7 Major-to-Minor Orderings

7.1 Introduction

The core concepts of the TR model are the Field Values Table and the Record Reconstruction Table, and I've now completed my description of those concepts, at least in their simplest form. As I explained in Chapter 4, however, the TR model also includes many "optional extras" or "frills" (some of which are so important that they'll almost certainly be included in any real implementation), and the time has come to start taking a look at some of those optional extras.

The present chapter is all about a "frill"—*refinement* is really a better word—that applies to the Record Reconstruction Table specifically. (By contrast, the refinements to be discussed in Chapters 8 and 9 apply to the Field Values Table, at least primarily, though in both cases there are implications for the Record Reconstruction Table as well.) Anyway, the refinement I want to discuss right now has to do with **major-to-minor ordering,** by which I mean, in SQL terms, the kind of ordering that results from a query that includes an ORDER BY specification of the form

```
ORDER BY A, B, C, ..., Z
```

As I'm sure you know, the tuples in the result of such a query are ordered, first, by ascending *A* value; then, for any given *A* value, by ascending *B* value; then, for any given *A-B* value combination, by ascending *C* value; and so on, finishing up with, for any given *A-B-C-*... value combination, by ascending *Z* value. The sequence of attribute names *A*, then *B*, then *C*, ..., then *Z*, is said to specify a *major-to-minor ordering* (where *A* is the major attribute, *B* is the next, *C* is the next, ..., and *Z* is the minor attribute).

Note: For each attribute mentioned within a given ORDER BY specification, SQL also lets us specify either ASC or DESC, where ASC means ascending sequence and DESC means descending sequence, and ASC is the default. I showed an example using DESC in Section 4.4 in Chapter 4. In what follows, I'll assume for simplicity that we always want ascending sequence specifically, barring explicit statements to the contrary.

7.2 The Suppliers-Parts-Projects Example

The suppliers relation S has served us well as a basis for examples ever since Chapter 2; however, it isn't really adequate to illustrate the points I want to make in the present chapter. Consider instead, therefore, the shipments relation SPJ depicted in Fig. 7.1.[1] That relation is meant to be interpreted as follows: The indicated supplier (S#) is supplying, or *shipping,* the indicated part (P#) to the indicated project (J#) in the indicated quantity (QTY). The attribute combination {S#,P#,J#} is a key (that is, no two tuples appearing in the relation at the same time ever have the same value for that combination of attributes); in fact, that combination is the *only* key. For definiteness, let's assume that attributes S#, P#, J#, and QTY are defined on types S#, P#, J#, and INTEGER, respectively (where INTEGER is a system-defined type and the other three are user-defined types).

S#	P#	J#	QTY
S1	P1	J1	200
S1	P3	J2	100
S2	P1	J1	200
S2	P1	J2	500
S2	P2	J2	500
S3	P1	J1	100
S3	P2	J2	500
S3	P3	J1	200
S3	P3	J2	200

Fig.7.1: The shipments relation SPJ

Note that I've deliberately chosen sample values for relation SPJ such that:

- No single attribute *A* has the property that every tuple has a value for *A* that's different from the value of *A* in all other tuples in the relation.
- Likewise, no attribute pair *A-B* has the property that every tuple has a value for *A-B* that's different from the value of *A-B* in all other tuples in the relation.
- Likewise, no attribute triple *A-B-C* has the property that every tuple has a value for *A-B-C* that's different from the value of *A-B-C* in all other tuples in the relation—except, of course, for the attribute triple {S#,P#,J#}, which (as we already know) constitutes a key and therefore must have a unique value in every tuple, by definition.

Incidentally, the main reason why the suppliers relation is inadequate for the purposes of the present chapter is that it has a single-attribute key and thus necessarily does have a single attribute that "happens" to have a unique value in every tuple. (In fact, it also has another single attribute, SNAME, that happens to have a unique value in every tuple, but this latter fact truly is a matter of happenstance—that is, {SNAME} isn't a key.) Of course, it follows from the fact that it has a single-attribute key—also from the fact that supplier names happen to be unique—that the suppliers relation also has several attribute pairs and several attribute triples that also have unique values in every tuple, a fortiori.

Fig. 7.2 shows a possible file corresponding to the relation of Fig. 7.1 (for simplicity, I've shown the records and fields of that file in the orderings suggested by Fig. 7.1), and Fig. 7.3 shows the corresponding Field Values Table.

