Part III: Disk-Based Implementation

11 General Disk Considerations

11.1 Introduction

So far in this book I've tacitly assumed—for the most part, at any rate—that the entire database is in main memory at run time. Now I need to consider what happens if that assumption is invalid (which will usually be the case in practice, of course). Such is the purpose of this part of the book.

Now, I claimed in Chapter 1 that divide-and-conquer is always a good pedagogical approach, and I appealed to that fact as my justification for largely ignoring disk-specific issues prior to this point. But there's more to it than mere pedagogy; the fact is, divide-and-conquer can be a good approach to design problems as well. The reason is that, in general, making simplifying assumptions and sticking with them for as long as possible can serve to clarify issues that might otherwise remain comparatively opaque. By way of one example, it was the initial assumption of a static, read-only database that led to the highly original TR approach to updating described in Chapter 6. By way of another, it was the initial assumption that everything could be kept in main memory that led to the (again highly original) logical data transforms described throughout the chapters in Part II.

Recall now that in Chapter 8 I characterized the TR data representation as **permutations and histograms;** that is, the logical data transforms just mentioned can be thought of as transforms that map a direct-image version of the data into such permutations and histograms. So when we get to a disk-based implementation, the question becomes: How can we transform those permutations and histograms still further in order to get the best possible representation of them in terms of storage structures on the disk? In other words, what *physical* transforms should we now carry out on the already logically transformed data? Observe how divide-and-conquer comes into play again; we don't even begin to think about looking for a good physical transform until we've carried out a good logical transform first. That's because (as the history of direct-image implementations strongly suggests) it's hard to find an optimized disk representation if we don't have an optimized main-memory representation to start with.

What I want to do, then, in the rest of this chapter and in the next three, is describe a particular set of physical transforms that can be used in TR in order to achieve "main-memory performance off the disk" (to put matters catchily, if not all that precisely). The present chapter considers the problem in general terms; the next three chapters then go on to discuss certain highly TR-specific approaches to that general problem.

Please note that this part of the book, even more than Part II, is not meant to be exhaustive. Rather, it's meant to give you some idea of what's involved in producing a good disk-based implementation of the TR model, without getting too deeply into numerous variations and alternative possibilities. My major aim is to convince you that a good disk-based TR implementation is indeed feasible, and what's more is likely to display some very attractive characteristics (performance characteristics in particular).

I should say too that this part of the book does assume you've read Part II carefully and mastered the key ideas contained therein—probably by doing the exercises as recommended (though you might be glad to hear there aren't any exercises in Part III).

Let me close this section with a couple of points of terminology that I'll be relying on throughout what follows. First, I remind you from Chapter 1 that I use the term *memory,* unqualified, to mean main memory specifically. Second, I'll use the term *memory-resident* to mean that the pertinent data, whatever it might happen to be, has already been brought into memory before we need it at run time.

11.2 What's the Problem?

Clearly, the disk implementation problem in general terms is simply to minimize the time it takes to find the data we want and read it off the disk. So let's briefly review what's involved in that "finding and reading" process. There are two main aspects to consider:

- *Seek time:* This is the time it takes to move the disk read/write head from its current position to the desired block or page. Seek times are measured in milliseconds (msec); they can be anything from 2 to 60 msec, with 6 msec being a good typical figure. By contrast, a typical "seek time" for memory might be 60 *nanoseconds* (nsec); thus, disk access is around 100,000 times, or five orders of magnitude, slower than memory access. The implications are obvious—we clearly want to jump around randomly on disk as little as possible; that is, we want to keep seek activity to a minimum. For otherwise we'll be in a situation in which overall system performance is totally dominated by the time spent doing seeks on the disk.

- *Data rate:* This is the speed at which data can be read off the disk once the read/write head has been positioned to the desired block or page. Data rates are measured in megabytes per second (MB/sec); they can be anything from 4 to 40 MB/sec, with 10 MB/sec being a good typical figure.[1] However, several disk drives can be attached to the same I/O channel, and channel data rates can reach as much as 256 MB/sec; thus, it might be possible by interleaving accesses to different disks to achieve an *effective* data rate across the channel of (say) 160 MB/sec or so. At that rate, if we take the average seek time to be 6 msec as suggested above, then one seek takes about the same amount of time as it takes to read one megabyte off the disk. What's more, it also takes about the same amount of time as it takes to scan one megabyte of data in memory, at least to a first approximation (I'm assuming here, not very realistically, that data is accessed in memory a single byte at a time).

