10 Implementing the Relational Operators

10.1 Introduction

I've now completed my tutorial overview of the basic TR model; I've described the core concepts (principally the Field Values Table and the Record Reconstruction Table) in Chapters 4 and 5, and some very important refinements to those concepts (major-to-minor orderings, condensed columns, and merged columns) in Chapters 7, 8, and 9, respectively. I've also given some idea in Chapter 6 as to what's involved in implementing the INSERT, DELETE, and UPDATE operators. In the present chapter, I want to say a little more about what's involved in using the TR model to implement the relational operators restrict, project, and the rest: partly just to illustrate TR in action, as it were, and partly to reinforce my claim that TR is indeed an excellent foundation on which to implement the relational model (even without all of the additional refinements that I don't intend to discuss in this introductory book—refinements that, as I'm sure you'd expect, offer the possibility of numerous additional improvements).

Let me say immediately that I don't want to get into a lot of detail in what follows—I just want to indicate in outline how certain relational operators might be implemented in terms of the TR model, and offer some observations on the differences between TR and "prior art" in this regard. I'll base my examples on the suppliers and shipments relations shown in Fig. 10.1 (a repeat of Fig. 9.9). The corresponding Field Values Table—a *merged* table, please note—is shown in Fig. 10.2 (a repeat of Fig. 9.14); corresponding Record Reconstruction Tables are shown in Figs. 10.3 (a repeat of Fig. 9.11) and 10.4 (a repeat of Fig. 9.13, except that columns *2-4* have been renumbered as columns *5-7* in order to agree with the column numbering in Fig. 10.2). *Note:* You might want to make a copy of these figures for subsequent reference.

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

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. 10.1: The suppliers and shipments relations S and SPJ

	1	2	3	4	5	6	7
	S#	SNAME	STATUS	CITY	P#	J#	QTY
1	S1[1:2]	Adams[1:1]	10[1:1]	Athens[1:1]	P1[1:4]	J1[1:4]	100[1:2]
2	S2[3:5]	Blake[2:2]	20[2:3]	London[2:3]	P2[5:6]	J2[5:9]	200[3:6]
3	S3[6:9]	Clark[3:3]	30[4:5]	Paris [4:5]	P3[7:9]		500[7:9]
4	S4[:]	Jones[4:4]					
5	S5[:]	Smith[5:5]					

Fig. 10.2: Merged Field Values Table for suppliers and shipments

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

Fig. 10.3: Record Reconstruction Table for suppliers

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

Fig. 10.4: Record Reconstruction Table for shipments

One last preliminary point: I won't bother to include any discussion of ORDER BY operations in my examples, because (a) I think they've been adequately discussed in earlier chapters already, and in any case (b) ORDER BY isn't really a relational operator as such, inasmuch as it doesn't produce a relation as its result (see Chapter 2, especially Sections 2.1 and 2.2).[1]

10.2 Restrict

Consider the following simple SQL query, which asks for a restriction of the shipments relation to just those tuples in which the shipment quantity is 200 (an *equality* restriction):

```
SELECT SPJ.S#, SPJ.P#, SPJ.J#, SPJ.QTY
FROM SPJ
WHERE SPJ.QTY = 200 ;
```

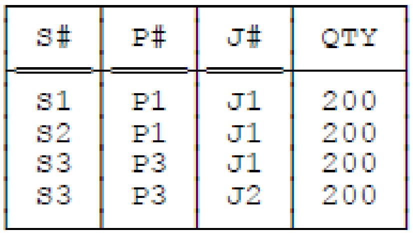
To implement this query, we[2] can start by doing a binary search on column QTY of the Field Values Table (Fig. 10.2), looking for a cell containing the value 200 (note that such a cell must be unique if it exists at all, because the column is condensed). If the search fails, we know immediately that the result of the query is an empty relation (one with no tuples). In the case at hand, however, the search succeeds; cell [*2,7*] of the Field Values Table is the one we want, and it contains, in addition to the specified QTY value, the row range [*3*:*6*]. It follows immediately that cells [*3,7*], [*4,7*], [*5,7*], and [*6,7*] of the shipments Record Reconstruction Table:

a) Contain row numbers for the cell in the merged Field Values Table that contains the QTY value 200 (and indeed they do all include the row number *2*), and
b) Contain row numbers for the "next" cell in the shipments Record Reconstruction Table.

Zigzags can therefore be constructed by following the appropriate pointer rings in the shipments Record Reconstruction Table. In the example, those zigzags look like this:

- *[3,7], [1,1], [2,5], [2,6]*
- *[4,7], [3,1], [3,5], [3,6]*
- *[5,7], [8,1], [7,5], [4,6]*
- *[6,7], [9,1], [9,5], [6,6]*

Following these zigzags through the shipments Record Reconstruction Table and accessing the merged Field Values Table accordingly, we obtain the desired result:



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

For a second example, let's modify the query so that it involves a "less-than" comparison instead of an "equals" one:

```
SELECT SPJ.S#, SPJ.P#, SPJ.J#, SPJ.QTY
FROM SPJ
WHERE SPJ.QTY < 150 ;
```

It should be clear that this query too is easily handled, this time by:

a) Doing a *sequential* search (instead of a binary one) on column QTY of the Field Values Table;
b) Reconstructing all corresponding records, and hence user-level tuples, for each cell encountered during that search; and
c) Stopping as soon as we find a cell in column QTY of the Field Values Table that contains a QTY value of 150 or greater.

