Database Design and Implementation
Chapter 3 Conceptual Exercises

Questions	Answers
3.1.a.	Total Sectors = Tracks per Platter × Sectors per Track Total Sectors = 50,000 × 500 = 25,000,000 Total Capacity= Total Sectors × Bytes per Sector Total Capacity = 25,000,000 × 512 = 12,800,000,000 bytes
3.1.b.	Time for one rotation = 60 seconds / 7200 rpm = 1/120 seconds Average Rotational Delay = 1/240 seconds
3.1.c.	Sectors per Rotation = 500 Maximum Transfer Rate = (Sectors per Rotation × Bytes per Sector) / Time for one rotation Maximum Transfer Rate = 500 × 512 / (1 / 120) = 30,720,000 bytes/sec
3.2.a.	80 GB = 80 × 1024 MB = 80 × 1024 × 1024 KB= 80 × 1024 × 1024 × 1024 bytes = 85,899,345,920 bytes Bytes per Track= (100 × 1024 × 1024) / (7200 × 60) = 146800.64 bytes/track Number of tracks = 85,899,345,920 bytes / 146800.64 bytes/track = 585,000
3.2.b.	New Bytes per track = (100 x 1024 x 1024) / (10,000 x 60) = 139.81 MB/s
3.3.a.	Actual Drive = M / (500 × 10) = M / 5000 Actual Sector = M mod 500
3.3.b.	Track-sized stripes might be more efficient, as they minimize the number of drives that need to be accessed for a single read or write operation. When you read an entire track, you only need to engage the read/write head on one drive. This reduces the latency and mechanical wear and tear associated with moving the heads across multiple drives.
3.3.c.	Track-sized stripes might be less efficient in handling small read or write requests. Each request would require reading or writing an entire track, even if only a small portion of the data is needed. This could result in unnecessary data transfers, and increased time and resource consumption.
3.4.a.	1. Identify the failed disk 2. Hot swap the failed disk 3. Initialize and synchronize 4. Monitor Based on my algorithm, the risk of a second disk failure is increased during the syncing process due to the added stress and workload on the remaining operational disk. Use disks from different manufacturers to decrease the likelihood that both will fail close to the same time. Also, monitor the health of the operational disk more closely during the sync process.
3.4.b.	1. Identify and mark the failed disk 2. Hot swap

	3. Rebuild parity 4. Update the second disk The risk of a second disk failure is present, especially when rebuilding parity. Use disks from different manufacturers to decrease the likelihood that both will fail close to the same time. Also, monitor the health of the operational disk more closely during the rebuild parity process.
3.5.a.	In RAID-5, data and parity are striped across all disks. If a disk fails, its data can be recovered using the remaining data and the striped parity information.
3.5.b.	Both RAID-4 and RAID-5 require reading the original data and the corresponding parity data to perform a write operation. Therefore, the number of disk accesses for reads and writes remains the same in both cases.
3.5.c.	RAID-5 distributes the parity information across all disks, avoiding the bottleneck that can occur with a dedicated parity disk in RAID-4. This results in better load balancing and improved overall performance.
3.6.	To reconstruct the content of a failed striped disk in a RAID setup with a parity disk, you would read the corresponding sectors from all the other striped physical disks and the parity disk. The parity information can then be used to recreate the data on the failed disk.
3.7.a.	Number of Blocks = (1024^3) / (4 * 1024) = 262,144
3.7.b.	Bits needed = 262,144 Blocks for Disk Map = 262,144 / (4 * 1024 * 8) = 8
3.8.	1. For the Disk Map, we see a sequence of binary values. A '0' represents a free block while a '1' represents an occupied block. After allocate(1,4): Blocks 1 to 4 are set to '1' (occupied). After allocate(4,10): Since block 4 is already occupied, this allocation starts from block 5 and occupies blocks 5 to 14. After allocate(5,12): Since blocks 5 to 14 are already occupied, this allocation starts from block 15 and occupies blocks 15 to 26. Resulting Disk Map: 00101111111111111111111111...