Note: Actually there's a third aspect to the problem of finding and reading disk data, the *latency* or *rotational delay* aspect, which is the time spent waiting for the rotation of the disk to bring the desired block or page under the read/write head. For simplicity I've lumped this aspect in with seek time above.

Let me now elaborate briefly on the implications of considerations such as those above in the case of TR specifically. Observe first that, from the user's perspective, there are basically two general tasks that any DBMS needs to be able to perform (and perform well):

a) Given a particular tuple, find all of its attribute values;
b) Given a particular attribute value, find all of the tuples that contain it.

Now, classical direct-image systems are quite good on the first of these tasks (even on disk), but they're not very good on the second (not even in memory). By contrast, TR is very good on both, at least so long as we limit ourselves to a memory-based implementation:

a) Finding all attribute values for a particular tuple is basically the process of record reconstruction, using the appropriate zigzag in the Record Reconstruction Table;
b) Finding all tuples with a particular attribute value is basically the process of doing a binary search on the appropriate column in the Field Values Table.

But what happens on disk? The algorithms that work so well in memory clearly won't work so well on disk. To be specific, following zigzags and doing binary searches both effectively imply a lot of random jumping around, and disk performance is thus likely to be terrible unless we can come up with some good physical transforms. Such transforms are the subject of the remainder of this chapter.

11.3 Addressing the Problem

In this section I'll offer some general remarks regarding those good physical transforms; in subsequent sections, I'll focus in on some more specific issues and go into more detail. However, I should warn you that those subsequent sections do unavoidably involve a certain amount of cross-referencing among themselves, because the techniques I'll be describing aren't all independent of one another. But first things first.

First of all, then, we'd obviously like to have as much of the database as possible resident in memory at run time. One important technique for achieving this goal is *data compression,* which reduces not only the amount of space the data requires on the disk but also, and more importantly, the amount of space it requires in memory. (Of course, it also reduces the amount of time it takes to find and read the data, and so it's also relevant to the discussion of seek and read times below.) Sections 11.4 and 11.5 discuss specific compression techniques that apply to the Field Values Table and the Record Reconstruction Table, respectively.

Second, when we do have to access the disk because the data we need isn't memory-resident, we'd clearly like to minimize the amount of seeking we have to do. Several techniques are available to help here:

- *Large pages:* Page sizes in today's commercial DBMS products typically range from a minimum of one kilobyte, or even less, to a maximum of perhaps 64 kilobytes (1KB-64KB). As a consequence, the ratio of seek time to read time—"the seek-to-read ratio"—is usually quite high, ranging from around 1,000:1 for 1KB pages to around 16:1 for 64KB pages, if seek time is 6 msec and data rate is 160 MB/sec. In other words, most disk access time in today's systems is typically taken up in seek activity. Using larger pages of (say) one megabyte each will clearly reduce the seek-to-read ratio to something much more reasonable (approximately 1:1 for 1MB pages).

Note: The foregoing analysis does tacitly assume that everything in the page in question is "useful," in the sense that reading the whole page doesn't mean bringing into memory—and taking the time to bring into memory—a lot of data that's irrelevant to the purpose at hand. If we're in a complex-query environment (a data warehouse system, for example), then this assumption isn't too unrealistic. By contrast, if we're in an environment in which the queries are comparatively simple—an OLTP system, for example[2]—then the design tradeoffs might be different (in particular, smaller pages might be desirable). In what follows, I'll tend to assume the complex-query environment, where it makes any difference.

- *Streaming:* Next, we try to arrange matters such that if page *p2* is needed immediately after page *p1* at run time, then page *p2* immediately follows page *p1* on the disk. In this way, moving the read/write head from page *p1* to page *p2* involves little or no seeking, and data can be "streamed" off the disk into memory at a data rate close to the theoretical maximum. *Note:* The next item below, column-wise storage, is highly pertinent to this idea of streaming.

