5 Core Concepts (Continued)

5.1 Introduction

This chapter continues our examination of the core constructs of the TR model (principally the Field Values and Record Reconstruction Tables). However, the chapter is rather more of a potpourri than the previous one. Its structure is as follows. Following this short introductory section, Section 5.2 offers some general observations regarding performance. Section 5.3 then briefly surveys the TR operators, and Sections 5.4 and 5.5 take another look at how the Record Reconstruction Table is built and how record reconstruction is done. Sections 5.6 and 5.7 describe some alternative perspectives on certain of the TR constructs introduced in Chapter 4. Finally, Section 5.6 takes a look at some alternative ways of implementing some of the TR structures and algorithms also first described in that previous chapter.

5.2 Some Remarks on Performance

It seems to me undeniable that the mechanisms described in the previous chapter for representing and reconstructing records and files are vastly different from those found in conventional DBMSs, and I presume you agree with this assessment. At the same time, however, they certainly look pretty *complicated* ... How does all of that complexity square with the claims I made in Chapter 1 regarding good performance? Let me remind you of some of the things I said there:

> [TR is] a technology that lets us build database management systems (DBMSs) that are ... orders of magnitude faster than any previous system ... [A] relational system ... using TR technology should dramatically outperform even the fastest of those [previous] systems ... [and] I don't just mean that queries should be faster ... [Updates] should be faster as well.
>
> *—from Chapter 1*

Well, let me say a little more now regarding query performance specifically (I haven't really discussed updates yet, so I'll have to come back to the question of update performance later—actually in the next chapter). Now, any given query involves two logically distinct processes:

a) Finding the data that's required, and then
b) Retrieving that data.

TR is designed to exploit this fact. Precisely because it separates field value information and linkage information, it can treat these two processes more or less independently. To find the data, it uses the Field Values Table; to retrieve it, it uses the Record Reconstruction Table. (These characterizations aren't 100 percent accurate, but they're good to a first approximation—good enough for present purposes, at any rate.) And the Field Values Table in particular is designed to make the finding of data very efficient (for example, via binary search), as we saw in Chapter 4. Of course, it's true that subsequent retrieval of that data then involves the record reconstruction process, and this latter process in turn involves a lot of pointer chasing, but:

- Even in a disk-based implementation, the system will do its best to ensure that pertinent portions of both the Field Values Table and the Record Reconstruction Table are kept in main memory at run time, as we'll see in Part III. Assuming this goal is met, the reconstruction will be done at main-memory speeds.

- The "frills" to be discussed in Chapters 7-9 (as well as others that are beyond the scope of this book) have the effect, among other things, of dramatically improving the performance of various aspects of the reconstruction process.

- *Most important of all:* Almost always, finding the data that's wanted is a much bigger issue than returning that data to the user is. In a sense, the design of the TR internal structures is biased in favor of the first of these issues at the expense of the second. Observe the implication: *The more complex the query, the better TR will perform*—in comparison with traditional approaches, that is. (Of course, I don't mean to suggest by these remarks that record reconstruction is slow or inefficient—it isn't—nor that TR performs well on complex queries but not on simple ones. I just want to stress the relative importance of finding the data in the first place, that's all.)

I'd like to say more on this question of query performance. In 1969, in his very first paper on the relational model [5], Codd had this to say:

> Once aware that a certain relation exists, the user will expect to be able to exploit that relation using any combination of its attributes as "knowns" and the remaining attributes as "unknowns," because the information (like Everest) is there. This is a system feature (missing from many current information systems) which we shall call (logically) symmetric exploitation of relations. Naturally, symmetry in performance is not to be expected.
>
> *—E. F. Codd*

Note: I've reworded Codd's remarks just slightly here. In particular, the final sentence (the caveat concerning performance) didn't appear in the original 1969 paper [5] but was added in the expanded 1970 version [6].