Here's the result:

S#	P#	J#	QTY
S1	P3	J2	100
S3	P1	J1	100

Now consider this query:

```
SELECT SPJ.S#, SPJ.P#, SPJ.J#, SPJ.QTY
FROM SPJ
WHERE SPJ.S# = S#('S3') AND SPJ.QTY = 100 ;
```

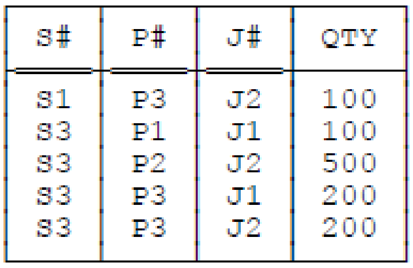
Here the WHERE clause involves two separate equality comparisons ANDed together. By means of searches on the S# and QTY columns of the Field Values Table, however, we can easily discover, from the applicable row ranges, that there are four shipments with supplier number S3 but only two with quantity 100. The best strategy is therefore to use the zigzags associated with quantity 100 and check during record reconstruction to see whether the supplier number is S3, stopping reconstruction of the record in question if it isn't.[3] Here's the result:

S#	P#	J#	QTY
S3	P1	J1	100

Finally, let's consider the effect of replacing the AND by an OR:

```
SELECT SPJ.S#, SPJ.P#, SPJ.J#, SPJ.QTY
FROM SPJ
WHERE SPJ.S# = S#('S3') OR SPJ.QTY = 100 ;
```

We can implement this query by, first, finding all tuples for supplier S3, and then finding all tuples not already found in the first step that have QTY value 100 (or the other way around). Assuming, reasonably enough, that the two steps are executed in such a manner that the two results produced are ordered in the same way (in ascending S# order, say), then they can simply be merged to produce the desired overall result. That result looks like this:



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

One happy—but novel—result of the foregoing is that, loosely speaking, OR and UNION have the same performance characteristics. That is, the following logically equivalent SQL query should be implemented in exactly the same way (and therefore exhibit exactly the same performance) as the one shown above:

```
SELECT SPJ.S#, SPJ.P#, SPJ.J#, SPJ.QTY
FROM SPJ
WHERE SPJ.S# = S#('S3')
UNION
SELECT SPJ.S#, SPJ.P#, SPJ.J#, SPJ.QTY
FROM SPJ
WHERE SPJ.QTY = 100 ;
```

Let me close the present section by contrasting the implementation approaches sketched above with what direct-image systems typically have to do. In general, such systems don't have the same kind of exact cardinality information that TR does;[4] to be specific, they typically don't know exactly how many tuples have a given value for a given attribute at a given time. Instead, they have to execute some kind of *statistics utility* every so often in order to compute those cardinalities, and then store them away somewhere. For example, in IBM's DB2 product [45], the utility in question is called RUNSTATS, and the computed statistics—cardinalities and other similar information—are stored in the DB2 catalog. Typically, the database administrator will ask for RUNSTATS to be executed whenever the database is reorganized or whenever it's been heavily updated. Quite apart from the overhead involved in actually running the utility, the fact is that computed values will naturally be out of date and inaccurate much of the time, and the optimizer might thus fail to choose the best strategy for implementing the query.

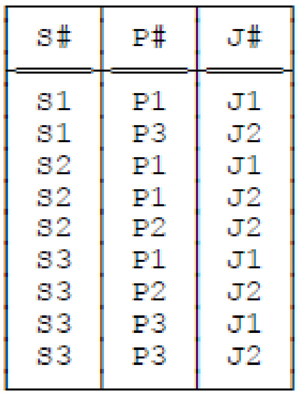
Note: You might reasonably object that the statistics will be out of date with TR too, if the implementation compiles user requests ahead of time (as DB2 and certain other SQL systems in fact do), instead of when those requests are actually executed. The point is, however, that the access path selection process is so simple and straightforward in TR that there's very little point in compiling requests ahead of time—not to mention the fact that TR will almost certainly select the access path that genuinely is optimal. This state of affairs is in strong contrast to "prior art," where the optimizer has to do a great deal of computation and yet still fails, frequently, to come up with the overall best access path.

10.3 Project

Here's an SQL example of a query involving projection ("Project the shipments relation over attributes S#, P#, and J#"):

```
SELECT SPJ.S#, SPJ.P#, SPJ.J#
FROM SPJ ;
```

Implementing this query is straightforward; essentially, we just go through the usual file reconstruction process for shipments, but skip the reconstruction step for attribute QTY in each record. Here's the result:



S#	P#	J#
S1	P1	J1
S1	P3	J2
S2	P1	J1
S2	P1	J2
S2	P2	J2
S3	P1	J1
S3	P2	J2
S3	P3	J1
S3	P3	J2

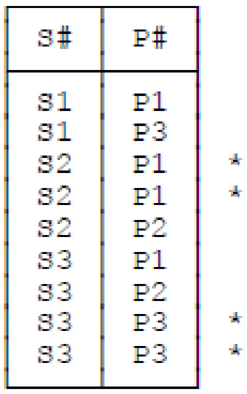
However, you might have noticed that I was cheating a little in this example. Since the attributes over which the projection is taken—that is, the attributes that aren't "projected away"—include all of the attributes of the sole key {S#,P#,J#} for relation SPJ, we know ahead of time that the query can't possibly produce any duplicate tuples. But suppose I change the query slightly, thus:

```
SELECT SPJ.S#, SPJ.P#
FROM SPJ ;
```

If you examine the previous result, you'll see that:

a) There are two tuples that both contain supplier number S2 and part number P1, and
b) There are two tuples that both contain supplier number S3 and part number P3,

and so it looks as if this query ought to produce a result that looks like this:



S#	P#	
S1	P1	
S1	P3	
S2	P1	*
S2	P1	*
S2	P2	
S3	P1	
S3	P2	
S3	P3	*
S3	P3	*

As a matter of fact, this *is* the result that SQL would give. However, that result is not a relation—it includes duplicate tuples (flagged above with asterisks). In particular, it has no candidate key, and a fortiori no primary key (notice that I haven't shown any attributes with double underlining).

Of course, TR can certainly produce this nonrelational result if desired—I mean, it can be used to implement SQL systems as well as relational ones, as already mentioned in Chapter 3—but I'm interested here in implementing relational operations specifically. In order to request the true relational projection operation (to obtain the true relational result) in an SQL system, we would have to amend the query to include the specification DISTINCT, as follows:[5]

```
SELECT DISTINCT SPJ.S#, SPJ.P#
FROM SPJ ;
```

The implementation of this revised query is essentially the same as before, except that the system should if possible process the Record Reconstruction Table for shipments in a sequence that will deliver tuples according to the major-to-minor ordering S#-then-P# (or P#-then-S#). In the example, this ordering is obtained by processing the Record Reconstruction Table (Fig. 10.4) in sequence by the S# column. Duplicates will be adjacent in this ordering and thus can easily be eliminated. The final result is:

