6 Implementing the Update Operators

6.1 Introduction

By now I hope it's clear that, even without the refinements to be discussed in later chapters, the TR model is certainly good for retrieval. (At least in principle! I'll describe in more detail how retrievals are actually implemented in Chapter 10.) But what about updates?[1] Conventional wisdom has always been that a given data structure can be good for either retrieval or update, but not both. In a direct-image implementation, for example, indexes are generally held to be good for retrieval but bad for update. So what about TR? How are updates done in TR? This chapter examines this question.

To repeat from Chapter 5, then, the operators we need to consider are as follows (see Section 5.3):

- *INSERT:* Insert a new record.
- *DELETE:* Delete the record "passing through" cell [*i,j*] of the Record Reconstruction Table.
- *UPDATE:* Update the record "passing through" cell [*i,j*] of the Record Reconstruction Table.

Note: The notion of a record "passing through" some cell of the Record Reconstruction Table was also explained in Section 5.3.

Section 6.2 immediately following discusses the three update operators in general terms; Sections 6.3 then presents a detailed example, and Section 6.4 discusses *the swap algorithm*. Section 6.5 briefly describes an alternative implementation technique that makes use of an *overflow structure*. Finally, Section 6.6 offers some observations regarding the performance aspects of TR update operations.

6.2 Overview

It's convenient to begin by discussing the INSERT operator specifically. Consider the suppliers file shown in Fig. 6.1 (it's the same as the one shown in Fig. 4.1 in Chapter 4, except that the last record, the one for supplier S3, has been omitted). Figs. 6.2 and 6.3 show the corresponding Field Values Table and a corresponding Record Reconstruction Table, respectively. **Exercise 6:** Check that these tables are correct.

field sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
record sequence:	1	S4	Clark	20	London
	2	S5	Adams	30	Athens
	3	S2	Jones	10	Paris
	4	S1	Smith	20	London

Fig. 6.1: A suppliers file

column sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
row sequence:	1	S1	Adams	10	Athens
	2	S2	Clark	20	London
	3	S4	Jones	20	London
	4	S5	Smith	30	Paris

Fig. 6.2: Field Values Table corresponding to the file of Fig. 6.1

column sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
row sequence:	1	4	4	4	4
	2	3	2	2	3
	3	2	1	3	1
	4	1	3	1	2

Fig. 6.3: Record Reconstruction Table corresponding to the file of Fig. 6.1

Now suppose the user asks the system to insert the following tuple into the suppliers relation:

S#	SNAME	STATUS	CITY
S3	Blake	30	Paris

In terms of the file of Fig. 6.1, of course, we can imagine a new record corresponding to this tuple simply being appended at the end, in position *5* (since record ordering within files is arbitrary). If we now rebuild the Field Values Table, it'll appear as shown on the left-hand side of Fig. 6.4 (a copy of the Field Values Table from Fig. 4.3 in Chapter 4). And if we then build a corresponding Record Reconstruction Table, it might appear as shown on the right-hand side of Fig. 6.4

	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. 6.4: Field Values Table and Record Reconstruction Table after inserting supplier S3

As you can see by comparing Fig. 6.4 with Figs. 6.2 and 6.3, respectively, inserting supplier S3 has caused both the Field Values Table and the Record Reconstruction Table to change dramatically. It follows that INSERT operations have the potential to be quite disruptive, and hence (possibly) to display very poor performance. What can be done about this problem?

Well, let me say right away that the effect on the Field Values Table is actually not as dramatic as it might appear. Although I've been calling it a table and showing it as a table in figures like Fig. 6.4, the Field Values Table doesn't necessarily have to be physically stored as a table; in fact, it almost certainly won't be. Much more likely, it'll be stored "column-wise" as a set of *vectors* (one-dimensional arrays), or possibly as a set of *chained lists,* one such vector or list for each column. Indeed, such an implementation is virtually certain to be used in practice if the refinements to be discussed in Chapters 8 and 9 are adopted, as we'll see.[2]

For definiteness, let's assume a vector implementation. Of course, those vectors will be kept in the sort orders associated with the corresponding columns of the Field Values Table. As a consequence, the insert point in each such vector for the pertinent field value from the new record is easily determined—for example, by binary search—and the vectors themselves, and hence the overall Field Values Table, are thus easily maintained.