Anyway, the point I want to make is that the TR approach gives us symmetry in performance, too—or, at least, it comes much closer to doing so than previous approaches ever did. This is because, as we saw in Chapter 4, the separation of field values from linkage information effectively allows the data to be physically stored in several different sort orders simultaneously. When Codd said "symmetry in performance is not to be expected," he was tacitly assuming a direct-image style of implementation, one involving auxiliary structures like those described in Chapter 2. However, as I said in that chapter:

> [Auxiliary structures such as pointer chains and] indexes can be used to impose different orderings on a given file and thus (in a sense) "level the playing field" with respect to different processing sequences; all of those sequences are equally good from a logical point of view. But they certainly aren't equally good from a performance point of view. For example, even if there's a city index, processing suppliers in city name sequence will involve (in effect) random accesses to storage, precisely because the supplier records aren't physically stored in city name sequence but are scattered all over the disk.
>
> *—from Chapter 2*

As we've seen, however, these remarks simply don't apply to the TR data representation.

And now I can address another issue that might possibly have been bothering you. We've seen that the TR model relies heavily on pointers. Now, the CODASYL "network model" [14,25] also relies heavily on pointers—as the "object model" [3,4,28,29] and "hierarchic model" [25,56] both do also, as a matter of fact—and I and many other writers have criticized it vigorously in the past on exactly that score (see, for example, references [10], [21], and [37]). So am I arguing out of both sides of my mouth here? How can TR pointers be good while CODASYL pointers are bad?

Well, in fact there are several differences between TR pointers and CODASYL pointers. The biggest of those differences has to with the question of the *target audience:* Who is the user[1] of the technology supposed to be in each case?

- The target audience for TR is clearly **system programmers,** whose job it is to build DBMSs and other data management systems—for example, data mining tools—on top of a TR implementation. In other words, a TR implementation, viewed in isolation, is not and is not meant to be a complete DBMS as such, and the TR model is not and is not meant to be the application programming interface to such a DBMS.

- By contrast, the target audience for CODASYL is **application programmers,** whose job it is to build application systems—for example, a payroll system—on top of a CODASYL DBMS. In other words, a CODASYL implementation definitely *is* (or was) meant to be a complete DBMS as such,[2] and the "CODASYL model" is (or was) meant to be the application programming interface to such a DBMS.

And, of course, it's well established that system programmers do need to be able to make use of pointers, for all kinds of reasons. On the other hand, it's equally well established that allowing—or, worse, requiring—application programmers to make use of pointers is a very bad idea, again for all kinds of reasons (indeed, this fact is a major justification for the exclusion of pointers from the relational model [40]).

Just as an aside, I simply can't let the foregoing remarks go by without mentioning the distressing fact that, as I write, most of the mainstream SQL vendors (following the current SQL standard [53]) are busily incorporating pointers—pointers, that is, that are visible to the application programmer—into their "model." Reference [40] refers to this "feature" of SQL as a **Great Blunder,** and explains just why it *is* a blunder; for example, it shows among other things that pointers and a good model of type inheritance are fundamentally incompatible. And reference [30] gives numerous additional reasons as to why the relational model should categorically not be extended or "improved" to include pointers.

Back to the comparison with CODASYL. Another big difference between CODASYL pointers and TR pointers is that CODASYL pointers apply at the *record* level, while TR pointers apply at the *field* level. One consequence of this difference is that CODASYL structures are in fact parent/child structures, in the sense of Chapter 2; as a direct consequence, they suffer from all of the problems of such structures identified in that chapter. In particular, therefore, while CODASYL pointers might in principle be used to provide "symmetric exploitation" (although they certainly aren't used that way in practice), they certainly don't provide symmetry in performance, because the records can be physically clustered in at most one way (again, see Chapter 2). The same is not true with TR pointers, as we know.

5.3 TR Operators

Now we come to another issue that I've been ducking slightly so far. I've claimed repeatedly that TR is a model. As such, it must provide some operators to operate on the "objects"—the Field Values Table, etc.—that I've been concentrating on so far (as well as providing those "objects" themselves, of course). So what are the TR operators?

Well, at the most fundamental level, of course, TR certainly includes everything necessary to build, search, access, and maintain tables such as the Field Values Table, including in particular all of the obvious subscripting, assignment, and comparison operators. It also includes operators for allocating and deallocating storage and carrying out other such utility functions. All of these operators are only to be expected.