	1	2	3	4
	S#	P#	J#	QTY
1	S1	P1	J1	200
2	S1	P3	J2	100
3	S2	P1	J1	200
4	S2	P1	J2	500
5	S2	P2	J2	500
6	S3	P1	J1	100
7	S3	P2	J2	500
8	S3	P3	J1	200
9	S3	P3	J2	200

Fig.7.2: File corresponding to the shipments relation of Fig. 7.1

	1	2	3	4
	S#	P#	J#	QTY
1	S1	P1	J1	100
2	S1	P1	J1	100
3	S2	P1	J1	200
4	S2	P1	J1	200
5	S2	P2	J2	200
6	S3	P2	J2	200
7	S3	P3	J2	500
8	S3	P3	J2	500
9	S3	P3	J2	500

Fig.7.3: Field Values Table corresponding to the file of Fig. 7.2

7.3 A Preferred Record Reconstruction Table

Recall now from Chapter 4 that the Record Reconstruction Table corresponding to a given file (and corresponding Field Values Table) is, in general, not unique. *We can turn this fact to our advantage*. It turns out that, given a particular file (and Field Values Table), certain Record Reconstruction Tables are "more equal than others," in the sense that they have certain very desirable properties. In this section, I'll show an example of what such a "preferred" Record Reconstruction Table might look like, and I'll explain what some of those desirable properties are. In the next section, I'll show how such preferred Record Reconstruction Tables can be built in the first place.

The particular preferred Record Reconstruction Table I want to discuss first is shown in Fig. 7.4.

	1	2	3	4
	S#	P#	J#	QTY
1	2	1	2	2
2	8	2	3	6
3	3	3	4	1
4	4	7	5	3
5	5	8	1	8
6	1	9	6	9
7	6	4	7	4
8	7	5	8	5
9	9	6	9	7

Fig.7.4: A preferred Record Reconstruction Table for the file of Fig. 7.2

Let's just do a spot check on the table in Fig. 7.4 to make sure it does indeed reconstruct at least one record correctly. Starting arbitrarily at cell [*4,3*], we have:

- Cell [*4,3*] of the Field Values Table contains the project number J1; cell [*4,3*] of the Record Reconstruction Table contains *5,* so—remembering that that *5* means *row* number *5* and the next *column* is column number *4*—we go next to cell [*5,4*].

- Cell [*5,4*] of the Field Values Table contains the quantity 200; cell [*5,4*] of the Record Reconstruction Table contains *8,* so now we go to cell [*8,1*] (the next or "fifth" column in fact wrapping around to the first).

- Cell [*8,1*] of the Field Values Table contains the supplier number S3; cell [*8,1*] of the Record Reconstruction Table contains *7,* so we go to cell [*7,2*].

- Cell [*7,2*] of the Field Values Table contains the part number P3; cell [*7,2*] of the Record Reconstruction Table contains *4,* and so we're back where we started, having reconstructed the shipment record:

	S#	P#	J#	QTY
8	S3	P3	J1	200

(More precisely, we've reconstructed a version of this record in which the left-to-right field sequence is J#, then QTY, then S#, then P#.)

Anyway, at least we do now have some empirical evidence tending to confirm that the table shown in Fig. 7.4 is indeed a valid Record Reconstruction Table for the file of Fig. 7.2. But what's special about it? Why is it "preferred"?

Well, suppose we try reconstructing the entire file, starting at cell [*1,1*] of each of the two tables for the first record in that reconstruction and then continuing down column *1* (the S# column)—that is, starting successive record reconstructions with cells [*2,1*], [*3,1*], ..., [*9,1*] for the second, third, ..., ninth record in the overall file reconstruction process. Suppose we then do the same thing again, but this time going down column *2* (the P# column) instead; and then again, going down column *3* (the J# column); and one final time, going down column *4* (the QTY column). What happens?

Unfortunately, I'm afraid you're going to have to do some work here—it's no good my just presenting you with the results, you really need to work through the process and determine those results for yourself (this is **Exercise 8**). When you do, however, you'll find that the results are in fact as shown in Fig. 7.5 (I've labeled them a., b., c., and d. for purposes of future reference).