The Record Reconstruction Table is another matter, however. Is there a way to avoid rebuilding the entire table every time a new record is inserted into the user file? The answer, of course, is *yes*. One possible approach is as follows (the details are a little complicated, but the fundamental idea is straightforward): When a record is *deleted* from the user file, we[3] don't physically remove the corresponding entries from the Field Values and Record Reconstruction Tables, we just *flag* those entries as "logically deleted." Those flagged cells can then be regarded as *free space* in each of the two tables. Then, when we subsequently insert a new record, it might be possible to use such flagged cells for the record in question (removing the flags, of course), thereby avoiding the overhead of completely rebuilding the Record Reconstruction Table (and the overhead of completely rebuilding the Field Values Table also, as a matter of fact). Detailed examples illustrating this process are given in Sections 6.3 and 6.4 below.

I should immediately add that the scheme just described in outline makes considerably more sense if the refinements to be discussed in Chapters 8 and 9 are adopted. If they are—and in practice it's virtually certain they will be—then it becomes possible for distinct records at the file level to *share* entries in the Field Values Table. For example, the supplier records for suppliers S2 and S3 might share the entry in that table that contains the city name Paris. Thus, when a new record is inserted, it might well be the case that most if not all of the field values in that record already exist in the Field Values Table—perhaps logically deleted, perhaps not—and such values can simply be shared by that new record with previously existing records. In effect, the ability to share field values in this way means that INSERT operations work at the field level instead of the usual record level—yet another significant difference between the TR approach and conventional implementation technology. Of course, analogous remarks apply to DELETE and UPDATE operations also, as you'd surely expect.

But what if we want to insert new records before any existing records have been deleted (or, perhaps, before *enough* have been deleted)? Well, in conventional database systems, it's customary to leave a certain amount of free space when the database is initially loaded, in order to accommodate future growth more gracefully. In the same kind of way, with TR, we can specify when we first load the database that certain cells in the Field Values and Record Reconstruction Tables are to be initialized (flagged) as "free-space" cells. In this way, the database can start out with the ability to handle subsequent INSERT operations in a nondisruptive manner.

It follows from all of the above that the algorithm for implementing INSERT operations looks something like this (and here I'm assuming that field values are indeed shared among records in the manner to be explained in detail in Chapter 8):

> *Step 1:* Let *r* be the record to be inserted, and let *f1, f2,* ..., *fn* be the field values within *r*.
>
> *Step 2:* For each *j* in turn (*j* = 1, 2, ..., *n*), do *Step 3*.
>
> *Step 3:* Search column *j* of the Field Values Table for *fj*. If *fj* is not found, insert it. Adjust the Field Values Table to show that the *fj* entry is used by record *r*. Adjust the Record Reconstruction Table accordingly.

Note: Just what it means to adjust the Field Values Table to show that the *fj* entry is used by record *r* is explained in Chapter 8 (Section 8.2, subsection "Row Ranges").

As for DELETE operations, we already know in essence how these operations are implemented: The appropriate entries in the Field Values and Record Reconstruction Tables are simply flagged as logically deleted and thus become free-space cells. As noted earlier, DELETEs are thus effectively done at the field level, rather than the more usual record level.

Analogous remarks apply to UPDATE operations as well, of course, since they can be thought of logically as a DELETE followed by an INSERT. Note in particular that if record *r* is updated to become record *r′*,[4] but the value of field *F* in *r′* is the same as that in *r* (meaning, loosely, that "field *F* hasn't been updated"), then the internal-level operation that the implementation has to execute for field *F* is essentially "Do nothing"!—in effect, the new record *r′* and the old record *r* can simply share the applicable field value. (Of course, the foregoing is just a manner of speaking; I don't mean to suggest that the old record *r* is still kept around after the update has been done. Though it might be, if the database in question is a temporal one [42].)

6.3 A Detailed Example

Now let's take a closer look at exactly how updates are done in TR. *Note:* Actually TR supports a variety of distinct update techniques, and it's obviously not possible to cover them all in a book of this nature. The explanations that follow are thus certainly not meant to be exhaustive; rather, they're offered just as an indication of the kinds of techniques that might be used in practice.