- *Column-wise storage:* Both the Field Values Table and the Record Reconstruction Table are accessed column-wise, at least initially (the Field Values Table when doing binary searches and the Record Reconstruction Table when starting to chase successive zigzags). For this reason, it's a good idea to store both tables column-wise on the disk, so that data items that are logically required together are physically close together on the disk. (In case it isn't clear what I mean when I say the tables are stored column-wise on the disk, let me explain briefly. In essence, what I mean is that column *1* is stored as a set of consecutive pages on the disk, then column *2* is stored as an immediately following set of consecutive pages, and so on.)

Note: The idea of storing the data column-wise is not so important (though certainly not *un*important) in the case of the Field Values Table, because that table will almost certainly be in memory at run time anyway (see Section 11.4). However, it's very important in the case of the Record Reconstruction Table (see Section 11.5), and becomes even more so if the techniques of Chapter 14 are adopted.

- *Banding:* We've seen that zigzags don't work so well if they mean jumping all over the disk. Banding is a solution to this problem; it's discussed briefly in Section 11.6 and in more detail in Chapter 13.

- *Using stars instead of zigzags:* Another solution to the problem of "zigzagging all over the disk" is to replace the zigzags by **stars**. Stars are also discussed briefly in Section 11.6 and in considerably more detail in Chapter 14.

- *Controlled redundancy:* Both banding and stars have the property that they can undermine the objective of symmetric performance (see Chapter 5, Section 5.2). We can address this problem by introducing a degree of controlled redundancy into the storage representations. Controlled redundancy is also discussed briefly in Section 11.6 and in more detail in Chapters 13 and 14.

11.4 Compressing the Field Values Table

Recall from Chapter 4 that the Field Values Table is the only TR table that contains user data as such. Recall too from Chapter 6 that although we refer to it as a table, it isn't physically stored as a table; instead, as noted in the previous section, it's stored column-wise, or in other words as a set of *vectors* (typically), one such vector for each column. And a variety of techniques, some primarily logical in nature and others more physical, are available for compressing those vectors. Let's take a closer look.

Logical Compression

By the term *logical compression,* I mean techniques that transform the data before it even reaches the disk, as it were. All of the techniques discussed in Chapters 8 and 9 fall into this category, including in particular the fundamental ones of condensing and merging columns. A variety of other possibilities also exist, including:

- *Mapping combinations of fields to a single column* (see Chapter 8, Section 8.5). This technique allows two or more vectors to be replaced by one whose length is less—often much less—than the sum of the lengths of the original ones.

- *Breaking fields into subfields,* also known as *subfield encoding* (see below). This technique allows one long vector to be replaced by two or more much shorter ones.

- And several others (again, see Chapter 8, Section 8.5).

Note: File factoring is another logical compression technique that applies to the Field Values Table. However, that technique, though it does indeed have the effect of compressing the Field Values Table, usually has a much more dramatic effect on the Record Reconstruction Table, and it's this latter compression that's the real point. For that reason, I'll defer further discussion of such factoring to the next section.

By the way, regarding column condensing specifically, I'd like to remind you that the amount of compression achievable can be dramatic—recall the example from Chapter 8 of a relation representing drivers' licenses, where the compression ratio was quite literally of the order of a million or so to one. On the other hand, column condensing won't do much for a field whose values are unique or almost unique; for such a field, subfield encoding (see below) or the techniques of Chapter 12 are likely to be more appropriate.

Let me now explain the concept I've mentioned a couple of times already, **subfield encoding**. Subfield encoding represents an additional refinement on the basic idea of condensed columns. The objective is to reduce overall space requirements still further, by breaking a given field into "subfields," each of which has far fewer distinct values than does the original field overall. For example, suppose we have a relation containing one tuple for each phone number in the United States, giving the names of persons or organizations reachable via those phone numbers. Assume for definiteness that there are 200 million tuples in the relation, so the number of distinct values of the PHONE# attribute (equivalently, the number of distinct values of the PHONE# field in the file corresponding to the relation) is 200 million.[3] Assume too for simplicity that the PHONE# field is ten bytes wide, one byte for each digit. Then column PHONE# of the Field Values Table will require 2,000 megabytes, and column PHONE# of the Record Reconstruction Table will require 1,400 megabytes (two pointers per cell, each pointer requiring 28 bits), for a total of 3,400 megabytes. *Note:* I'm relying here and throughout my discussion of this example that pointers are only as big as they logically need to be. This concept is discussed in detail in Section 11.5.