S#	P#
S1	P1
S1	P3
S2	P1
S2	P2
S3	P1
S3	P2
S3	P3

Actually, this result can be obtained more directly from the Record Reconstruction Table for shipments (that is, without first constructing and then explicitly eliminating duplicates). Here are the first two columns of that table, extracted from Fig. 10.4:

	1	*5*
	S#	P#
1	*1▪2*	*1▪1*
2	*1▪8*	*1▪2*
3	*2▪3*	*1▪3*
4	*2▪4*	*1▪7*
5	*2▪5*	*2▪8*
6	*3▪1*	*2▪9*
7	*3▪6*	*3▪4*
8	*3▪7*	*3▪5*
9	*3▪9*	*3▪6*

Now consider (by way of example) supplier S2. From the row range [*3*:*5*] for this supplier in the Field Values Table (Fig 10.2), we know among other things that the rows of the shipments Record Reconstruction Table that apply to this supplier are rows *3, 4,* and *5*—that is, the applicable *cells* of that table are [*3*,*1*], [*4*,*1*], and [*5*,*1*], respectively. These cells happen to contain "next cell" row numbers *3, 4,* and *5,* respectively (see Fig. 10.4), and so the "next" cells in the Record Reconstruction Table, according to the usual zigzags, are cells [*3*,*5*], [*4*,*5*], and [*5*,*5*], respectively. And these latter cells contain pointers to the Field Values Table rows *1, 1,* and *2,* respectively. It's thus immediately clear that there are only two *distinct* part numbers corresponding to supplier S2—the one in the Field Values Table cell [*1*,*5*], which is P1, and the one in the Field Values Table cell [*2*,*5*], which is P2.

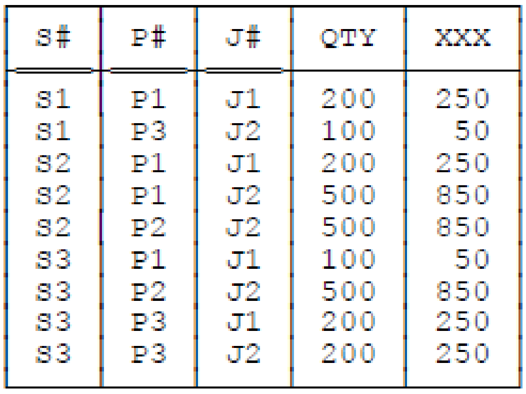
I'll close this section by pointing out explicitly that the example we've been discussing illustrates another important application of the major-to-minor orderings discussed in detail in Chapter 7. To be specific, such orderings can be very helpful in implementing the internal-level operation of eliminating duplicates. In general, duplicate elimination is required in connection with projection operations (as we've just seen), also with union operations (see Section 10.7) and certain aggregation operations (see Section 10.5).

10.4 Extend

You might possibly not be familiar with the relational *extend* operator (the term "extend" isn't used in SQL contexts, at least not with the meaning intended here, though SQL does provide the desired functionality). Basically, the extend operator takes a relation and returns a relation containing an extended form of each tuple from the given relation, where the extension in each case consists of an additional attribute value that's computed in accordance with some specified computational expression. Here's an—admittedly rather contrived—SQL example:

```
SELECT DISTINCT SPJ.S#, SPJ.P#, SPJ.J#, SPJ.QTY,
                                ( ( 2 * SPJ.QTY ) - 150 ) AS XXX
FROM SPJ ;
```

Result:



S#	P#	J#	QTY	XXX
S1	P1	J1	200	250
S1	P3	J2	100	50
S2	P1	J1	200	250
S2	P1	J2	500	850
S2	P2	J2	500	850
S3	P1	J1	100	50
S3	P2	J2	500	850
S3	P3	J1	200	250
S3	P3	J2	200	250

The only point I want to make in connection with this example is that if we know this query is going to be executed fairly frequently, then we can treat the "computed" attribute XXX just like the regular ("base") attributes S#, P#, and so on; to be specific, we can map it to a column of its own in the Field Values Table. That column can then be sorted and condensed (possibly even merged), just like other such columns, and analogous benefits—fast binary search, use in major-to-minor orderings, and so on—will then immediately accrue.

How then can we know whether a given query will be frequently executed? Well, one possibility is to let the database administrator tell us, of course. Another is to *guess* ... If the foregoing SQL query is specified as the defining expression for a **view,** as here—

```
CREATE VIEW XSPJ
     AS SELECT DISTINCT SPJ.S#, SPJ.P#, SPJ.J#, SPJ.QTY,
                                   ( ( 2 * SPJ.QTY ) - 150 ) AS XXX
           FROM SPJ ;
```

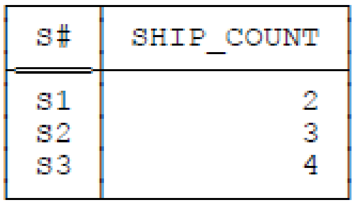
—then it's a pretty safe bet that the query is indeed going to be executed fairly frequently.

10.5 Summarize

Summarize is the relational operator that underpins SQL's aggregation and GROUP BY operations. Here's an SQL example:

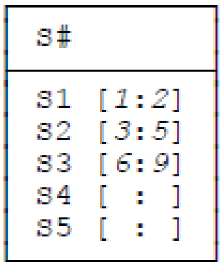
```
SELECT DISTINCT SPJ.S#, COUNT(*) AS SHIP_COUNT
FROM SPJ
GROUP BY SPJ.S# ;
```

The effect of this query is to "summarize" the shipments relation in a certain way. To be specific, it returns a relation that contains a tuple for each distinct supplier number in SPJ, giving a count (SHIP_COUNT) of the number of shipments the supplier in question is involved in. The result looks like this:



S#	SHIP_COUNT
S1	2
S2	3
S3	4

Observe now that this result is directly obtainable from the S# column of the Field Values Table. Here is that column, extracted from Fig. 10.2:



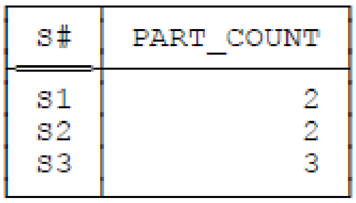
S#
S1 [*1*:*2*]
S2 [*3*:*5*]
S3 [*6*:*9*]
S4 [:]
S5 [:]

Recall that the row ranges indicate (among other things) which rows of the uncondensed Field Values Table for shipments the corresponding supplier number would appear in, if such a table were actually to be built. Thus we can see immediately that there are two shipments for supplier S1, three for supplier S2, four for supplier S3, and none at all for suppliers S4 and S5.[6] Note, however, that the result doesn't include tuples for suppliers S4 and S5 (with zero counts), because the SQL query specified "FROM SPJ," and suppliers S4 and S5 don't appear in relation SPJ at all. A relational query using SUMMARIZE that does include suppliers S4 and S5 in the result can easily be formulated (and easily implemented in TR). An SQL query to do the same thing can be formulated too, but the specifics are rather more complicated, and the details are beyond the scope of this book; for more discussion, see reference [32].