At a slightly higher level, TR also includes a set of operators that are described in some detail in reference [63]. However, most of those "higher-level" operators are still quite low-level in nature; indeed, most of them are intended for use in the implementation of still higher-level operators that will presumably be used by the system programmers mentioned in the previous section. For that reason, I don't think it's worth getting into details of those lower-level operators here. However, I do want to say a little about the "system programming interface" ones, even though those operators aren't really primitive operators of the TR model as such. (Indeed, reference [63] shows how they could actually be implemented in terms of the lower-level operators that it does describe.)

Let's assume that techniques such as those discussed in Chapter 4—for example, binary searches on columns of the Field Values Table—have already been used to determine that some particular record is of interest. Let me immediately explain what I mean when I say that some record has been "determined to be of interest." To be specific:

- When I say "some particular record," I mean a record of the applicable user file.

- When I say such a record is "of interest," I mean we want the record in question—or the tuple corresponding to that record, rather—to be retrieved, deleted, or updated. (Inserting a new record or tuple is different, of course; we can't sensibly talk about the new record having been "determined to be of interest," because the record doesn't exist yet—at least, not in the database.)

- And when I say techniques have been used "to determine" that the record in question is of interest, I mean it's been determined that some cell [*i,j*] of the Record Reconstruction Table corresponds to some portion of that record; more precisely, it's been determined that cell [*i,j*] of the Record Reconstruction Table contains a pointer that points to a cell in the Field Values Table that contains some portion of that record. In other words, the record in question is that unique record that corresponds to that particular cell [*i,j*] of the Record Reconstruction Table. It's convenient to say, loosely, that the record in question "passes through" that cell [*i,j*] of the Record Reconstruction Table.

  With all of that preamble out of the way, then, the TR operators I want to consider are as follows:

- **Retrieve** the record passing through cell [*i,j*] of the Record Reconstruction Table.

- **Delete** the record passing through cell [*i,j*] of the Record Reconstruction Table.

- **Update** the record passing through cell [*i,j*] of the Record Reconstruction Table.

- **Insert** a new record.

Of these operators, **retrieve** has effectively been discussed at length already in Chapter 4—it's essentially just the business of record reconstruction as described in that chapter (in Section 4.4 in particular). The other three operators are discussed in detail in the next chapter.

One last remark to close the present section: If you happen to be familiar with traditional approaches to implementing the relational model, you might have been expecting to see certain other operators mentioned in the discussion above. For example, the System R prototype [1] consisted of a frontend called the Relational Data System (RDS) and a backend called the Relational Storage System (RSS);[3] the RDS translated user requests—SQL statements, in other words—into RSS operations, and those RSS operations performed such functions as searching indexes, committing and rolling back transactions, and so forth. And those RSS operators included many things that have no direct counterpart in the TR model at all. Some of those operators (for example, those to do with indexes) are omitted from TR because TR simply has no need of them. However, others (for example, COMMIT and ROLLBACK) are omitted because such functionality is meant to be provided above the TR interface. (Indeed, the RSS was really an entire multiuser DBMS in its own right, albeit one whose user interface was rather low-level. By contrast, TR—or a TR implementation, rather—is *not* a complete DBMS in its own right; rather, it's meant among other things to serve as the storage manager component for such a DBMS.)

5.4 Building the Record Reconstruction Table: An Alternative Approach

In the introduction to Chapter 4, I said I'd occasionally make some mention of alternative implementation schemes for certain aspects of the TR model. In keeping with that promise, I'd now like to take a look at an alternative way of building the Record Reconstruction Table. *Note:* It might help to repeat the point from Chapter 4 that the Record Reconstruction Table is built directly from the file (the Field Values Table isn't involved in the process at all).

Now, you might recall that in Chapter 4, Section 4.5, I showed how we could use the Permutation Table instead of the Record Reconstruction Table in order to perform the record reconstruction process. However, I also said it wouldn't be very efficient to use the Permutation Table in that way, because we'd have to do sequential searches on the columns of that table in order to find the record numbers (that's why we replaced the Permutation Table by the Record Reconstruction Table in the first place). The trouble is, though, the algorithm for *building* the Record Reconstruction Table from the Permutation Table still involves doing those same sequential searches—admittedly only when the Record Reconstruction Table is built, not every time it's used, but those searches still represent overhead, and it would be nice to eliminate that overhead if we can.