Note, however, that even though there are 200 million different phone numbers, there certainly aren't 200 million different area codes—in fact, there are only a few hundred. For definiteness again, let's assume there are just 250 area codes, with an average of 800,000 phone numbers within each one (250 * 800,000 = 200 million). Let's assume further that there are just 200 different prefixes or "exchanges" within each area code (first three digits of the phone number) and 4,000 different numbers within each area code and prefix (last four digits). So let's break the PHONE# field down into three subfields: AREA_CODE (three bytes), PREFIX (three bytes), and REST (four bytes). Here's what happens:

- Column AREA_CODE requires just 750 bytes (plus space for row ranges) in the Field Values Table, which we can ignore; 200 megabytes in the Record Reconstruction Table for pointers into the Field Values Table (each such pointer will be eight bits); and 700 megabytes in the Record Reconstruction Table for "next cell" pointers.

- Column PREFIX requires just 600 bytes (plus space for row ranges) in the Field Values Table, which we can ignore; 200 megabytes in the Record Reconstruction Table for pointers into the Field Values Table (each such pointer will again be eight bits); and 700 megabytes for "next cell" pointers.

- Column REST requires 16,000 bytes (plus space for row ranges) in the Field Values Table, which once again we can ignore; 300 megabytes in the Record Reconstruction Table for pointers into the Field Values Table (each such pointer will be twelve bits); and 700 megabytes for "next cell" pointers.

The grand total is approximately 2,800 megabytes, or a saving of roughly 17.6 percent compared with the original figure of 3,400 megabytes. What's more, this saving has been achieved even though the relevant portions of the Record Reconstruction Table have actually doubled in size. The point is, the relevant portions of the Field Values Table have effectively been reduced to *zero* size.

Note: Suppose we decide not to include pointers from the Record Reconstruction Table into the Field Values Table (after all, such inclusion was characterized as an "optional extra" in Chapter 8, at the end of Section 8.3). Then the grand totals of 3,400 megabytes and 2,800 megabytes reduce to 2,800 megabytes and 2,100 megabytes, respectively, and the saving becomes 25 percent.

Physical Compression

I'm using the term *physical compression* to mean techniques that effectively treat the output from the logical compressions discussed above—condensing, merging, and so on—simply as a set of very long bit strings and compress those bit strings "mechanically," without paying any attention to what those bit strings might represent. Under this general heading, there's just one point I want to discuss in any detail: namely, the fact that, in TR, such bit strings are always stored **bit-aligned** on the disk, instead of being aligned on (say) a fullword or four-byte boundary. By way of example, suppose we have a field *F* of type INTEGER; assume for the sake of the example that type INTEGER denotes integers in the range -2^{31} to $2^{31}-1$. Suppose, however, that field *F* actually holds values in the range 0 to 99 only. In a conventional system, each *F* value will still require four bytes of storage. In TR, by contrast, it will require only *seven bits*—a saving of over 78 percent (see Fig. 11.1).

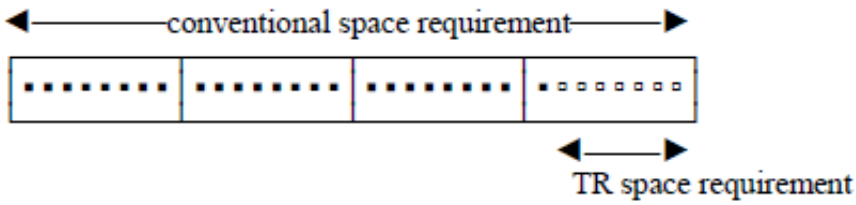

Fig. 11.1: Bit alignment (example)

In addition to the foregoing, conventional physical compression techniques might also be used. *Front compression* is an example (see Chapter 2); since the field values in any given Field Values Table column are sorted into sequence, front compression can be applied directly if desired (though I should point out that such compression will complicate the binary search and record reconstruction tasks somewhat). What's more, the row ranges in any given Field Values Table column are sorted too (more precisely, they are in ascending sequence by either the range begin points or the range end points), and they can therefore be compressed as well.

