves space in the Record Reconstruction Table by reducing the number of columns in the Field Values Table. However, it does make it more difficult to search on the basis of columns other than the leading one in any such column combination.

- By contrast, a large part of the point regarding file factoring is precisely that distinct columns in the Field Values Table *aren't* combined into one. Thus, searches on the basis of individual columns are still straightforward. (In fact, as we've seen, we might even want to split one column in the Field Values Table into two or more columns—or, more precisely, we might want to have two or more columns in the Field Values Table for one field in the user-level file.)

12.5 Record Reconstruction

What are the implications of factoring for the record reconstruction process? In order to consider this question, I think it's best to return to a much simpler example. Let's go all the way back to the original parts example from Chapter 8. Fig. 12.1, a copy of Fig. 8.2, shows a file corresponding to the parts relation of Fig. 8.1. *Note*: Of course, this example is really much too simple to illustrate the need for factoring or any of the potential benefits described elsewhere in this chapter. So I'll just have to ask you to suspend disbelief and work through the example with me anyway.

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

Now let's assume we want to factor out the color / city combination. As explained in Section 12.3, we'll have to introduce an artificial identifier, CC# say, to link the resulting subfiles together, logically speaking. Thus, Figs. 12.2, 12.3, and 12.4 show, respectively, the subfiles that result from this factoring, the corresponding Field Values Tables, and the corresponding Record Reconstruction Tables. *Note:* With respect to the Field Values Tables in Fig. 12.3, I think it's clearer not to merge them together, though in practice they probably would be so merged; with respect to the Record Reconstruction Tables in Fig. 12.4, I think it's clearer to omit the direct pointers into the Field Values Tables that, again, they might contain in practice (or not, as the case may be—recall that those pointers are basically just an optional extra anyway).

	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*

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

Fig. 12.2: Subfiles after factoring out COLOR and CITY

	1	2	3	4
	P#	PNAME	WEIGHT	CC#
1	P1	Bolt [1:1]	12.0 [1:2]	*cc1* [1:3]
2	P2	Cam [2:2]	14.0 [3:3]	*cc2* [4:4]
3	P3	Cog [3:3]	17.0 [4:5]	*cc3* [5:5]
4	P4	Nut [4:4]	19.0 [6:6]	*cc4* [6:6]
5	P5	Screw [5:6]		
6	P6			

	1	2	3
	CC#	COLOR	CITY
1	*cc1*	Blue [1:2]	London [1:1]
2	*cc2*	Green [3:3]	Oslo [2:2]
3	*cc3*	Red [4:4]	Paris [3:4]
4	*cc4*		

Fig. 12.3: Field Values Tales for the subfiles of Fig. 12.2

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

	1	2	3
	CC#	COLOR	CITY
1	4	2	1
2	3	4	3
3	1	3	2
4	2	1	4

Fig. 12.4: Record Reconstruction Tables for the subfiles of Fig. 12.2

Now let's consider the problem of doing record reconstruction for, say, part records for parts in Oslo (I deliberately choose an example in which there's just one qualifying record, for simplicity). Noting that the CITY field is in the small subfile, we see that the sequence of events must be as follows:

- Search column CITY of the small Field Values Table in Fig. 12.3 for the specified value Oslo. We discover that the sole record we want passes through cell [*2*,*3*] of the small Record Reconstruction Table (row *2* because the row range for Oslo is [*2*:*2*], and column *3* because column CITY is indeed the third column of the small tables).

- Follow the zigzag passing through cell [*2*,*3*] of the small Record Reconstruction Table in Fig. 12.4. That zigzag looks like this:

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

  The corresponding field values are:

  ```
  Oslo, cc3, Blue
  ```

- Now search column CC# of the large Field Values Table in Fig. 12.3 for the value *cc3* (in effect, we're using the "candidate key" value in the small Field Values Table to find the rows containing a corresponding "foreign key" value in the large Field Values Table). We discover that the sole record we want passes through cell [*5*,*4*] of the large Record Reconstruction Table.

- Follow the zigzag passing through cell [*5*,*4*] of the large Record Reconstruction Table in Fig. 12.4. That zigzag looks like this:

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

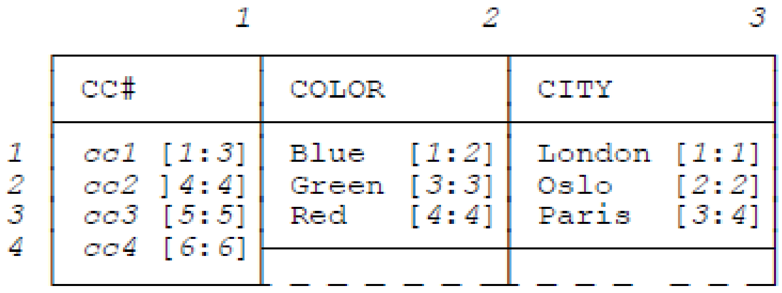
  The corresponding field values are:

  ```
  cc3, P3, Screw, 17.0
  ```

Reconstruction of the desired record is now complete. However, note the fact that we've had to traverse two separate zigzags, "hooking them together" (so to speak) by means of the artificial identifier CC#. Without going into details, it should be clear that we'd have to follow a similar procedure in order to reconstruct, say, part records for parts with weight 19.0, except that this time we'd have to use the "foreign key" value in the large Field Values Table to look up the corresponding "candidate key" value in the small Field Values Table, instead of the other way around (because WEIGHT is a field in the large subfile, not the small one).

