CHAPTER 11

Storage and File Structure

Exercises

11.3 Answer: This arrangement has the problem that P_i and B_{4i-3} are on the same disk. So if that disk fails, reconstruction of B_{4i-3} is not possible, since data and parity are both lost.

11.4 Answer:

a. To ensure atomicity, a block write operation is carried out as follows:-

i. Write the information onto the first physical block.

ii. When the first write completes successfully, write the same information onto the second physical block.

iii. The output is declared completed only after the second write completes successfully.

During recovery, each pair of physical blocks is examined. If both are identical and there is no detectable partial-write, then no further actions are necessary. If one block has been partially rewritten, then we replace its contents with the contents of the other block. If there has been no partial-write, but they differ in content, then we replace the contents of the first block with the contents of the second, or vice versa. This recovery procedure ensures that a write to stable storage either succeeds completely (that is, updates both copies) or results in no change.

The requirement of comparing every corresponding pair of blocks during recovery is expensive to meet. We can reduce the cost greatly by keeping track of block writes that are in progress, using a small amount of non-volatile RAM. On recovery, only blocks for which writes were in progress need to be compared.

b. The idea is similar here. For any block write, the information block is written first followed by the corresponding parity block. At the time of recovery, each set consisting of the n^{th} block of each of the disks is considered. If none of the blocks in the set have been partially-written, and the parity block contents are consistent with the contents of the information blocks, then no further action need be taken. If any block has been partially-written, it's contents are reconstructed using the other blocks. If no block has been partially-written, but the parity block contents do not agree with the information block contents, the parity block's contents are reconstructed.

11.6 Answer:

a. MRU is preferable to LRU where $R_1 \bowtie R_2$ is computed by using a nested-loop processing strategy where each tuple in R_2 must be compared to each block in R_1. After the first tuple of R_2 is processed, the next needed block is the first one in R_1. However, since it is the least recently used, the LRU buffer management strategy would replace that block if a new block was needed by the system.

b. LRU is preferable to MRU where $R_1 \bowtie R_2$ is computed by sorting the relations by join values and then comparing the values by proceeding through the relations. Due to duplicate join values, it may be necessary to "back-up" in one of the relations. This "backing-up" could cross a block boundary into the most recently used block, which would have been replaced by a system using MRU buffer management, if a new block was needed.

Under MRU, some unused blocks may remain in memory forever. In practice, MRU can be used only in special situations like that of the nested-loop strategy discussed in example 0.a

11.7 Answer:

a. Although moving record 6 to the space for 5, and moving record 7 to the space for 6, is the most straightforward approach, it requires moving the most records, and involves the most accesses.

b. Moving record 7 to the space for 5 moves fewer records, but destroys any ordering in the file.

c. Marking the space for 5 as deleted preserves ordering and moves no records, but requires additional overhead to keep track of all of the free space in the file. This method may lead to too many "holes" in the file, which if not compacted from time to time, will affect performance because of reduced availability of contiguous free records.

11.8 Answer: (We use "$\uparrow i$" to denote a pointer to record "*i*".)
The original file of Figure 11.9.

header	$\uparrow 1$			
record 0		Perryridge	A-102	400
record 1	$\uparrow 4$			
record 2		Mianus	A-215	700
record 3		Downtown	A-101	500
record 4	$\uparrow 6$			
record 5		Perryridge	A-201	900
record 6				
record 7		Downtown	A-110	600
record 8		Perryridge	A-218	700

a. The file after **insert** (Brighton, A-323, 1600).

header	$\uparrow 4$			
record 0		Perryridge	A-102	400
record 1		Brighton	A-323	1600
record 2		Mianus	A-215	700
record 3		Downtown	A-101	500
record 4	$\uparrow 6$			
record 5		Perryridge	A-201	900
record 6				
record 7		Downtown	A-110	600
record 8		Perryridge	A-218	700

b. The file after **delete** record 2.

header	$\uparrow 2$			
record 0		Perryridge	A-102	400
record 1		Brighton	A-323	1600
record 2	$\uparrow 4$			
record 3		Downtown	A-101	500
record 4	$\uparrow 6$			
record 5		Perryridge	A-201	900
record 6				
record 7		Downtown	A-110	600
record 8		Perryridge	A-218	700

The free record chain could have alternatively been from the header to 4, from 4 to 2, and finally from 2 to 6.

c. The file after **insert** (Brighton, A-626, 2000).

header	↑ 4			
record 0		Perryridge	A-102	400
record 1		Brighton	A-323	1600
record 2		Brighton	A-626	2000
record 3		Downtown	A-101	500
record 4	↑ 6			
record 5		Perryridge	A-201	900
record 6				
record 7		Downtown	A-110	600
record 8		Perryridge	A-218	700

11.11 Answer:

a. **insert** (Mianus, A-101, 2800) changes record 2 to:

2	Mianus	A-215	700	A-101	2800	$\perp$	$\perp$

b. **insert** (Brighton, A-323, 1600) changes record 5 to:

5	Brighton	A-216	750	A-323	1600	$\perp$	$\perp$

c. **delete** (Perryridge, A-102, 400) changes record 0 to:

0	Perryridge	A-102	900	A-218	700	$\perp$	$\perp$

11.13 Answer:

a. The figure after **insert** (Mianus, A-101, 2800).