Net Effect

The net effect of all of the above is that the Field Values Table is always memory-resident, at least to a first approximation. Startling though this claim might appear at first sight, on reflection it should seem plausible enough; after all, how many distinct attribute values do real databases actually contain? (And note that this rhetorical question appeals only to the idea of logical compression. Physical compression can only make the situation better.) *Note:* Even in those rare cases when the Field Values Table is *not* 100 percent memory-resident, there are still efficient ways of accessing it on the disk. Details of what's involved in such cases are beyond the scope of this chapter, however.

So—assuming that the Field Values Table is indeed memory-resident—we've now solved one of our two original problems: All binary searches on columns of that table will be done in memory, not on disk.

There are a few further points I want to make to close this section.

- First, as we'll see in the next section, reducing the size of the Field Values Table reduces the number of bits needed to represent pointers into that table as well. In other words, reducing the size of the Field Values Tables reduces the size of the Record Reconstruction Table as well.

- Second, in computing the size of a given Field Values Table, we ought by rights to take the space required by the row ranges into account as well (even though it's likely that those row ranges will be physically stored separately from the field values per se). Now, if Field Values Table column *C* contains *N* values, then the corresponding row ranges will require a total of *N* log *N* bits—probably much less space than the *N* values themselves require. Perhaps we might say to a first approximation that row ranges cause the overall size of the Field Values Table to double, though I think their effect is likely to be much less than that in practice. But it's simpler—given that I'm usually not trying in this book to do precise analyses—just to assume that if the Field Values Table without row ranges is small and can fit into memory, then the Field Values Table with row ranges is also small and can fit into memory too. In other words, I'm going to ignore the space required for row ranges from this point forward. I don't believe this simplifying assumption has any material effect on any of the arguments to come.

- Third, I've said the Field Values Table is stored column-wise. Now, in Chapter 4 I mentioned the fact that certain other systems, both prototypes and commercial products, "store the data attribute-wise." The two notions aren't directly comparable, though. To be specific, in TR we're not really talking about storing some attribute of some user-level relation at all; rather, we're talking about storing some condensed, merged, and possibly otherwise transformed column of the Field Values Table,[4] and as we've seen there's no direct correlation (in general) between a user-level attribute and a Field Values Table column.

11.5 Compressing the Record Reconstruction Table

The subfield encoding example in the previous section illustrates a point that you might or might not have realized for yourself, but is in any case worth calling out explicitly. To be specific: *In any real database, the amount of space required for the Field Values Table is likely to be negligible compared to the space required for the Record Reconstruction Table*. In other words, the Record Reconstruction Table in any real database is likely to be orders of magnitude bigger than the Field Values Table, and so we'd definitely like to find ways to compress it if we can. The trouble is, the Record Reconstruction Table contains what are in effect *permutations,* and permutational data is notoriously hard to compress. Even so, there are some useful things we can do ... This time I'd like to discuss physical compression techniques first.

Physical Compression

My first point has to with **TR pointer size**. As I explained in Chapter 2, the pointers we're talking about, though conceptually addresses, certainly aren't physical addresses, neither on disk nor in memory. In TR, in fact, they aren't even of constant size—they aren't all 32 bits in length, for example. Rather, the pointers within any given Record Reconstruction Table are *just as big as they need to be*. For example, given the Record Reconstruction Table of Fig. 11.2 (a copy of the Record Reconstruction Table from Fig. 4.3 in Chapter 4), it's clear that there are only five different pointer values, and three bits are thus sufficient to represent any of them.

	1	2	3	4	
	S#	SNAME	STATUS	CITY	1
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. 11.2: Record Reconstruction Table for the suppliers file of Fig. 4.1

Recall now that (of course) the Field Values Table is condensed. If we expand the Record Reconstruction Table of Fig. 11.2 to include direct pointers into (say) the CITY column of the condensed Field Values Table, then those pointers will require only two bits, not three, because there are only three distinct CITY values and not five. More realistically, suppose there were 100,000 rows in the uncondensed Field Values Table but only 20 distinct CITY values; then the pointers we're talking about would require only five bits instead of the 17 they would otherwise require (2^{17} = 131,072).[5] In general, the space saving could be considerable (over 70 percent, in this particular example).