	2. For the free list, each block points to the next free block. Initial state: Block 0 points to block 2, block 2 is free. After allocate(1,4): Block 2 still points to the next free block since it's outside the allocation range. After allocate(4,10): Block 2 will point to block 15 since blocks 2 to 14 are now occupied. After allocate(5,12): Block 2 will still point to block 15 since the start of the new allocation (15) doesn't affect it, but it fills up blocks till 26. Resulting Free List: Block 2 points to block 27 (or whichever is the next free block post-26).
3.9.	Failed Bit = Striped Physical Disks XOR Parity Disk 1011 0001 0011
3.10.a.	When using the free list allocation strategy, multiple contiguous chunks can end up on the free list. To prevent this and allow for merging of contiguous chunks: Sorted Free List: Keep the free list sorted by the starting address. This ensures that contiguous chunks are next to each other in the list. Merging: When deallocating (or adding) a chunk to the free list, check its neighbors in the list. If the chunk to be added is contiguous with its neighbors, merge them to form a single larger chunk.
3.10.b.	Merging unallocated chunks is beneficial for contiguous file allocation for two reasons. Reducing Fragmentation: Merging helps in reducing external fragmentation, ensuring that large chunks of free space are available when needed. Improving Allocation Efficiency: With reduced fragmentation, it's easier and quicker to find a suitable chunk of space for a new file, especially when files are allocated contiguously.
3.10.c.	Merging is not important for extent-based or indexed file allocation for three reasons.

	Non-Contiguous Nature: Both extent-based and indexed file allocations inherently support non-contiguous data. Files are not necessarily stored in one continuous chunk of space, so there's less need to have large contiguous free spaces. Fixed Size Extents: In extent-based allocation, files are divided into fixed-size extents, reducing the importance of merging free spaces. Indexed Allocation Flexibility: In indexed allocation, a file can be spread over numerous non-contiguous blocks, referred to by an index block. This reduces the need to merge free blocks.
3.11.a	Size = Sum of all extents = 240 + 132 + 60 + 252 + 12 + 24 = 720 blocks.
3.11.b	Block 2 is in the first extent (240): Physical block = 240 + 2 = 242. Block 12 is in the first extent (240): Physical block = 240 + 12 = 252. Block 23 is in the second extent (132): Physical block = 132 + (23 - 12) = 143. Block 34 is in the third extent (60): Physical block = 60 + (34 - 24) = 70. Block 55 is in the fourth extent (252): Physical block = 252 + (55 - 36) = 271.
3.12.	Given block size is 4K bytes, in indexed file allocation, each entry in the index block would point to a data block. If each entry is 4 bytes (size of a memory address): Maximum blocks indexed = (4K bytes/block) / (4 bytes/entry) = 1024 blocks. Largest possible file = 1024 blocks * 4K bytes/block = 4MB.
3.13.a.	As the block size is 4K bytes in an inode structure… Number of data blocks an index block refers to. = 4K bytes/block / 4 bytes/entry = 1024 blocks.
3.13.b.	Ignoring the double-index block, maximum UNIX file size = 12 direct blocks + 2 index blocks * 1024 blocks/index block = 12 + 2*1024 = 2060 blocks. File size = 2060 blocks * 4K bytes/block = 8.24MB
3.13.c.	Data blocks a double-index block refers to. = 1024 (from one index block) * 1024 (from the second) = 1,048,576 blocks.
3.13.d.	Combining direct, single-index, and double-index = 12 + 2*1024 + 1,048,576 = 1,050,612 blocks. Largest Possible File size = 1,050,612 blocks * 4K bytes/block = 4.2TB.
3.13.e.	Block accesses for the last block of a 1GB file. = 1 (to access the inode) + 1 (to access the double-index block) + 1 (to access the index block) + 1 (to access the data block) = 4 accesses.
3.13.f.	1. Check if the desired block is among the direct blocks in the inode. If so, access it.

	2. If not, calculate which index block it's in. Access the appropriate index block. 3. If it's in the double-index block range, access the appropriate index block within the double-index, then the desired block.
3.14.	The pun "On a clear disk you can seek forever" is clever and plays on the similarity in sound between "day" and "disk" and "see" and "seek". It humorously conveys that on a clean or empty disk, the disk head can keep seeking without hitting any data. While it's entertaining, in a real-world scenario, seeking endlessly isn't beneficial. Efficient disk operations aim to minimize seek times.