It turns out we can improve matters by exploiting the **inverses** of the permutations in the Permutation Table. Consider once again the original Permutation Table from Chapter 4, Section 4.5 (see Fig. 5.1). As you can see from that table, the S# permutation (for example) is the sequence

```
4, 3, 5, 1, 2
```

The meaning, to remind you, is that if the records of the original file (see Fig. 4.1 in Chapter 4) are sorted into ascending S# order, record *4* will appear first, record *3* will appear second, and so on. And the inverse of this permutation is the sequence

```
4, 5, 2, 1, 3
```

This inverse permutation is that unique permutation that, if applied to the original sequence *4, 3, 5, 1, 2,* will produce the sequence *1, 2, 3, 4, 5*. (If *SEQ* is the original sequence *4, 3, 5, 1, 2,* then the fourth entry in *SEQ* is *1,* the fifth is *2,* the second is *3,* and so on.)

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

Fig. 5.1: Permutation Table for the suppliers file of Fig. 4.1

More generally, if we think of any given permutation as a vector *V,* then the inverse permutation *V′* can be obtained in accordance with the simple rule that if $V[i] = i'$, then $V'[i'] = i$. Applying this rule to each of the permutations in our given Permutation Table, we obtain the **Inverse Permutation Table** shown in Fig. 5.2. (**Exercise 4:** Check that the table is correct.)

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

Fig. 5.2: Inverse Permutation Table corresponding to Fig. 5.1

We can now use the Inverse Permutation Table to build the Record Reconstruction Table without doing any sequential searches. For example, the first (S#) column of the Record Reconstruction Table can be built as follows:

> 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, ..., 5* yields the entire S# column of the Record Reconstruction Table. The other columns are built analogously. **Exercise 5:** Check that the foregoing algorithm, when applied to the given Inverse Permutation Table, does indeed produce the Record Reconstruction Table shown in Fig. 4.3 in Chapter 4. (Doing this exercise should convince you that this algorithm is much easier to apply than the one given in Chapter 4; it should also make you understand why the algorithm works, if you haven't figured it out already. In future chapters, when I need to build a Record Reconstruction Table, I'll use this new algorithm.)

5.5 Record Reconstruction Revisited

Like the previous section, this one too is concerned with a possible implementation alternative. In that previous section, the Permutation and Inverse Permutation Tables served as purely temporary structures, used in building the Record Reconstruction Table but then discarded. However, it would be possible not to discard them after all, but rather to use them together as a replacement for the Record Reconstruction Table. For example, consider the following SQL query (a projection of a restriction):

```
SELECT S.S#, S.STATUS
FROM S
WHERE S.CITY = 'London' ;
```

We can implement this query as follows:

- *Step 1:* Use a binary search to find the London entries in the Field Values Table (see Fig. 4.2 in Chapter 4) and extract the corresponding row numbers. In the example, this step yields the row numbers *2* and *3*.

- *Step 2:* Use those row numbers to look up entries in the CITY column of the Permutation Table (see Fig. 5.1). This step yields the corresponding *record* numbers, *1* and *4*.

- *Step 3:* Use those record numbers as *row* numbers to look up entries in the S# column of the Inverse Permutation Table (see Fig. 5.2). This step yields the row numbers *4* and *1,* and these values can be used to access the corresponding S# values in the Field Values Table, S1 and S4.

- *Step 4:* Likewise, use the record numbers from Step 2 to look up entries in the STATUS column of the Inverse Permutation Table. This step yields the row numbers *2* and *3,* and these values can be used to access the corresponding status values in the Field Values Table, which are both 20, as it happens. Execution of the query is now complete.

Comparing the foregoing with what we would have had to have done using the Record Reconstruction Table, we can see that one advantage is that we don't have to chase pointers through columns that aren't involved in the query (a fact that could be useful in implementing projection operations, for example). On the other hand, the Permutation and Inverse Permutation Tables together occupy twice as much space as the Record Reconstruction Table does.