Like other data, pointers in TR are bit-aligned on the disk (in particular, therefore, they aren't necessarily even byte-aligned, let alone word-aligned).

While I'm on the subject of pointer size, let me explain something else that might possibly have been bothering you. Suppose we're using an overflow structure to hold newly inserted values as described in Chapter 6 (Section 6.5). Then pointers in that overflow structure don't necessarily have to be the same size as their counterparts in the main database. Suppose, for example, that a given field in the main database contains exactly 128 distinct values, so that associated pointers are just seven bits, and then a new 129th value is inserted (implying that seven bits are no longer adequate). Then pointers in the overflow structure might have to be eight bits—or not, as the case may be—but pointers in the main database won't have to change in size until such time as the merging process is done (that is, until the overflow structure is merged in with the main database, as described in Chapter 6, Section 6.5).

Back to physical compression techniques for the Record Reconstruction Table. Here are some relevant considerations.

- First of all, we don't have to include those direct pointers from the Record Reconstruction Table into the Field Values Table anyway; they're there (as explained in Chapter 8) merely to speed up the record reconstruction process, and they aren't logically necessary. So we could delete them if desired (and we probably would, on the disk).

- Second, even if we do include those direct Field Values Table pointers after all, we can at least apply (for example) front compression to them, since their values within any given Record Reconstruction Table column are at least guaranteed to be in ascending sequence.

- Third, we've seen that compressing the Field Values Table has the desirable side-effect of reducing the size of those direct Field Values Table pointers anyway.

- Fourth, suppose the Record Reconstruction Table is a "cyclic" one (refer to Chapter 7, Section 7.5, for an explanation of this term). Then, within any given column of that table, the zigzag pointers corresponding to a given field value within the Field Values Table are also guaranteed to be in ascending sequence; they can therefore also be compressed.

However, despite all of the above, the fact remains that the Record Reconstruction Table is still likely to be quite large in practice. By way of an example, suppose we start with a user-level relation of ten attributes and 200 million tuples. Suppose we decide not to include direct pointers from the Record Reconstruction Table into the Field Values Table; for simplicity, however, suppose also that the table isn't a cyclic one, and so the compression techniques applicable to such tables aren't available. Then we're going to need a total of two billion pointers of 28 bits each, for a grand total of seven billion bytes. What can we do about this problem?

Logical Compression

Before I attempt to offer an answer to the question just posed, let me first say a little more about the problem of zigzags on the disk.

Now, I've already explained that the Record Reconstruction Table is stored column-wise on the disk. By way of example, consider Figs. 11.3 and 11.4, which show the Field Values Table for the parts relation from Chapter 8 and a corresponding Record Reconstruction Table (the figures are identical to Figs. 8.6 and 8.4, respectively, in Chapter 8). To keep the example simple, I've omitted the direct pointers from the Record Reconstruction Table into the Field Values Table.

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

Fig. 11.3: Field Values Table for parts

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

Fig. 11.4: Record Reconstruction Table for parts

Now consider the query "Get all red parts." In order to implement this query, the system will do an in-memory binary search on the COLOR column of the Field Values Table and will discover that the corresponding row range is [*4*:*6*]. Then it'll go to the COLOR column of the Record Reconstruction Table and chase three zigzags, beginning at cells [*4,3*], [*5,3*], and [*6,3*], respectively. (Recall that in the subscript expression [*i,j*], *i* is a row number and *j* is a column number.)

From this example, we can see that we certainly want to store cells [*4,3*], [*5,3*], and [*6,3*] contiguously in storage; that is, column-wise storage for column CITY of the Record Reconstruction Table is obviously desirable. And, of course, analogous arguments show that column-wise storage is desirable for every column of that table.

But the problem is, even if (in terms of our example) we *start* chasing the zigzags from contiguous locations, we very quickly find ourselves performing essentially random lookups "all over the disk." Indeed, the three zigzags actually look like this in the example:

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

In other words, although the starting points are physically contiguous, the zigzags quickly splay out to what are essentially random positions within the Record Reconstruction Table—effectively implying a separate seek and read operation for every point after the starting point in each zigzag, if the zigzag in question isn't in memory at run time.