	1	2	3	4
a.	S#	P#	J#	QTY
1	S1	P1	J1	200
2	S1	P3	J2	100
3	S2	P1	J1	200
4	S2	P1	J2	500
5	S2	P2	J2	500
6	S3	P1	J1	100
7	S3	P2	J2	500
8	S3	P3	J1	200
9	S3	P3	J2	200

	1	2	3	4
b.	S#	P#	J#	QTY
1	S3	P1	J1	100
2	S1	P1	J1	200
3	S2	P1	J1	200
4	S2	P1	J2	500
5	S2	P2	J2	500
6	S3	P2	J2	500
7	S3	P3	J1	200
8	S1	P3	J2	100
9	S3	P3	J2	200

	1	2	3	4
c.	S#	P#	J#	QTY
1	S3	P1	J1	100
2	S1	P1	J1	200
3	S2	P1	J1	200
4	S3	P3	J1	200
5	S1	P3	J2	100
6	S3	P3	J2	200
7	S2	P1	J2	500
8	S2	P2	J2	500
9	S3	P2	J2	500

	1	2	3	4
d.	S#	P#	J#	QTY
1	S1	P3	J2	100
2	S3	P1	J1	100
3	S1	P1	J1	200
4	S2	P1	J1	200
5	S3	P3	J1	200
6	S3	P3	J2	200
7	S2	P1	J2	500
8	S2	P2	J2	500
9	S3	P2	J2	500

Fig.7.5: Reconstructed files corresponding to the preferred Record Reconstruction Table of Fig. 7.4

Of course, Fig. 7.5 effectively shows four different versions of the original shipments file of Fig. 7.2. What I want to draw your attention to here, though, is just which versions they are. To be specific, note that the four versions represent, respectively, the results of the following four SQL queries (I've deliberately shown the attribute names in the various ORDER BY specifications in **bold**):

```
a. SELECT S#,P#,J#,QTY
   FROM SPJ
   ORDER BY S#,P#,J#,QTY;

b. SELECT S#,P#,J#,QTY
   FROM SPJ
   ORDER BY P#, J#, QTY, S# ;

c. SELECT S#, P#, J#, QTY
   FROM SPJ
   ORDER BY J#, QTY, S#, P#;

d. SELECT S#,P#,J#,QTY
   FROM SPJ
   ORDER BY QTY, S#, P#, J#;
```

In other words, the "preferred" Record Reconstruction Table of Fig. 7.4 doesn't just reflect four single-attribute orderings simultaneously—it actually reflects four major-to-minor orderings simultaneously. That is, column P# (for example) corresponds not just to a simple ORDER BY of the form ORDER BY P#, but rather to an ORDER BY of the form ORDER BY P#, J#, QTY, S#—and similarly for the other three columns.

Observe next that, since column P# of the preferred Record Reconstruction Table does reflect the major-to-minor ordering on P#-J#-QTY-S#, it also reflects, a fortiori, the major-to-minor ordering on P#-J#-QTY, the major-to-minor ordering on P#-J#, and the "major-to-minor" ordering on P# as well. Again, analogous remarks apply to the other three columns. *Note:* In the particular case of column S#, however, I should point out that there's no real difference between the specifications ORDER BY S#, P#, J#, QTY and ORDER BY S#, P#, J# (that is, the QTY specification is irrelevant in this particular example), because the combination {S#,P#,J#} is a key.

What's more, if we process our preferred Record Reconstruction Table by starting with (say) the S# column but processing it in reverse order (that is, from bottom to top), then it should be clear that we will obtain a result identical to part a. of Fig. 7.5, except that the rows will be in reverse sequence. In other words, we will have implemented the SQL query

```
SELECT S#, P#, J#, QTY
FROM SPJ
ORDER BY S# DESC, P# DESC, J# DESC, QTY DESC ;
```

So the preferred Record Reconstruction Table actually reflects an equal number of reverse major-to-minor orderings, too.

Note finally that we effectively get all of this extra functionality "for free"[2]—we have to have some Record Reconstruction Table, and it might just as well be a preferred one. And the foregoing discussion should be sufficient to explain why we regard such a Record Reconstruction Table as "preferred" in the first place.

7.4 Building a Preferred Record Reconstruction Table

In any given Record Reconstruction Table, each column effectively reflects a certain sort order that's associated with that column. Let me immediately explain this remark:

- First, the sort order "associated with" a given column is, in general, a major-to-minor ordering in which the corresponding attribute of the user relation is the major attribute.
- Second, when I say a given column "reflects" some particular sort order, I mean that if we process the Record Reconstruction Table in sequential order according to that column, then we will reconstruct the corresponding file in that specific sort order.

To obtain a preferred Record Reconstruction Table, therefore, it's necessary to associate the particular sort order we want with each individual column. How? Well, let's do it; let's build the specific Record Reconstruction Table used in the previous section. I'll use the technique described in Chapter 5, Section 5.4 (the one involving the Inverse Permutation Table).