By way of an example, consider the suppliers file shown in Fig. 6.5 (it's the same as the one shown in Fig. 6.1, except that supplier S3 has been reinstated and two new suppliers, S6 and S7, have been added). Figs. 6.6 and 6.7 show the corresponding Field Values Table and a corresponding Record Reconstruction Table, respectively. **Exercise 7:** Once again, check that these tables are correct.

field sequence: 1 2 3 4

record sequence:	S#	SNAME	STATUS	CITY
1	S4	Clark	20	London
2	S5	Adams	30	Athens
3	S2	Jones	10	Paris
4	S1	Smith	20	London
5	S3	Blake	30	Paris
6	S6	Brady	30	Athens
7	S7	Patel	40	Haifa

Fig. 6.5: The suppliers file of Fig. 6.1 after inserting suppliers S3, S6, and S7

column sequence: 1 2 3 4

row sequence:	S#	SNAME	STATUS	CITY
1	S1	Adams	10	Athens
2	S2	Blake	20	Athens
3	S3	Brady	20	Haifa
4	S4	Clark	30	London
5	S5	Jones	30	London
6	S6	Patel	30	Paris
7	S7	Smith	40	Paris

Fig. 6.6: Field Values Table corresponding to the file of Fig. 6.5

column sequence: 1 2 3 4

row sequence:	S#	SNAME	STATUS	CITY
1	7	4	6	5
2	5	6	4	6
3	2	5	5	7
4	4	3	1	1
5	1	1	2	4
6	3	7	7	2
7	6	2	3	3

Fig. 6.7: Record Reconstruction Table corresponding to the file of Fig. 6.5

Now suppose the user asks for the records—or, rather, the tuples corresponding to the records—for suppliers S3 (Blake) and S7 (Patel) to be deleted. All the implementation does at this point is follow the applicable zigzags and flag the applicable cells in the Field Values and Record Reconstruction Tables as free space (see Figs. 6.8 and 6.9, where the flags are shown as asterisks). *Note:* I'll refer to such flagged cells as *free cells* from this point forward.

column sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
row sequence:	1	S1	Adams	10	Athens
	2	S2	*Blake	20	Athens
	3	*S3	Brady	20	*Haifa
	4	S4	Clark	30	London
	5	S5	Jones	30	London
	6	S6	*Patel	*30	Paris
	7	*S7	Smith	*40	*Paris

Fig. 6.8: Field Values Table after deleting suppliers S3 and S7

column sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
row sequence:	1	7	4	6	5
	2	5	*6	4	6
	3	*2	5	5	*7
	4	4	3	1	1
	5	1	1	2	4
	6	3	*7	*7	2
	7	*6	2	*3	*3

Fig. 6.9: Record Reconstruction Table after deleting suppliers S3 and S7

Now suppose the user asks the system to insert the following tuple into the suppliers relation:

S#	SNAME	STATUS	CITY
S3	Paige	40	Paris

Note: In practice, the DELETE that caused the old S3 tuple to be deleted and the INSERT that's now asking for the new S3 tuple to be inserted might have been bundled into a single UPDATE request, of course:

```
UPDATE S
SET SNAME = NAME('Paige'),
STATUS = 40
WHERE S# = S#('S3') ;
```

Be that as it may, you can see from Fig. 6.8 that free cells for supplier number S3, status 40, and city name Paris do all exist in the Field Values Table. As for the supplier name, Paige, at least there is a free cell at the right place in the applicable sort order (namely, that for Patel), so we can use that one, too, so long as we change the name it contains from Patel to Paige. Figs. 6.10 and 6.11 show the revised versions of the Field Values Table and the Record Reconstruction Table, respectively. Note that we've removed the flags in both tables from the free cells we've used (namely, cells [*3,1*], [*6,2*], [*7,3*], and [*7,4*]). We've also revised the Record Reconstruction Table so that the corresponding cells are linked together into a zigzag appropriately (to be specific, we've changed the contents of cell [*3,1*] from *2* to *6* and the contents of cell [*7,3*] from *3* to *7*).

column sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
row sequence:	1	S1	Adams	10	Athens
	2	S2	*Blake	20	Athens
	3	S3	Brady	20	*Haifa
	4	S4	Clark	30	London
	5	S5	Jones	30	London
	6	S6	Paige	*30	Paris
	7	*S7	Smith	40	Paris