So reducing the size of the Record Reconstruction Table (so that the zigzags can be in memory at run time after all) is highly desirable. Such is the aim of **file factoring**. File factoring can be regarded as a highly effective logical compression technique—so effective, in fact, that it's likely to mean that large portions, at least, of the Record Reconstruction Table will be memory-resident after all in any real database. And if we can achieve this desirable goal, we'll have solved the other of our two original problems: All zigzagging through that table will be done in memory, not on disk.

File factoring is described in detail in the next chapter.

11.6 Minimizing Seeks

In this section I want to consider, very briefly, what happens if the techniques described in previous sections aren't sufficient to get everything into memory. If that's the case, then we'll still have to perform some degree of disk access at run time, and (as we saw in Section 11.3) we clearly want to keep the amount of seek activity involved in that process to a minimum.

Now, I listed a variety of techniques in Section 11.3 for reducing run-time seeking. Just to remind you, here's that list again:

- Large page sizes
- Streaming data off the disk
- Storing data column-wise
- Banding
- Using stars instead of zigzags
- Controlled redundancy

Of these six items, I've said as much as I'm going to say regarding the first three. The remainder of this section presents a brief overview of the rest.

Banding

For simplicity, I've tended to talk in this book in terms of "the" Field Values Table and "the" Record Reconstruction Table, as if there were just one of each. In practice, of course, there'll be not one but many of each; loosely speaking, there'll be one of each for each user-level relation—though as we already know, in the case of the Field Values Table(s) in particular, the picture is complicated somewhat by the possibility (or likelihood, rather) of column merging and certain other features of the TR model.[6] However, there'll certainly be many Record Reconstruction Tables, in general, and *banding* will lead to more.

Banding is an attack on the problem of zigzags that splay out all over the disk. The basic idea is to split the original file (conceptually) into a set of horizontal *bands,*[7] and then to treat each such band as a file in its own right, with its own TR-level representation. In other words, each band will have its own Field Values Table and its own Record Reconstruction Table—implying in particular that zigzags within any given Record Reconstruction Table will be wholly contained within the relevant band. Band size is chosen such that any given band will fit entirely into memory at run time, and bands are laid out on the disk in such a way as to facilitate streaming data off the disk. See Chapter 13 for further discussion.

Using Stars Instead of Zigzags

Like banding, *stars* too are an attack on the problem of zigzags that splay out all over the disk. Recall from Chapter 5, Section 5.8, that the linkage information that ties together the field values for a given record doesn't have to be implemented as a zigzag specifically—other possibilities exist, and *stars* are one such. Basically, stars are functionally equivalent to zigzags but have different performance characteristics. In particular, they avoid the splay problem and thus reduce the amount of random seeking required. See Chapter 14 for further discussion.

Controlled Redundancy

Banding and stars both have the property that access based on one particular field, the so-called *characteristic* (or *core*) field, will perform better than access based on any other; that is, access via any field other than the characteristic one will involve more seeks than access via the characteristic one. In other words, as noted in Section 11.3, symmetry of performance will be lost (see Chapter 5, Section 5.2, for a discussion of this notion). We can address this problem by introducing a degree of controlled redundancy into the storage structures. See Chapters 13 and 14 for further discussion.

Endnotes

1. The term *megabyte* is sometimes defined to mean exactly one million bytes, sometimes 2^{20} = 1,048,576 bytes. Similarly, the term *kilobyte* is sometimes defined to mean exactly one thousand bytes, sometimes 2^{10} = 1,024 bytes. The differences aren't significant for our purposes.
2. OLTP = online transaction processing.
3. I've no idea how realistic the numbers are that I'm using in this example, but they're good enough to illustrate the point I want to make.
4. Or some column of the Record Reconstruction Table, since that table is also stored column-wise—but here the parallel with conventional attribute-wise storage is even weaker.
5. I note in passing that the pointers we're talking about here (namely, the ones appearing "first" in each Record Reconstruction Table cell) act as surrogates for field values in exactly the manner explained in Chapter 5, Section 5.6. The others (the ones appearing "second" in each such cell) can be regarded as surrogates too, but the decoding mechanism by which the field values are obtained from those surrogates is slightly different in the latter case.
6. In the extreme, in fact, there could be just one Field Values Table after all. I'll discuss this possibility further in Chapter 15 (Section 15.2).
7. "Horizontal" because the splitting occurs "between records," as it were.