One implication of the foregoing is, of course, that factoring can lead to some inefficiencies in the record reconstruction process. However, matters are not as bad as they might seem. As I pointed out in Section 12.3 (in the subsection "Factoring Based on Approximate FDs"), artificial identifier values—CC# values in the example—can be **pointers**. If they are, then instead of the associative lookup we had to do in the "WEIGHT = 19.0" example from the large Field Values Table to the small one, we can now *follow a pointer* directly from that large table to the small *Record Reconstruction* Table (bypassing the small Field Values Table entirely). Reconstruction "from the large to the small" will thus be more efficient.

What about the other direction (reconstructing "from the small to the large," as in the "CITY = Oslo" example)? Well, we can make this process more efficient too if we want, by carrying some additional row ranges in the small Field Values Table (in the CC# column of that table, to be specific). Fig. 12.5 shows what happens to the small Field Values Table in the example if we adopt this approach.



	1	2	3
	CC#	COLOR	CITY
1	*cc1* [*1*:*3*]	Blue [*1*:*2*]	London [*1*:*1*]
2	*cc2*]*4*:*4*]	Green [*3*:*3*]	Oslo [*2*:*2*]
3	*cc3* [*5*:*5*]	Red [*4*:*4*]	Paris [*3*:*4*]
4	*cc4* [*6*:*6*]		

Fig. 12.5: Small Field Values Tables with CC# row ranges for the large Record Reconstruction Table

By way of example, consider cell [*1*,*1*] of the table in Fig. 12.5. That cell includes the row range [*1*:*3*]. That row range in turn shows that the rows in the large Record Reconstruction Table that correspond to CC# value *cc1* are rows *1*, *2*, and *3*. Thus, instead of the associative lookup we previously had to do from the small Field Values Table to the large one, we can now follow pointers directly from that small table to the large Record Reconstruction Table (bypassing the large Field Values Table entirely). Reconstruction "from the small to the large" will thus also be more efficient than it was before.

12.6 Additional Benefits

The foregoing sections should be sufficient to give you some idea of the possibilities inherent in file factoring. For real databases, where relations often start out at the user level with many more than just eight attributes—and sometimes with many more than ten million tuples—the savings achievable are likely to be much greater than those shown in the examples (certainly more than the comparatively miserly percentages we saw in those examples). As I've said, factoring can be based on any kind of "statistical clumpiness" in the data whatsoever. And the fact is, the vast majority of data in the real world exhibits such "clumpiness" in great abundance; thus, the vast majority of data is a candidate for treatment by means of the techniques described in this chapter.[6]

In this final section, I'd like to point out that file factoring has certain additional benefits as well, over and above its primary one of reducing space requirements. To be specific, it offers certain benefits in connection with (a) aggregate queries and (b) update operations. Those benefits are the subject of the next two subsections. First, though, it's only fair to mention one potential drawback too: namely, that replacing one large Record Reconstruction Table by two or more smaller ones means we can't have a "preferred" Record Reconstruction Table (as described in Chapter 7) that provides major-to-minor orderings over all of the fields of the original file. However, the Record Reconstruction Tables for the subfiles—for the "small" subfile in particular—can still be "preferred" (as they are in Fig. 12.4, in fact), and in practice that's likely to be sufficient. Why? Because the fields of the small subfile are likely to be the ones over which orderings will most often be requested, as we'll see in the subsection immediately following.

Aggregate Queries

Aggregate queries—see the discussion of the relational operator SUMMARIZE in Chapter 10, Section 10.5—naturally tend to be framed in terms of fields in the small file, precisely because those fields are the low-cardinality ones. In SQL terms, for example, the following ("Sum weights by city") is certainly a realistic query on the parts relation P—

```
SELECT DISTINCT P.CITY, SUM ( P.WEIGHT ) AS SUMWT
FROM P
GROUP BY P.CITY ;
```

—whereas the following ("Sum weights by name") probably isn't:

```
SELECT DISTINCT P.PNAME, SUM ( P.WEIGHT ) AS SUMWT
FROM P
GROUP BY P.PNAME ;
```

(I'm relying here on the fact that part names are "almost unique.")