Having said all of the above, let me now say that for definiteness I'll assume an implementation from this point forward that does do reconstruction via the Record Reconstruction Table, not via the Permutation and Inverse Permutation Tables (barring explicit statements to the contrary). In other words, I'll assume the Permutation and Inverse Permutation Tables aren't kept around at run time.

5.6 Pointers are Field Value Surrogates

Consider Fig. 5.3, a repeat of Fig. 4.3 from Chapter 4, which shows the Field Values Table for the suppliers file of Fig. 4.1 together with a corresponding Record Reconstruction Table; more specifically, consider the Field Values Table in that figure. Clearly, the position—that is to say, the row number—of any given field value within its containing column in that table serves as a unique encoding, or **surrogate,** for the value in question (in other words, the table provides an *encoding mechanism* for its values). For example, consider the CITY column, which contains, in sequence, the city names Athens, London, London, Paris, and Paris; clearly, the corresponding row numbers *1, 2, 3, 4, 5* can be regarded as surrogates for those values (in sequence as indicated).[4] What's more, those very same row numbers can also be regarded as surrogates for the supplier numbers S1, S2, S3, S4, and S5; the names Adams, Blake, Clark, Jones, and Smith; and the status values 10, 20, 20, 30, and 30 (in sequence as indicated in every case).

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	S1	Adams	10	Athens
2	S2	Blake	20	London
3	S3	Clark	20	London
4	S4	Jones	30	Paris
5	S5	Smith	30	Paris

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

Fig. 5.3: Field Values Table of Fig. 4.2 and a corresponding Record Reconstruction Table

It follows from the foregoing that the Record Reconstruction Table can be regarded as containing such field value surrogates (and likewise for the Permutation and Inverse Permutation Tables, of course). For example, the STATUS column in the Record Reconstruction Table of Fig. 5.3 contains, in sequence, the row numbers *4, 2, 3, 1, 5*. These row numbers are surrogates for CITY values (not STATUS values); they stand for the values Paris, London, London, Athens, and Paris, respectively, and this sequence is the sequence in which the city names will appear if we ask to see suppliers in status sequence, thus:

```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY
FROM S
ORDER BY STATUS ;
```

So now we know that row numbers serve as surrogates for field values, and the Record Reconstruction Table in particular contains such surrogates. This alternative perspective is occasionally useful, as we'll see in Part III of this book. Now, in the TR model as I've described it so far (and indeed as I'll continue to describe it throughout the remainder of this book), the surrogates in question are *always* row numbers. But other surrogate schemes are possible and could be useful in different implementation environments—and so such alternative schemes are yet another illustration of the fact that the TR model is capable of many different concrete implementations. Further details are beyond the scope of this book.

5.7 The Field Values Table is a Directory

In this section, I want to consider (briefly) another alternative perspective that can also be helpful on occasion. Consider the Field Values Table in Fig. 5.3 once again; more specifically, consider the CITY column in that table. Let *c* be a value (city name) in that column, and let the containing cell *C* be cell [*i,4*] (meaning city *c* has surrogate *i*). Then that subscript [*i,4*] identifies the unique cell, *C′* say, in the Record Reconstruction Table that corresponds to cell *C* in the Field Values Table. That cell *C′* in turn is part of a zigzag that allows a record containing the CITY value *c* to be reconstructed.

All of the foregoing should really be familiar to you by now, and I mention it here mainly by way of review. However, let me now point out something that I deliberately haven't mentioned before: namely, that each column of the Field Values Table effectively serves as a kind of **directory**—an index, almost!—to the Record Reconstruction Table and thence, eventually, to the corresponding records. For example, consider the city name Athens, which appears in cell [*1,4*] of the Field Values Table. Following the zigzag through cell [*1,4*] of the Record Reconstruction Table, we obtain the record :

	S#	SNAME	STATUS	CITY
2	S5	Adams	30	Athens

(More precisely, we obtain a version of this record in which the left-to-right field ordering is CITY, then S#, then SNAME, then STATUS.)

Please note carefully, however, that though we might indeed say that each column of the Field Values Table is "almost an index," it certainly isn't an index in the conventional sense of that term. Perhaps a better way to put it would be to say that the column in question—together with the Record Reconstruction Table, which is certainly needed too—*provides the functionality of* an index. That is, the column in question and the Record Reconstruction Table together provide indexing functionality (both direct- and sequential-access functionality) on the user file on the basis of values of the corresponding field.