We start by considering, for each field in turn of the shipments file, the sort order we do in fact want. For the S# field, that sort order is the one produced by the ORDER BY specification

```
ORDER BY S#, P#, J#, QTY
```

Take another look at the file in Fig. 7.2. It turns out, as it happens, that the file has been shown in that figure in exactly this sort order, so the "S# permutation" we want—see Chapter 4, Section 4.5, or Chapter 5, Section 5.4, if you need to refresh your memory regarding this notion—is as follows:

- S# - P# - J# - QTY : *1, 2, 3, 4, 5, 6, 7, 8, 9*

The P# permutation is the permutation that corresponds to the ORDER BY specification ORDER BY P#, J#, QTY, S#:

- P# - J# - QTY - S# : *6, 1, 3, 4, 5, 7, 8, 2, 9*

(you can easily check this by comparing parts a. and b. of Fig. 7.5). And the other two permutations are:

- J# - QTY - S# - P# : *6, 1, 3, 8, 2, 9, 4, 5, 7*
- QTY - S# - P# - J# : *2, 6, 1, 3, 8, 9, 4, 5, 7*

Next, we figure out the corresponding *inverse* permutations:

- S# - P# - J# - QTY : *1, 2, 3, 4, 5, 6, 7, 8, 9*

  Inverse : *1, 2, 3, 4, 5, 6, 7, 8, 9*
- P# - J# - QTY - S# : *6, 1, 3, 4, 5, 7, 8, 2, 9*

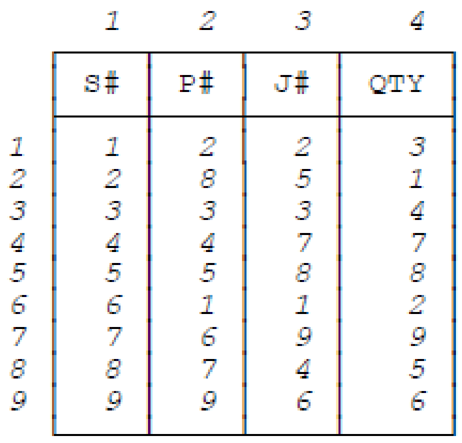
  Inverse : *2, 8, 3, 4, 5, 1, 6, 7, 9*
- J# - QTY - S# - P# : *6, 1, 3, 8, 2, 9, 4, 5, 7*

  Inverse : *2, 5, 3, 7, 8, 1, 9, 4, 6*
- QTY - S# - P# - J# : *2, 6, 1, 3, 8, 9, 4, 5, 7*

  Inverse : *3, 1, 4, 7, 8, 2, 9, 5, 6*

Incidentally, note that the S# permutation is its own inverse.

Here then is the Inverse Permutation Table:



	1	2	3	4
	S#	P#	J#	QTY
1	1	2	2	3
2	2	8	5	1
3	3	3	3	4
4	4	4	7	7
5	5	5	8	8
6	6	1	1	2
7	7	6	9	9
8	8	7	4	5
9	9	9	6	6

To build the corresponding Record Reconstruction Table, we use the algorithm from Chapter 5 (Section 5.4):

> Go to cell [*i,1*] of the Inverse Permutation Table. Let that cell contain the value *r;* also, let the next cell to the right, cell [*i,2*], contain the value *r′*. Go to the *r*th row of the Record Reconstruction Table and place the value *r′* in cell [*r,1*].

Executing this algorithm for *i = 1, 2, ..., 9* yields the entire S# column. The other columns are built analogously. The result is the "preferred" Record Reconstruction Table shown in Fig. 7.4. **Exercise 9:** Check this claim.

7.5 Another Example

In my discussions so far, I've considered only major-to-minor orderings that correspond to the left-to-right sequence of fields in the shipments file as shown in Fig. 7.2, together with cyclic shifts of that sequence, such as J#-QTY-S#-P#. Don't assume that such always has to be the case, however; the only hard requirement is that the sort order associated with a given column must be one for which the corresponding attribute of the user relation serves as the major attribute. By way of example, notice that the "preferred" Record Reconstruction Table discussed so far doesn't support either of the potentially useful sort orders P#-J#-S# and J#-S#-P#. (The minor attribute QTY can be ignored in both of these examples, of course, since {S#,P#,J#} is a key.) But there's no reason why we shouldn't build a Record Reconstruction Table that does support those sort orders, or indeed any others we might desire. To be specific, suppose the sort orders we want are as follows:

- For column S# : `S# - P# - J#`
- For column P# : `P# - J# - S#`
- For column J# : `J# - S# - P#`
- For column QTY : `QTY - S# - P# - J#`

I'll leave it as an exercise for you—this is **Exercise 10**—to determine that the corresponding "preferred" Record Reconstruction Table is as shown in Fig. 7.6, and that it does indeed exhibit the desired behavior.