Let's assume once again that we're dealing with the original parts relation P with its attributes P#, PNAME, COLOR, WEIGHT, and CITY (only). Let's assume too (as in the previous section) that the file corresponding to that relation P has been factored as follows:

Large subfile	*Small subfile*
P#	CC#
PNAME	COLOR
WEIGHT	CITY
CC#	

If the parts file is as originally shown in Fig. 12.1, then here are the actual values of the two subfiles (repeated from Fig. 12.2):

	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*

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

Observe now that if we partition the original file by COLOR and CITY, then each row in the small file corresponds to just one partition in the result. So it's not at all unreasonable to **precompute** certain aggregates for those partitions—compute them at load time, that is—and keep the results with the rows in the small file.[7] For example, if we were to treat *sums of weights* in this fashion, the small file might look like this:

	1	2	3	4
	CC#	COLOR	CITY	SW
1	*cc1*	Red	London	50.0
2	*cc2*	Green	Paris	17.0
3	*cc3*	Blue	Oslo	17.0
4	*cc4*	Blue	Paris	12.0

(I've shown the computed values as an extra field of the file, called SW. Of course, the computations aren't all that interesting in this particular example, but you get the general idea.)

Now suppose the user issues the SQL query from the start of this subsection ("Sum weights by city"):

```
SELECT DISTINCT P.CITY, SUM ( P.WEIGHT ) AS SUMWT
FROM P
GROUP BY P.CITY ;
```

At run time, then, instead of having to partition the large original file by CITY and do a series of possibly lengthy summations, the system can simply partition the *small* file by CITY and do a series of much shorter summations instead. *Note:* The savings can be particularly dramatic if there's a HAVING clause to eliminate certain of the partitions before the summations are done.

Analogous remarks apply to many other aggregate operators as well.[8] Note in particular that the technique could help in the case where the partitioning is done on the basis of a *subfield*. The parts example doesn't illustrate this point, but "Count subscribers in the 415 area code" might be an example of a query that could benefit from treatment similar to that described above.

Finally, ORDER BY requests too naturally tend to be framed in terms of fields in the small file, again because those fields are the low-cardinality ones. In SQL terms, for example, the following is certainly a realistic query on the parts relation P—

```
SELECT ...
FROM P
ORDER BY CITY, COLOR ;
```

—whereas the following probably isn't:

```
SELECT ...
FROM P
ORDER BY CITY, PNAME ;
```

The relevance of factoring here should be obvious; in fact, I explained it earlier, when I said that the small Record Reconstruction Table, at least, could still be a "preferred" one.

Update Operations

The point here is essentially a simple one: When a new record is inserted, it's likely that values in low-cardinality fields within that record will already exist in the small file, precisely because those fields *are* low-cardinality. For example, let *p* be a new part record. Then it's quite likely that *p* will involve a color and a city—and even a color / city combination, and hence, implicitly, a CC# value too—that already exists in the small file. (By contrast, of course, *p* will definitely involve a new part number, and possibly a new name and/or weight as well.) In general, in other words, insert operations will typically "touch" the large file only. Analogous remarks apply to delete operations also.

Endnotes

1. Note that the normalization process is basically a process of taking projections; in other words, the decomposition operator is projection (the *re*composition operator, by contrast, is join).
2. Recall from Chapter 10 that the cardinality of a set is the number of elements in that set. Thus, when we say some field is of such and such cardinality, what we mean is that the set of values in that field is of that cardinality; in other words, we're referring to the number of distinct values in that field, not the number counting duplicates if any.
3. In the interests of accuracy, I should point out that conventional normalization isn't entirely based on functional dependencies either. FDs take us only as far as *Boyce/Codd normal form* (BCNF). *Fourth normal form* (4NF) relies on a generalization of functional dependencies called *multivalued dependencies* (MVDs). Likewise, *fifth normal form* (5NF) relies in turn on a generalization of MVDs called *join dependencies*. And I've recently been involved myself in the definition of a new *sixth normal form* (6NF), which relies on a generalization of JDs (see reference [42]).
4. Factoring on the basis of subfields is not the same as subfield encoding (see the previous chapter), though the concepts are related. By the way, you might have noticed that the first two examples in this section, though advertised as "further possibilities," were really, like the examples in the previous section, based on the idea of approximate FDs—and the same is at least arguably true of the subfield example I'm about to consider. Factoring techniques do exist that genuinely aren't based in any way on approximate FDs, but further details are beyond the scope of this book.
5. Note the implication that not all FDs, genuine or approximate, are useful for factoring. To be specific, the FD (or "FD" in quotes, possibly) LHS → RHS is useful for factoring only if the right-hand side involves at least two attributes.
6. An interesting idea for consideration is the following: Instead of starting with exactly the relations specified by the database designer, it might be nice to begin by denormalizing the database entirely, joining all of the relations together—conceptually, at any rate—into what's sometimes called a "universal relation," and then to go on and use file factoring repeatedly (hierarchically, in fact) to achieve a good disk representation. In this way, attributes that started out in different user-level relations might even find themselves mapped to fields in the same file (or subfile, rather) internally.
7. Conceptually, that is. In practice, the results will be kept in the small Field Values Table (after all, the files per se are just abstractions and don't physically exist).
8. In particular, the row ranges in column CC# in Fig. 12.5 provide analogous support for the COUNT operator.