By the way, it would make no difference to either the meaning or the result of the foregoing SQL query if we were to replace the COUNT argument "*" by SPJ.P#, or SPJ.J#, or SPJ.QTY, or SPJ.QTY + 1, or indeed by just about any other syntactically valid expression you can think of—*unless* the expression in question is preceded by the specification DISTINCT, as here:

```
SELECT DISTINCT SPJ.S#, COUNT ( DISTINCT SPJ.P# ) AS PART_COUNT
FROM SPJ
GROUP BY SPJ.S# ;
```

The effect of this revised query is to return a relation that contains a tuple for each distinct supplier number in SPJ, giving a count NP of the number of *distinct* parts the supplier in question is shipping, thus:



S#	PART_COUNT
S1	2
S2	2
S3	3

This revised query requires a revised implementation, too: Basically, the system now needs to use the Record Reconstruction Table for shipments, processing it in a sequence that will deliver tuples according to the major-to-minor ordering S#-then-P#. Duplicate part numbers for a given supplier will be adjacent in this ordering and thus can easily be eliminated from the corresponding count. (As in the case of projection—see Section 10.3—it shouldn't be necessary actually to materialize the duplicates before eliminating them; the necessary information can in fact be obtained directly from the Record Reconstruction Table.)

Let's consider some of the other aggregate operators. MAX and MIN are easy enough. For example, consider the SQL query:

```
SELECT DISTINCT SPJ.S#, MIN ( SPJ.QTY ) AS MNQ
FROM SPJ
GROUP BY SPJ.S# ;
```

Here's the result:

S#	MNQ
S1	100
S2	200
S3	100

To see how this query is implemented, consider supplier S2 once again. As we already know (see the discussion of projection in Section 10.3), the cells in the shipments Record Reconstruction Table that correspond to this supplier number are [*3,1*], [*4,1*], and [*5,1*], respectively. Following the zigzags to the corresponding QTY cells in that table, we find that those cells contain pointers to the Field Values Table rows *2, 3,* and *3,* respectively. Since the QTY column (like all columns) in that table is kept in ascending order, it's immediately clear that the minimum QTY value for supplier S2 is the one in row *2* of the Field Values Table—namely, the QTY value 200.

Note: It should be obvious that it makes no difference in the case of MAX and MIN whether or not the argument to the aggregate operator includes a DISTINCT specification.

Other aggregate operators for which TR technology is particularly suited include MEDIAN and MODE. In case you're unfamiliar with these operators, let me explain them briefly here. Suppose we're given a collection of values, possibly including duplicates. Then the *median* of that collection is the value that appears in the middle position when the values are sorted, while the *mode* is the value that appears the most frequently. (Of course, these definitions require certain refinements, beyond the scope of this book, in order to take care of the question of ties and the like, but you get the general idea.) I'll leave it to you to figure out the corresponding TR implementation in each case.

10.6 Join

The join operation is often regarded as the sine qua non of relational systems.[7] Certainly it's extremely important; some might even say that relational systems stand or fall on the basis of how well—how effectively, how efficiently—they implement joins. What's more, there's a widespread perception that joins must perform poorly, almost by definition. Here's a typical quote (from an article critizing relational systems in general and the proposals of reference [40] in particular): "Database application developers ... have been baffled by the intolerable performance [incurred] ... by performing joins" [54]. And reference [63] has this to say:

In prior art database systems, joins tend to be extremely costly in storage space and/or processing time, requiring either preindexed data to maintain sortedness or a time-intensive search involving multiple passes over the entirety of each attribute that is being joined.

—from the Initial Patent

Let's take a closer look. Reference [32] describes a variety of techniques for implementing joins, the following among them:

- Brute force
- Index lookup
- Hash lookup
- Merge
- Hash
- Various combinations of the foregoing

Let me focus first on the *brute force* technique. Let *r* and *s* be the relations to be joined; let *r* and *s* have *M* tuples and *N* tuples, respectively, and let them have just one common attribute, *A*.[8] Let *R* and *S* be direct-image stored files corresponding to *r* and *s,* respectively, with stored records, in sequence, *R*[1], *R*[2], ..., *R*[*M*] and *S*[1], *S*[2], ..., *S*[*N*], again respectively. Here then is the brute force algorithm:

```
do i := 1 to M ;
     do j := 1 to N ;
           if R[i].A = S[j].A then
           append joined record R[i] * S[j] to result ;
     end ;
end ;
```

(I'm using the expression *R*[*i*] * *S*[*j*] to denote the joined record that's formed from the records *R*[*i*] and *S*[*j*].)

As you can see, the brute force technique is very simple-minded—basically, it just examines all possible combinations of records, one from *R* and one from *S,* and joins them together if and only if they have the same value for the common attribute *A* (or for the stored field corresponding to the common attribute *A,* rather). *Note:* The brute force algorithm is often referred to as "nested loops," but this name is misleading because nested loops are in fact involved in all of the conventional implementation algorithms.