0	↑ 5	Perryridge	A-102	400
1		Round Hill	A-305	350
2	↑ 9	Mianus	A-215	700
3	↑ 7	Downtown	A-101	500
4		Redwood	A-222	700
5	↑ 8		A-201	900
6		Brighton	A-216	750
7			A-110	600
8			A-218	700
9			A-101	2800

b. The figure after **insert** (Brighton, A-323, 1600).

0	$\uparrow$ 5	Perryridge	A-102	400
1		Round Hill	A-305	350
2	$\uparrow$ 9	Mianus	A-215	700
3	$\uparrow$ 7	Downtown	A-101	500
4		Redwood	A-222	700
5	$\uparrow$ 8		A-201	900
6	$\uparrow$ 10	Brighton	A-216	750
7			A-110	600
8			A-218	700
9			A-101	2800
10			A-323	1600

c. The figure after **delete** (Perryridge, A-102, 400).

1		Round Hill	A-305	350
2	$\uparrow$ 9	Mianus	A-215	700
3	$\uparrow$ 7	Downtown	A-101	500
4		Redwood	A-222	700
5	$\uparrow$ 8	Perryridge	A-201	900
6	$\uparrow$ 10	Brighton	A-216	750
7			A-110	600
8			A-218	700
9			A-101	2800
10			A-323	1600

11.18 Answer:

course relation

course-name	*room*	*instructor*	
Pascal	CS-101	Calvin, B	c_1
C	CS-102	Calvin, B	c_2
LISP	CS-102	Kess, J	c_3

enrollment relation

course-name	*student-name*	*grade*	
Pascal	Carper, D	A	e_1
Pascal	Merrick, L	A	e_2
Pascal	Mitchell, N	B	e_3
Pascal	Bliss, A	C	e_4
Pascal	Hames, G	C	e_5
C	Nile, M	A	e_6
C	Mitchell, N	B	e_7
C	Carper, D	A	e_8
C	Hurly, I	B	e_9
C	Hames, G	A	e_{10}
Lisp	Bliss, A	C	e_{11}
Lisp	Hurly, I	B	e_{12}
Lisp	Nile, M	D	e_{13}
Lisp	Stars, R	A	e_{14}
Lisp	Carper, D	A	e_{15}

Block 0 contains: c_1, e_1, e_2, e_3, e_4, and e_5
Block 1 contains: c_2, e_6, e_7, e_8, e_9 and e_{10}
Block 2 contains: c_3, e_{11}, e_{12}, e_{13}, e_{14}, and e_{15}

11.19 Answer:

a. Everytime a record is inserted / deleted, check if the usage of the block has changed levels. In that case, update the corrosponding bits. Note that we don't need to access the bitmaps at all unless the usage crosses a boundary, so in most of the cases there is no overhead.

b. When free space for a large record or a set of records is sought, then multiple free list entries may have to be scanned before finding a proper sized one, so overheads are much higher. With bitmaps, one page of bitmap can store free info for many pages, so IO spent for finding free space is minimal. Similarly, when a whole block or a large part of it is deleted, bitmap technique is more convenient for updating free space information.

11.22 Answer:

If an object gets forwarded multiple times, the retrieval speed will decrease because accessing it will require accessing the series of locations from which the object has been successively forwarded to the current location.

To avoid multiple accesses, whenever an object is accessed using an old pointer, update each pointer in the path to point to the current location of the object. With this path compression, whenever the object is accessed again through any pointer in that path, the object can be directly reached.

11.24 Answer: While swizzling, if the short identifier of page 679.34278 is changed from 2395 to 5001, it is either because

a. the system discovers that 679.34278 has already been allocated the virtual-memory page 5001 in some previous step, or else

b. 679.34278 has not been allocated any virtual memory page so far, and the free virtual memory page 5001 is now allocated to it.

Thus in either case, it cannot be true that the current page already uses the same short identifier 5001 to refer to some database page other than 679.34278. Some other page may use 5001 to refer to a different database page, but then each page has its own independent mapping from short to full page identifiers, so this is all right.

Note that if we do swizzling as described in the text, and different processes need simultaneous access to a database page, they will have to map separate copies of the page to their individual virtual address spaces. Extensions to the scheme are possible to avoid this.