	1	2	3	4
	S#	P#	J#	QTY
1	1	1	3	2
2	8	2	4	6
3	2	3	2	1
4	4	6	5	3
5	5	7	1	8
6	3	8	7	9
7	6	4	8	4
8	7	5	9	5
9	9	9	6	7

Fig.7.6: Another preferred Record Reconstruction Table for the file of Fig. 7.2

Actually, there's a sense in which the "preferred" Record Reconstruction Table of Fig. 7.4 might be "more preferred" than that of Fig. 7.6. Let's agree to say that the table of Fig. 7.4 is *cyclic,* since (unlike the table of Fig. 7.6) it corresponds to all possible cyclic shifts of a certain sequence of the pertinent attributes. Then it turns out that, within such a cyclic table, the pointers that correspond to a given field value within the Field Values Table are guaranteed to be *in sorted order*. For example, consider the J# value J2. The pointers corresponding to that value in the cyclic Record Reconstruction Table of Fig. 7.4 are as follows:

```
1, 6, 7, 8, 9
```

By contrast, the pointers corresponding to that same value in the Record Reconstruction Table of Fig. 7.6 are as follows:

```
1, 7, 8, 9, 6
```

Thanks to this property of the cyclic table, certain additional efficiencies become possible in implementation; for example, certain compression techniques can be applied, and binary searches can be used, on (portions of) columns within such tables. In particular, queries involving the aggregate operators MAX and MIN can now be very efficient, as we'll see in Chapter 10. Further details are beyond the scope of this book.

7.6 Analysis

How many possible major-to-minor orderings are there for a given file? In the case of shipments, there are four fields, and hence 4! = 24 “complete” orderings if we consider ascending sequence only, or 2^4 * 4! = 16 * 24 = 384 such orderings if descending sequence is taken into account as well. (By the term “complete ordering” here, I mean the applicable ORDER BY specifies all four attributes.) And each of the Record Reconstruction Tables discussed in this chapter so far supports just four of those orderings, or eight if we include reverse orderings too. On the face of it, therefore, it looks as if we'd need six different Record Reconstruction Tables for the shipments file to support all possible “complete” major-to-minor orderings, or *forty-eight* if we wanted to take descending sequence into account as well.

In practice, of course, the prospect is usually not nearly so bleak. Here are some reasons why not:

- First of all, many orderings that are logically possible are simply not interesting. Some attributes might never participate in an ORDER BY specification at all (I gave the example near the end of the previous chapter of an attribute whose values are text strings representing natural-language comments). Other attributes might never participate in the major position (QTY might be an example here, in the case of relation SPJ).

- Second, ORDER BY specifications that involve all of the attributes of a user-level relation are quite rare in practice. And I've already pointed out that if the Record Reconstruction Table supports, say, a major-to-minor ordering on attributes *A*-*B*-*C*-*D* (in that sequence), then it implicitly supports the major-to-minor orderings on attributes *A*-*B*-*C* (in that sequence), on *A*-*B* (in that sequence), and on *A*.

- Third, in any ORDER BY specification that includes all of the attributes of some key, any attributes to the right of the rightmost of those key attributes within that specification can simply be ignored. Thus, for example, *any* Record Reconstruction Table for suppliers (not shipments) will certainly support ordering on the sole key {S#}, and will therefore automatically support all six of the following major-to-minor orderings:

```
S# - SNAME — STATUS - CITY
S# - SNAME — CITY - STATUS
S# - STATUS — CITY - SNAME
S# - STATUS — SNAME - CITY
S# - CITY — SNAME - STATUS
S# - CITY — STATUS - SNAME
```

- Fourth, it's easy to guess in practice which particular orderings are going to be useful.

- Last, it's easy to build additional Record Reconstruction Tables if they're needed.

Endnotes

1. As you might know, both this relation and the original suppliers relation are based on a running example used extensively in reference [32] and other database writings of mine.
2. Well, not entirely for free; if we're not careful, updates to the shipments relation could imply updates to the Record Reconstruction Table that could in turn cause that table to lose the desirable properties we're talking about. So the implementation has to make sure that such effects don't occur. Details of what's involved in this process are beyond the scope of this book; suffice it to say that the problem can be and has been solved (and implemented), and the solution involves comparatively little performance overhead.