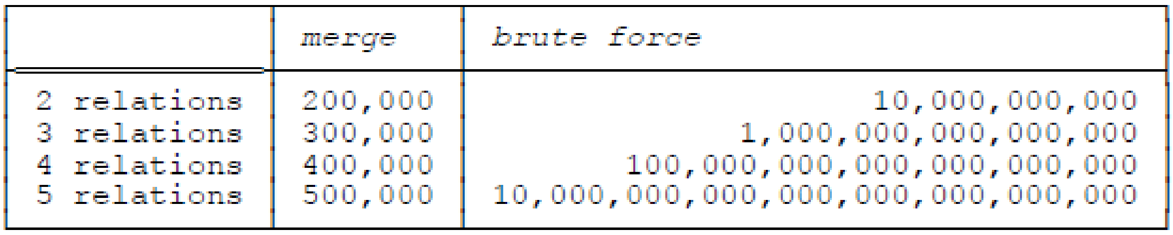
Now, it should be obvious that the brute force approach involves a total of *M***N* record read operations. It should also be obvious that if we wanted to join *three* relations, *r, s,* and *t,* say, then the brute force approach will involve *M***N***P* record reads (where *P* is the number of tuples in *t*), and so on. In other words, the costs associated with the brute force algorithm are inherently **multiplicative** in nature. For that reason, that algorithm is generally regarded as the worst case, which is precisely why so much energy has been expended over the past 30 years or so on alternative approaches (index lookup, hash lookup, and the rest).

I don't want to go into a lot of detail on those alternative approaches here. Suffice it to say that they're all aimed, in one way or another, toward the goal of never having to read any record twice—or, preferably, toward the more demanding goal of being able to read each stored file in sequence just once (clearly an optimal state of affairs).

- For example, indexes or hashes on *R.A* and *S.A* could certainly mean that no record of either *R* or *S* is ever read twice. However, they probably wouldn't mean that the stored files are read in sequence just once, as I pointed out in Chapter 2. Also, of course, indexes and hashes lead to other problems, again as discussed in Chapter 2.

- Alternatively, we could sort the two stored files appropriately and then do a *merge* join—and merge join does mean that each stored file is read in sequence just once. Thus, a merge join of *r* and *s* will involve *M+N* record reads; a merge join of the three relations *r, s,* and *t* will involve *M+N+P* record reads; and so on. In other words, the costs associated with the merge approach are inherently **additive** (or **linear**), not multiplicative, in nature. (Of course, I'm ignoring the sort costs here, and those costs can be very significant in practice.)

I'd like to emphasize the dramatic difference between linear and multiplicative costs. Suppose for simplicity that every relation has 100,000 tuples (not at all a large number, by the way, in modern databases). Then the following table shows the number of record reads involved in various joins implemented by merge vs. the same joins implemented by brute force (assuming a direct-image style of implementation in both cases, of course):



	merge	*brute force*
2 relations	200,000	10,000,000,000
3 relations	300,000	1,000,000,000,000,000
4 relations	400,000	100,000,000,000,000,000,000
5 relations	500,000	10,000,000,000,000,000,000,000,000

Note in particular that each step (from two relations to three, from three to four, and so on) involves several orders of magnitude performance degradation with the brute force approach. In order to emphasize the point, suppose each record read takes ten microseconds. Then a merge join of the five relations will take just five seconds, while a brute force join of the same five relations will take over three trillion years, or some 200 times the current best estimate of the age of the universe (!). No wonder merge join is a preferred technique ... But the trouble with merge join, of course, is that it requires the stored files to be sorted into appropriate sequence first (that's why the technique is usually called, more specifically, *sort*/merge). And the beauty of the TR approach, as I've shown in earlier chapters, is that the stored files are already in the desired sort order, always. As I put it in Chapter 4, TR lets us do a sort/merge join without having to do the sort (indeed, we saw in Chapter 9 that it might effectively let us do the join without having to do the merge either). Thus, TR always does a merge join. Note the following implications:

- The more relations that need to be joined, the more the gain. In other words, the more complex the query, the more significant the TR advantage over direct-image systems (as already noted in Chapter 5).

- Because all joins are implemented the same way, we don't have to do that complex access path selection process that those direct-image systems do have to do.

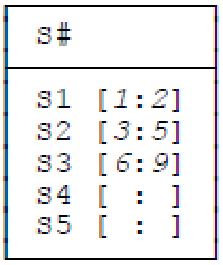
- That access path selection process that direct-image systems have to do is of dubious accuracy anyway, because of the difficulty of estimating intermediate result sizes, among other reasons.

- In fact, as reference [32] shows, there can easily be a huge number of possible strategies for implementing any given query in direct-image systems, precisely because of all the redundancies that indexes and other auxiliary structures introduce. For this reason, those systems typically employ a variety of heuristics for "reducing the search space"—that is, for eliminating certain strategies very early on in the access path selection process (possibly never even considering them at all). Those heuristics in turn (a) make the implementation still more complicated and (b) imply that a good strategy will sometimes be rejected in favor of a bad one.

By way of example, let's consider what's involved in TR in implementing the following SQL query (which asks for suppliers and shipments to be joined on supplier numbers):

```
SELECT DISTINCT S.S#, S.SNAME, S.STATUS, S.CITY, SPJ.P#, SPJ.J#, SPJ.QTY
FROM S, SPJ
WHERE S.S# = SPJ.S# ;
```

Here again is the S# column from the Field Values Table (extracted from Fig. 10.2):



S#
S1 [*1*:*2*]
S2 [*3*:*5*]
S3 [*6*:*9*]
S4 [:]
S5 [:]

From the information in this column we can see immediately that:

- The first tuple of relation S (for supplier S1) joins to the first and second tuples of relation SPJ.[9] The two joined tuples can be built by starting at cell [*1,1*] of the suppliers Record Reconstruction Table and cells [*1,1*] and [*2,1*] of the shipments Record Reconstruction Table.

- The second tuple of relation S (for supplier S2) joins to the third, fourth, and fifth tuples of relation SPJ. The three joined tuples can be built by starting at cell [*2,1*] of the suppliers Record Reconstruction Table and cells [*3,1*], [*4,1*], and [*5,1*] of the shipments Record Reconstruction Table.

- The third tuple of relation S (for supplier S3) joins to the sixth, seventh, eighth, and ninth tuples of relation SPJ. The four joined tuples can be built by starting at cell [*3,1*] of the suppliers Record Reconstruction Table and cells [*6,1*], [*7,1*], [*8,1*], and [*9,1*] of the shipments Record Reconstruction Table.

Execution of the query is now complete. Note in particular that the fourth and fifth tuples of relation S (for suppliers S4 and S5) don't join to any tuples of relation SPJ at all.