Fig. 6.10: Field Values Table after inserting (a revised version of) supplier S3

column sequence:		1	2	3	4
		S#	SNAME	STATUS	CITY
row sequence:	1	7	4	6	5
	2	5	*6	4	6
	3	2	5	5	*7
	4	4	3	1	1
	5	1	1	2	4
	6	3	7	*7	2
	7	*6	2	3	3

Fig. 6.11: Record Reconstruction Table after inserting (a revised version of) supplier S3

6.4 The Swap Algorithm

As indicated in Sections 6.2 and 6.3, a key notion underlying the TR update algorithms is that cells in the Field Values Table and Record Reconstruction Table can be "recycled" (that is, they can be used and reused, over and over again). To be more specific, deletions cause certain "holes" (free cells) to open up in the TR tables, and those "holes" can then be used by subsequent insertions. However, one important point I deliberately glossed over previously is that it might be necessary *to move those "holes" around from time to time within their containing tables* (I'm speaking pretty loosely here, as I'm sure you'll appreciate). The update algorithms are designed to work in such a way as to keep that moving around localized to as small a region as possible, with the overall objective of keeping the TR tables as static as possible and thereby minimizing the overhead. A variety of techniques can be used to achieve this desirable effect; in this section, I want to focus on just one of those techniques: namely, the so-called *swap algorithm* [63].

In order to illustrate the swap algorithm in action, as it were, let me revise the example from the previous section as follows. First, assume that the Field Values Table and Record Reconstruction Table are again as given in Figs. 6.8 and 6.9, respectively. Now assume that the tuple to be inserted looks like this:

S#	SNAME	STATUS	CITY
S3	Blake	20	Athens

Referring again to Fig. 6.8, we see that:

- Free cells exist in the Field Values Table for supplier number S3 and supplier name Blake, so we can use those cells directly, as in the example in the previous section. That takes care of the S# and SNAME values.

- As for the city name, Athens, there's no free cell for Athens as such, but at least there's a free cell in a suitable position (where by "suitable" I mean a position that doesn't disturb the sort order)—namely, the free cell for Haifa. So we can use that cell too, changing the name it contains from Haifa to Athens. That takes care of the CITY value.

But what do we do about the STATUS value? Not only is there no free cell for status 20, there isn't even any free cell in the right position (that is, immediately before or after the sequence of cells for status 20 that do currently exist).

Observe, however, that at least there *is* a free STATUS cell, for status 30, in row *6* of the Field Values Table (that is, the cell in question is cell [*6,3*]). If we could somehow *swap* that cell with cell [*4,3*]—which also contains the status value 30—the free cell would then be immediately adjacent to the existing sequence of cells for status 20, and we could then use it for the new record, just as we used the Haifa cell for Athens.

In order to implement that swap, we need to reroute the zigzag in the Record Reconstruction Table that runs through cell [*4,3*] so that it runs through cell [*6,3*] instead. That zigzag corresponds (as it happens) to supplier S5, and the effect of the swap will be that supplier S5's status value, 30, will then be the one in cell [*6,3*]—instead of cell [*4,3*]—of the Field Values Table. After making the swap, we can flag cell [*4,3*] as free and "unflag" cell [*6,3*] to mark it "unfree." To spell out the details:

- From Fig. 6.9, we can see that the zigzag for supplier S5 currently looks like this (that is, it currently involves the following sequence of Record Reconstruction Table cells):

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

- The swap can thus be done by:

  a) Changing the contents of cell [*1,2*] of the Record Reconstruction Table from *4* to *6*, and
  b) Changing the contents of cell [*6,3*] of the Record Reconstruction Table from *7* to *1*.

- The zigzag for supplier S5 will then look like this:

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

  (as required).

Now we can replace the status value 30 in cell [*4,3*] of the Field Values Table by the status value 20 (and flag that cell as free and remove the flag from cell [*6,3*], at the same time flagging and unflagging the corresponding cells in the Record Reconstruction Table analogously).