5.8 Miscellaneous Implementation Alternatives

I'd like to close this chapter by briefly mentioning a few miscellaneous points regarding alternative implementation possibilities for various other TR constructs.

- I've been talking so far as if the linkage information that ties together the field values for a given record must be implemented as a pointer ring or zigzag specifically. But other possibilities exist. For example, we could replace each such ring (within any given Record Reconstruction Table) by two subrings that are connected by means of some common "bridging" column. In the case of suppliers, for example, we might have one subring connecting S# and CITY and another connecting S#, SNAME, and STATUS (column S# being the bridging column, in this particular example). Such an arrangement would be advantageous if projection over S# and CITY is a frequently requested operation—in other words, if queries of the form SELECT S.S#, S.CITY FROM S are common, in SQL terms. What's more, the pointers in such rings or subrings could be either one-way (as I've been assuming so far) or two-way. Other options are also available; one such will turn out to be important in connection with disk-based implementations, and I'll discuss it in detail in Chapter 14.

- I've also been talking so far as if every column in the Field Values Table *has* to be maintained in sorted order. In practice, however, such is not the case; there might well be some columns for which such sorting is just not worth the overhead. An example might be a text column in which the entries are natural-language comments.

- Furthermore, those columns that are sorted don't all have to be sorted in the same way. In our examples, I've shown all columns sorted in ascending sequence. However, it might be better to keep some columns in descending sequence instead; and in the case of columns defined over a user-defined data type, the sort order might be defined in terms of a user-defined "<" operator [40] or in some other way (see Chapter 15, Section 15.5). As reference [63] puts it: "A sort order should be chosen based on its usefulness for display or retrieval purposes *in actual applications*" (my italics).

- Since there's a one-to-one correspondence between the cells of the Field Values Table and the cells of the Record Reconstruction Table, the two tables could if desired be physically collapsed into one. *Note:* This option will cease to be available, however, if the refinements to be discussed in Chapters 8 and 9 are adopted (which in practice they probably will be).

- The same is true, and is perhaps a more sensible proposition, in the case of the Permutation and Inverse Permutation Tables (assuming, of course, that those two tables are indeed both kept around at run time, as we saw in Section 5.5 was a possibility—but I remind you that I'm not going to make that assumption).

- The Permutation and Inverse Permutation Tables differ from the Field Values Table and the Record Reconstruction Table in that there's no reason why their left-to-right column order need be the same as the left-to-right field order of the corresponding file. As a consequence, their left-to-right column orders can be arbitrarily rearranged. *Note:* Actually, the same is effectively true of the Field Values Table and the Record Reconstruction Table as well, inasmuch as such ordering has no meaning at the relational level. In practice, it's probably a good idea to choose a left-to-right column order for those tables such that, if attributes *A* and *B* often appear together in user-level queries (especially if WHERE clauses often include conditional expressions of the form WHERE *A* = ... AND *B* = ...), then the columns corresponding to those attributes are adjacent in the two tables. In particular, these remarks are true of attributes that are components of the same key; that is, columns that correspond to attributes in a multiattribute key should generally be adjacent. See also the further remarks on this topic at the very end of Chapter 8.

Endnotes

1. The word *user* is always a little ambiguous. I don't mean it here in the sense of the Chapter 3 "user level," I mean whoever is the direct, immediate user of the technology in question.
2. It's true that, with hindsight, we might regard CODASYL (like TR) not as a model for "a complete DBMS as such" but rather as an implementation technology, even though such was not the original intent. However, CODASYL is fundamentally unsuited to that role for the kinds of reasons discussed in Chapter 2, as well as many others.
3. The backend name was rather inappropriate, because relations aren't a storage-level concept at all. In any case, the name was subsequently changed for political reasons to *Research* Storage System.
4. Note that the very same field value can have two or more distinct surrogates; for example, the value London has surrogates *2* and *3*. If the refinements to be discussed in Chapters 8 and 9 are adopted, however, surrogates will be unique, in the sense that every field value will have just one of them.