Now, I mentioned earlier in this section (by way of an endnote) that there are other kinds of joins as well as the natural join: equijoins, greater-than joins, and so on. Here's an SQL example of a greater-than join: to be specific, a greater-than join over city names between the suppliers relation S and the parts relation P from Chapter 8. *Note:* "Greater than" here just means—let's assume—"later in alphabetic ordering than" (recall our assumption in Chapter 2 that city names are simple CHAR strings).

```
SELECT DISTINCT S.S#, S.SNAME, S.STATUS, S.CITY AS SCITY
                 P.P#, P.PNAME, P.COLOR, P.WEIGHT, P.CITY AS PCITY
FROM S, P
WHERE S.CITY > P.CITY ;
```

Here's the result:

S#	SNAME	STATUS	SCITY	P#	PNAME	COLOR	WEIGHT	PCITY
S2	Jones	10	Paris	P1	Nut	Red	12.0	London
S2	Jones	10	Paris	P4	Screw	Red	14.0	London
S2	Jones	10	Paris	P6	Cog	Red	19.0	London
S2	Jones	10	Paris	P3	Screw	Blue	17.0	Oslo
S3	Blake	30	Paris	P1	Nut	Red	12.0	London
S3	Blake	30	Paris	P4	Screw	Red	14.0	London
S3	Blake	30	Paris	P6	Cog	Red	19.0	London
S3	Blake	30	Paris	P3	Screw	Blue	17.0	Oslo

And here are the CITY columns from the suppliers and parts Field Values Tables (suppliers on the left, parts on the right):

CITY
Athens [*1*:*1*]
London [*2*:*3*]
Paris [*4*:*5*]

CITY
London [*1*:*3*]
Oslo [*4*:*4*]
Paris [*5*:*6*]

For convenience, let's merge these two columns together, as follows[10] (the first row range for each city corresponds to suppliers and the second to parts):

CITY
Athens [*1*:*1*] [:]
London [*2*:*3*] [*1*:*3*]
Oslo [:] [*4*:*4*]
Paris [*4*:*5*] [*5*:*6*]

It's clear from this merged column that the "fourth" and "fifth" supplier tuples both join to each of the "first," "second," "third," and "fourth" part tuples—where "fourth," fifth," etc., are to be interpreted in terms of CITY ordering in both cases—and nothing else joins to anything else. Thus, I think you can see that the desired greater-than join can again be implemented by a kind of merging process, although the details are a little more complicated than they are in the natural join case; in particular, several passes are needed over the row ranges for either parts or suppliers (not both). *Note:* This latter fact might be a good reason for physically storing row ranges in a table of their own, separate from the Field Values Table, as suggested in Chapter 8.

10.7 Union, Intersect, and Difference

The relational operators union, intersect, and difference all require their two input relations to have exactly the same attributes [33]. As a basis for my examples in this section, therefore, I'll consider the *projections* of the suppliers and parts relations on their CITY attributes (since those two projections certainly do have exactly the same attributes). Here then are some SQL examples, with corresponding results:

```
SELECT DISTINCT S.CITY
FROM   S
UNION
SELECT DISTINCT P.CITY
FROM   P ;
```

CITY
Athens
London
Oslo
Paris

```
SELECT DISTINCT S.CITY
FROM   S
INTERSECT
SELECT DISTINCT P.CITY
FROM   P ;
```

CITY
London
Paris

```
SELECT DISTINCT S.CITY
FROM   S
EXCEPT
SELECT DISTINCT P.CITY
FROM   P ;
```

CITY
Athens

```
SELECT DISTINCT P.CITY
FROM   P
EXCEPT
SELECT DISTINCT S.CITY
FROM   S ;
```

CITY
Oslo

Note that SQL uses the keyword EXCEPT to denote the relational difference operator. Note too that UNION, INTERSECT, and EXCEPT—unlike SELECT—all eliminate duplicates by default in SQL (implying that all of the DISTINCT operators shown above are in fact logically unnecessary).

Here now, repeated from the previous section, is a merged Field Values Table CITY column (supplier row ranges on the left, part row ranges on the right):

```
CITY
------------------------------
Athens [1:1] [ : ]
London [2:3] [1:3]
Oslo   [ : ] [4:4]
Paris  [4:5] [5:6]
```

The use of this merged column in implementing the foregoing union, intersect, and difference operations should be obvious. In essence:

- *Union:* A given city name appears in the result if and only if it has a nonempty row range for suppliers or parts or both. In other words, the union is just the set of all city names in the merged column.
- *Intersect:* A given city name appears in the result if and only if it has a nonempty row range for both suppliers and parts.
- *Difference:* For the difference between supplier cities and part cities, in that order, a given city name appears in the result if and only if it has a nonempty row range for suppliers and an empty one for parts. Similarly, for the difference between part cities and supplier cities, in *that* order, a given city name appears in the result if and only if it has a nonempty row range for parts and an empty one for suppliers.

All of these operations can clearly be implemented in a single pass over the merged Field Values Table CITY column.

Incidentally, if that CITY column contains a large number of entries, the performance of intersect and difference operations, at least, might be improved by means of *bitmaps*. In the example, we would have two such bitmaps, one to indicate whether the city name in question appears in the suppliers relation and the other to indicate whether it appears in the parts relation. Here's a modified version of the merged column that includes such bitmaps (1 = yes, 0 = no):[11]

CITY			
Athens [*1*:*1*]	1	[:]	0
London [*2*:*3*]	1	[*1*:*3*]	1
Oslo [:]	0	[*4*:*4*]	1
Paris [*4*:*5*]	1	[*5*:*6*]	1

The city names appearing in the intersection can now be pinpointed by executing a logical AND on the two bitmaps, while those appearing in the difference between supplier cities and part cities, in that order, can be pinpointed by executing a logical AND on the suppliers bitmap and the negation (logical complement) of the parts bitmap. Since logical operations like AND and NOT are usually supported directly in hardware, the implementation of the corresponding relational operations now has the potential to be very fast indeed.

10.8 Materializing Derived Relations

Sometimes it's necessary for a relational implementation to materialize some derived relation—that is, to build a concrete representation in storage of the result of some relational expression. Just why and when such materialization might be necessary is a question I don't particularly want to get into here; rather, what I do want to do is examine the question of what's involved in performing such materialization, when it *is* necessary, in the case of TR specifically.