After all the foregoing activity has been completed, there's a free cell in the Field Values Table for every value in the record that we're trying to insert. We can therefore accomplish the desired insertion by linking those free cells into a zigzag in the Record Reconstruction Table—to be specific, a zigzag that looks like this:

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

In other words, we set cell [*3,1*] to contain *2,* cell [*2,2*] to contain *4,* cell [*4,3*] to contain *3,* and cell [*3,4*] to contain *3* as well. Also, of course, we remove the flags from all of these previously free cells, in both the Field Values Table and the Record Reconstruction Table. The net effect is shown in Figs. 6.12 and 6.13.

column sequence: 1 2 3 4

row sequence:	S#	SNAME	STATUS	CITY
1	S1	Adams	10	Athens
2	S2	Blake	20	Athens
3	S3	Brady	20	Athens
4	S4	Clark	20	London
5	S5	Jones	30	London
6	S6	*Patel	30	Paris
7	*S7	Smith	*40	*Paris

Fig. 6.12: Field Values Table after inserting (a different revised version of) supplier S3

column sequence: 1 2 3 4

row sequence:	S#	SNAME	STATUS	CITY
1	7	6	6	5
2	5	4	4	6
3	2	5	5	3
4	4	3	3	1
5	1	1	2	4
6	3	*7	1	2
7	*6	2	*3	*3

Fig. 6.13: Record Reconstruction Table after inserting (a different revised version of) supplier S3

Points Arising

The swap algorithm as described above has an interesting side effect in our particular example, as follows: While the Field Values Table still has one set of free cells after the INSERT (because we started with seven records, deleted two, and then inserted one), those cells are no longer chained together. Equivalently, the Record Reconstruction Table also has one set of free cells, but those cells don't form a valid zigzag—the pointer chain is broken. In general, in fact, if we start chasing pointers from a free cell in the Record Reconstruction Table, we won't necessarily find ourselves in a closed ring. Of course, this fact isn't very important, because we never need to do record reconstruction on deleted records anyway.

Another consequence is that, given any particular column of the Record Reconstruction Table, certain row numbers will be duplicated in that column and others will be missing (in general). For example, in Fig. 6.13, columns *3* and *4* both include two *3*'s and no *7*. Note, however, that within any such column:

- No two "unfree" cells will ever contain the same row number.
- Any missing row number would, if present, point to a free cell.

The algorithm has another side effect, too. Suppose we process column *3* (the STATUS column) of the Record Reconstruction Table top to bottom in order to reconstruct the suppliers file in ascending status sequence. With the original version of that table as shown in Fig. 6.7, supplier S5 will precede supplier S6 in the result; with the version of the table shown in Fig. 6.13, by contrast, supplier S6 will precede supplier S5 instead.

And one more point: In the particular example discussed above, the necessary free cell for the new status value, 20, was found in the Field Values Table in the immediately adjacent sequence of cells (namely, the sequence of cells for the next recorded status value, 30). Suppose there had been no free "30" cell but (say) a free "40" cell instead. Then two swaps would have been necessary, one to make that free "40" cell into a free "30" cell, and then another to make that free "30" cell into a free "20" cell. In general, if there are several intervening sequences without any free cells, then several swaps will need to be carried out, each swap moving the free cell one position closer to the place where it's really needed.

6.5 Using an Overflow Structure

In Section 6.2, we saw that insertion of a new record doesn't always require insertion of brand new field values; in fact, we'll see in Chapters 8 and 9 that it *very rarely* requires insertion of brand new field values. And if there are no new field values to insert, the INSERT operation will clearly be faster than it would otherwise be. In Sections 6.3 and 6.4, by contrast, we considered what happens if there are indeed new field values to insert; to be specific, in Section 6.4 we described the swap algorithm for moving "holes" around to get them into the right place for the new values. But there's another way to deal with such new values—one that involves no swapping as such—that might be more efficient in practice, especially in a disk-based implementation. I don't want to get into a lot of detail here, but in essence the technique involves storing the new values in a separate overflow structure and periodically merging the data from that structure into the main database.[5] This technique has the properties that:

- The overflow structure can be thought of as containing its own private Field Values Table and Record Reconstruction Table; thus, for example, binary searches can be used on the overflow data.

- The principal Field Values Table and Record Reconstruction Table—and hence the main database—remain unchanged most of the time; they change only during the process of performing the periodic merge (see the point immediately following).