Materializing a derived relation in TR means, of course, building an appropriate set of Field Values and Record Reconstruction Table entries for that relation. One obvious point that arises immediately, therefore, is that materialization is likely to be easier in TR than it is in other approaches, because a single Field Values Table can effectively be shared across several different relations, thanks to the merged-columns feature. In other words, it might not be necessary to build a new Field Values Table for the derived relation at all, in which case it could be argued that materialization as such isn't really being done (because it's simply not needed). These remarks apply directly to the monadic case, where the derived relation is obtained by means of some monadic relational operator (restrict, project, extend, summarize); they might possibly also apply to the dyadic case, where the derived relation is obtained by means of some dyadic relational operator (join, union, intersect, difference). *Note:* I'm using the terms *monadic* and *dyadic* here to refer to relational operators that take one relational operand and two relational operands, respectively.

Let's now make the worst-case assumption; that is, let's assume that we do actually have to build a brand new Field Values Table and a brand new Record Reconstruction Table for the derived relation in question. Suppose, for example, that we need to materialize the result of joining suppliers and parts over city names. Well, it's easy to see intuitively that, in general, the biggest overhead in building a Field Values Table and a Record Reconstruction Table is all the sorting of field values that's required. But in the case at hand, most if not all of the sorting has already been done—every column of the suppliers Field Values Table is already in sorted order, and the same is true of every column of the parts Field Values Table as well. Analogous remarks apply to the other relational operators, of course. (As a matter of fact, they even apply to some extent to the pointer values in the corresponding Record Reconstruction Tables also; they too tend to be sorted, at least partially. See, for example, the Record Reconstruction Table shown in Fig. 7.4 in Chapter 7, also the remarks on this topic at the end of Section 7.5 in that same chapter.)

In a nutshell, then, materialization in TR (a) is needed less often than it is in traditional implementations and (b) is more efficient, when it *is* needed, than it is in traditional implementations.

10.9 A Note Regarding Optimization

This brings me to the end of my discussion of how relational operators can be implemented using the TR model. However, there are still a few topics—three of them, to be precise—that I'd like to say something about, briefly, before I close the chapter. The first has to do with the system optimizer.

The optimizer is, of course, that component of the system that decides how to implement any given user request. Now, I've suggested at numerous points in previous discussions, both in this chapter and in several earlier chapters, that TR makes life easier for the optimizer; to be specific, it makes the access path selection process easier (even completely unnecessary, in some cases). However, I don't want to give the impression that the optimizer is no longer necessary. The fact is, there are two broad facets to the optimizer's job, both of them (in general) important, access path selection and **expression transformation** (sometimes called **query rewrite**). And even if access path selection does become unnecessary (or almost so), query rewrite does not.

Query rewrite is the process of converting a given relational expression into another such expression that (a) is logically equivalent to the original one, in the sense that it's guaranteed to produce the same result when evaluated, but (b) has a good likelihood of being more efficient—that is, performing better—than the original one. I'll give just one simple example (expressed in SQL for reasons of familiarity): The expression

```
SELECT DISTINCT X.CITY
FROM ( SELECT DISTINCT S.S#, S.STATUS, S.CITY
       FROM   S) AS X ;
```

(a projection of a projection) can be "rewritten" as the simpler expression

```
SELECT DISTINCT X.CITY
FROM S AS X ;
```

The rewrite has eliminated one of the projections, and that's why the result is more efficient.

Note: You might be thinking that the foregoing example is somewhat contrived ("no user in his or her right mind would state the query in the first form anyway"—right?). In fact, however, the example is quite realistic. Suppose we have the following view:

```
CREATE VIEW X
     AS SELECT DISTINCT S.S#, S.STATUS, S.CITY
           FROM S ;
```

And suppose the user issues the following query:

```
SELECT DISTINCT X.CITY
FROM X ;
```

Then the first thing the system does in processing this query is (in effect) convert it into the following:

```
SELECT DISTINCT X.CITY
FROM ( SELECT DISTINCT S.S#, S.STATUS, S.CITY
        FROM S ) AS X ;
```

Rewriting this query as previously suggested is thus clearly very desirable.

That said, I should now make it clear that query rewrite is not a TR responsibility as such; rather, it's a task that needs to be performed by code that sits above the TR level. For that reason, I don't want to discuss it any further here.

10.10 A Note Regarding Constraints

The second piece of unfinished business has to do with **integrity constraints**. Such constraints are vitally important, both in theory and in practice (see reference [36]), yet I've said almost nothing about them in this book so far, and it would be very remiss of me to ignore them altogether.

Basically, an integrity constraint is a conditional expression (also known as a boolean, truth-valued, or logical expression) that's required to evaluate to true. Here are a few examples, expressed in natural language for simplicity:

1. Every supplier status value is in the range 1 to 100 inclusive.
2. Every part weight is greater than zero.
3. Every supplier in London has status 20.
4. If there are any parts at all, at least one of them is blue.
5. No two distinct suppliers have the same supplier number.
6. Every shipment involves an existing supplier.
7. No supplier with status less than 20 supplies any part in a quantity greater than 500.

And so on.

Of course, it's the job of the database administrator to state such constraints (using SQL or some other formal language),[12] and it's the job of the DBMS to implement them. But implementing constraints isn't the same thing as implementing the relational operators; in fact, the system component that implements constraints will in all likelihood make use of the relational operators to do so, and therefore will have to invoke the lower-level component that does implement those relational operators. In a TR system, in other words, many constraints—perhaps most—will be implemented by code that sits, not on top of the TR level directly, but on top of the relational operator implementation level that does sit on top of the TR level directly. That's basically why I haven't had much to say about constraints in this book prior to this point.

I must now immediately add that there are likely to be some exceptions—rather important ones—to the foregoing. Consider again the following example:

5. No two distinct suppliers have the same supplier number.

The formal statement of this constraint is, of course, simply a specification to the effect that {S#} is a key—more precisely, a *candidate* key—for the suppliers relation, and the implementation has to guarantee that no two supplier tuples appearing in the suppliers relation at the same time ever have the same supplier number. But as I explained in Chapter 6 (Section 6.5), this guarantee is effectively built into the implementation of the INSERT operator (the UPDATE operator too, as a matter of fact). To repeat the example from that section, suppose we try to insert a supplier tuple for supplier S9. At the TR level, then, the system will have to inspect the supplier number column in the Field Values Table (probably using a binary search), looking for the appropriate insert point for the new supplier number S9; and if it discovers that the supplier number value S9 already exists, then clearly it can reject the INSERT (or UPDATE). In other words, key constraints can and will effectively be implemented directly at the TR level.

The second example I want to discuss is this one:

6. Every shipment involves an existing supplier.

The formal statement of this constraint is a specification to the effect that {S#} in the shipments relation SPJ is a foreign key referencing the candidate key {S#} of the suppliers relation S (every supplier number currently appearing in SPJ must currently appear in S as well). And the point I want to make here is this: If the Field Values Tables for suppliers and shipments are merged on their S# column, as shown in Fig. 10.2, then the mechanism for enforcing this foreign key constraint for shipments is very similar to that discussed above for enforcing the candidate key constraint for suppliers. In other words, foreign key constraints too can, and probably will, effectively be implemented directly at the TR level.

It's appropriate to close this section by mentioning that in direct-image systems, both candidate and foreign key constraints are typically enforced by means of indexes, or sometimes by hashes or other auxiliary structures.

10.11 What's Missing?

The third and last piece of unfinished business has to do with **missing information**. Examples of missing information include such things as "date of birth unknown," "speaker to be announced," "present address not known," and so on. And as you probably know, SQL systems in particular address this issue—or attempt to address it, rather—by means of a construct called a **null**. For example, suppose we know some particular part exists, but we don't know its weight. Then we might say, loosely, that "the weight is null"—meaning, more precisely, that (a) we do know the part has a weight, because all parts have a weight, but (b) to repeat, we don't know what that weight is. So we can't put any sensible value at all in the WEIGHT position within the pertinent tuple; instead, therefore, we *flag* or *mark* that position as "being null."

Now, you've probably noticed that I've said essentially nothing about this topic in this book prior to this point. And the major reason for that omission is that, so far as I'm concerned, *nulls in the foregoing sense have absolutely no place in the relational model*—and, of course, I've been concentrating in this book so far on the application of TR concepts to implementing the relational model specifically. I don't want to get into a lot of detail here as to why I—and indeed most other writers on the relational model, though not all [38]—reject nulls categorically; this book would be the wrong forum for such a discussion. Let me just say, therefore, that:

a) There are very sound reasons, both theoretical and practical,[13] for not including nulls in the relational model itself. See references [18], [32], [40], [43-44], and especially [58] for a discussion of some of the theoretical reasons, and references [18-19] and [22-23] for a discussion of some of the practical ones.
b) There are also very sound reasons for not using nulls, even when they're supported, as they are in SQL. Thus, I recommend strongly that, even if you have to use SQL, you don't try to "take advantage of" the nulls feature of that language. In other words, nulls are contraindicated even when they're supported. See references [17], [32], and [39] for arguments in support of this position.

Given the foregoing state of affairs, I don't propose to discuss the use of TR to implement an SQL-style nulls feature at all. I'll just say that—of course—TR *can* be used to implement such a feature if desired, and that many of the advantages I've been claiming for a TR implementation of the relational model would apply to such an implementation, too. So yes, TR can be used to implement SQL as well as the relational model.[14]

Endnotes

1. If you happen to be familiar with the relational model, you might notice another omission, too: There's no discussion of the relational divide operator. One reason for this omission (not the only one) is that I'll be arguing in Chapter 15 that relational comparisons really ought to be supported. If they are, then the divide operator becomes logically unnecessary [40].
2. I'll use the term "we" throughout this chapter, a trifle sloppily, to mean either the DBMS designers and implementers or the DBMS itself, as the context demands.
3. Checking the supplier number will be quite speedy, too, because column S# and column QTY happen to be logically adjacent within those zigzags. See the remarks on this subject at the very end of Chapter 5.
4. The *cardinality* of a set is the number of elements the set contains.
5. As I've written elsewhere [17], my own recommendation would be that users shouldn't have to waste time thinking about whether a given SQL query can produce duplicates or not but should always specify DISTINCT, and leave it to the system to figure out when such a specification can safely be ignored. Of course, I haven't followed my own advice in this respect in this book so far!—but I'll do so from this point forward (you might like to try the exercise of figuring out in each case whether the DISTINCT can safely be ignored). To quote Hugh Darwen [11]: "If you have to use ... DISTINCT to obtain a true [relational result], do not fail to do so, *but be annoyed about it*" (my italics).
6. Intuitively, the reason COUNT and the other aggregate operators discussed in the present section can be so easily and efficiently implemented in TR is because—as noted in Chapter 8, Section 8.2—the row ranges in the Field Values Table can effectively be regarded as histograms.

7. As you might have already noticed, I use the unqualified term *join* to mean the natural join specifically [33,40]. This practice is both common and convenient in relational contexts. However, other kinds of joins do exist: equijoins, greater-than joins, and so on (see reference [32]). I'll have a little more to say regarding these other kinds of joins toward the end of the present section.
8. I make this assumption for simplicity only. Everything said regarding TR in this section extends gracefully and straightforwardly to the case where there are two or more common attributes (recall from Chapter 9, Section 9.4, that the TR model effectively already includes a means by which two or more attributes can be treated as a single "combined" attribute if desired). Analogous remarks apply to other operators also, including in particular union, intersect, and difference (see Section 10.7).
9. See the remarks at the end of Section 9.2 in Chapter 9 for an explanation of what I mean by expressions like "the first tuple of relation S" and "the first and second tuples of relation SPJ."
10. In fact, the TR join implementation process will do this automatically, if the columns haven't been merged already. Of course, the merging does mean that changes will be required to the corresponding Record Reconstruction Tables, too, but those changes are essentially trivial.
11. The bitmaps are logically redundant, of course. Also, they're nothing to do with bitmap indexing, a topic that was mentioned in passing in Chapter 2 (Section 2.3).
12. All of the examples shown can in fact be formulated in SQL [39]. I omit such formulations for brevity.
13. Actually I believe theoretical reasons *are* practical ones, but that's another big discussion I don't want to get into here.
14. It's appropriate to add that TR is probably much better suited to implementing a truly relational solution—which isn't what the SQL "solution" is!—to the problem of missing information (thanks to Hugh Darwen for this observation). See references [43-44] and [58].