- The period between successive merges can be quite lengthy. For example, imagine a database containing sales records for every day of the past five years, with a nightly batch insert for that day's figures. Assume that batch insert corresponds to roughly one twentieth of one percent of the total database size. If we allow the overflow structure to grow to (say) ten percent of the total database size before we do the merge, the period between successive merges will be of the order of six or seven *months*.

Using an overflow structure has another big advantage, too: It greatly simplifies the familiar backup and recovery process. In outline, that process works as follows:

- We create a full backup copy of the entire database only when we do a merge (which, as we've just seen, isn't likely to be all that often).

- We create a backup copy of the (comparatively small) overflow structure every time it's been significantly changed—perhaps every night.

- If it's necessary to perform recovery, we simply restore the most recent full backup copy and the most recent overflow backup copy; it's *not* like having to apply a whole series of "incremental backups," which is what conventional systems typically do have to do.

6.6 Some Remarks on Performance

I'll close this chapter with a few remarks on the performance aspects of TR update operations. The fact is, a TR implementation should significantly outperform traditional DBMSs on updates as well as on queries. There are several reasons for this state of affairs:

- *DELETE and UPDATE:* Note first that certain of the remarks made in connection with retrieval performance in Chapter 5 (Section 5.2) apply to the performance of DELETE and UPDATE operations also. To be specific, it's often the case that a significant part of the work involved in deleting or updating data is in finding the relevant data in the first place—and TR is very good at finding data, and finding it fast.

- *INSERT:* Suppose we're trying to insert a new supplier tuple, for supplier S9, say. Then we want to check—or rather, as users, we want the *system* to check—that there's no tuple for supplier S9 in the suppliers relation already. At the implementation level, this requirement implies that the system has to search for a record for supplier S9 before doing the INSERT (if it finds one, the INSERT will have to be rejected, of course). As we saw in Section 6.2, however, it's going to do that search anyway. Once again, therefore, we're talking about the problem of "finding data and finding it fast" (see the previous paragraph; see also the discussion of integrity constraints in Chapter 10, Section 10.10).

- *Lack of redundancy:* There are two points here. First, since there aren't any auxiliary structures such as indexes, there aren't any auxiliary structures such as indexes to update. Second, the refinements to be discussed in Chapters 8 and 9 have the effect of reducing data redundancy still further—dramatically so, in fact—thereby simplifying the update process still further (and considerably).

- *Sorted data:* The precise means (that is, the Field Values Table) by which the stored data is kept in many sort orders simultaneously makes it easy to maintain those sort orders when updates occur—certainly much easier than the corresponding task with indexes or other conventional auxiliary structures.

- *Limiting the scope of impact:* In the real world, even in extremely active systems, updates tend to affect only a tiny portion of the overall database. The TR update algorithms are designed to take advantage of this fact; in effect, they regard the database as a large and static thing, and they keep all changes in a much smaller dynamic repository.[6] In other words (as I said near the beginning of Section 6.4, more or less), they generally try to keep the impact of any given user-level update confined to as small a portion of the database as possible, thereby minimizing the amount of update overhead. This philosophy is in marked contrast to that found in conventional systems; in particular, it's very different from what happens with conventional indexing, where everything is assumed to be completely dynamic (at least potentially), and individual updates can ripple out and cause further updates that need to be applied "all over the database."

Endnotes

1. I remind you from Chapter 1 that I use the term "update" (lower case) to mean the INSERT, DELETE, and UPDATE operators considered generically, and the term "UPDATE" (upper case) to mean the UPDATE operator specifically.
2. It isn't particularly relevant to the present discussion, but you should be aware that analogous remarks apply to the Record Reconstruction Table as well—that is, that table too will almost certainly be stored column-wise. And the same is true for the Permutation and Inverse Permutation Tables also, if those tables are physically stored.
3. "We" here really means the DBMS.
4. I'm being sloppy here. As explained in references [32] and [40], it would be more accurate to talk in terms of record *r* being replaced by record *r′*—but it's conventional to talk in terms of records being updated, even though, strictly speaking, records are values and can't possibly be updated. Analogous remarks apply to fields also.
5. In fact, the technique can be used for values that *aren't* brand new, too.
6. This perception is supported very directly by the overflow structure mechanism sketched in the previous section, but it's effectively supported by TR's other update techniques as well.