Database Design and Implementation
Chapter 6 Conceptual Exercises

<table>
<tr><th>Questions</th><th>Answers</th></tr>
<tr><td>6.1.</td><td>
Given:

Block size: 400 bytes
Empty/Full flag: 1 byte (based on Fig. 6.5)
Each record has a 1-byte flag preceding it

Steps:

Calculate the number of records for each slot size.
Calculate the total bytes used for these records.
Subtract the total bytes used from the block size to determine the wasted space.

Slot Size: 10 bytes

Max records = (400 bytes - 1 byte flag) / (10 bytes + 1 byte flag) = 36 records
Total bytes used = 36 records * (10 bytes + 1 byte flag) = 396 bytes
Wasted space = 400 bytes - 396 bytes = 4 bytes

Slot Size: 20 bytes

Max records = (400 bytes - 1 byte flag) / (20 bytes + 1 byte flag) = 19 records
Total bytes used = 19 records * (20 bytes + 1 byte flag) = 399 bytes
Wasted space = 400 bytes - 399 bytes = 1 byte

Slot Size: 50 bytes

Max records = (400 bytes - 1 byte flag) / (50 bytes + 1 byte flag) = 7 records
Total bytes used = 7 records * (50 bytes + 1 byte flag) = 357 bytes
Wasted space = 400 bytes - 357 bytes = 43 bytes

Slot Size: 100 bytes

Max records = (400 bytes - 1 byte flag) / (100 bytes + 1 byte flag) = 3 records
Total bytes used = 3 records * (100 bytes + 1 byte flag) = 303 bytes
Wasted space = 400 bytes - 303 bytes = 97 bytes

</td></tr>
<tr><td>6.2.</td><td>

Blocks might have been allocated to the table initially, but over time, records may be deleted leaving the block empty.
When records are moved due to operations like defragmentation, blocks can end up being empty.
Some database management systems might pre-allocate blocks for tables anticipating future data insertion.

</td></tr>
<tr><td>6.3.a.</td><td>
<table>
<tr><td>Name</td><td>Type</td><td>Length</td><td>Offset</td></tr>
<tr><td>DId</td><td>int</td><td>4</td><td>1</td></tr>
</table>
</td></tr>
</table>

	<table><tr><td>DName</td><td>Varchar20</td><td>22</td><td>5</td></tr></table> Note: This is a Dept Table.
6.3.b.	We would create a diagram similar to Fig. 6.5 with each slot showing 26 bytes.
6.3.c.	The layout would vary based on the actual length of the string values.
6.3.d.	For both fixed and variable-length implementations, the slot containing the second record would be marked as empty.
6.4.	**Reasons against storing large strings outside the database**: 1. **Data Integrity**: There's a risk that the OS file might get deleted, renamed, or moved, making the database reference invalid. 2. **Backup and Recovery**: Backup operations need to include both the database and the OS files, complicating the process. 3. **Transaction Management**: OS files aren't part of the database's transaction management, leading to potential consistency issues. 4. **Security**: Database security mechanisms can't be applied to OS files directly. 5. **Portability**: Transferring or replicating the database requires additional steps to ensure OS files are also transferred.
6.5.	No, it's not typically a good idea to directly store the record in the overflow block unless the primary block is full. Doing so can lead to inefficient data retrieval. Overflow blocks are utilized when there's no space left in the primary data block to accommodate new or modified data. When you want to insert a record, it's generally preferable to store it in the primary block for more direct access. Storing in the overflow block can lead to increased retrieval times since it requires an extra step to look into the overflow block after checking the primary block.
6.6.a.	If a variable-length value increases in size and cannot fit in its original space, it should be moved to a new location where it fits. An overflow block might be required if there's no space in the value area. The overflow block would look similar to the value area of the primary block, storing variable-length values. When a variable-length value gets modified (either increasing or decreasing in size), the space it occupies in the value area might change. If the modified value cannot fit in the original space, then a new space must be found or created for it. This might necessitate the use of an overflow block if no suitable space is available in the value area.
6.6.b.	The variable-length with offset method offers more direct access to variable-length data and might use space more efficiently but can suffer from fragmentation. The ID table method is more flexible and handles updates/deletions better but comes with the overhead of maintaining an additional structure.

	• **Variable-Length with Offset Method** (from Fig. 6.8a): This method uses a combination of fixed-length slots and a value area within the same block. The benefits include direct addressing of variable-length data via offsets, and potentially better space utilization as variable-length data can be packed closely together. However, modifying data can lead to fragmentation within the block. • **ID Tables** (from Fig. 6.8c): ID tables use a separate structure to keep track of record locations. The benefits of this method are flexibility and the ease of handling updates or deletions. It can handle variable-length records without much fragmentation since records can be reshuffled as needed. The downside is the additional overhead of maintaining the ID table and potentially more complex retrieval operations.
6.6.c.	It depends on the use-case. If the data experiences frequent updates or size changes, the ID tables might be more suitable because they can handle such modifications without much fragmentation. However, if space efficiency and direct access are more critical, and the data doesn't change size often, then the variable-length with offset method might be better.
6.7.a.	The bit array is more space-efficient for representing slot status but doesn't provide direct location information. The ID table uses more space but provides direct pointers to record locations. The bit array approach uses bits to represent the empty/in-use status of each record slot in a block. In contrast, the ID table approach from Fig. 6.8c uses values to indicate the position of the record. The bit array is more space-efficient for representing slot statuses but doesn't provide direct information about the record's location. The ID table, on the other hand, offers direct pointers to the record locations but takes up more space.
6.7.b.	First, let's find out how many records can be stored in a 4K block: 4K = 4096 bytes Maximum records = 4096 bytes / 15 bytes/record = 273.07 Round down, we get 273 records. Each integer (assuming 4 bytes) has 32 bits. Therefore, the number of integers required to store the bit array for 273 records is: Number of integers = 273 bits / 32 bits/integer = 8.53 Round up, we get 9 integers.
6.7.c.	package main import ("fmt") func findEmptySlot(bitArray []uint32) int {

```
        bitsPerInteger := 32

        for i := 0; i < len(bitArray); i++ {
                if bitArray[i] != 0xFFFFFFFF { // Not all ones
                        for j := 0; j < bitsPerInteger; j++ {
                                if (bitArray[i] & (1 << uint(j))) == 0 {
                                        return i*bitsPerInteger + j
                                }
                        }
                }
        }
        return -1 // No empty slot found
}
```

6.7.d.

```
func findNextNonEmpty(bitArray []uint32, startPosition int) int {
        bitsPerInteger := 32

        startInteger := startPosition / bitsPerInteger
        startBit := startPosition % bitsPerInteger

        for i := startInteger; i < len(bitArray); i++ {
                var startingBit int
                if i == startInteger {
                        startingBit = startBit
                } else {
                        startingBit = 0
                }
                for j := startingBit; j < bitsPerInteger; j++ {
                        if (bitArray[i] & (1 << uint(j))) != 0 {
                                return i*bitsPerInteger + j
                        }
                }
        }
        return -1 // No next non-empty record found
}

func main() {
        // Example usage
        bitArray := []uint32{0xFFFFFFFF, 0xFFFFFFFE, 0xFFFFFFFF}
        emptySlot := findEmptySlot(bitArray)
        fmt.Printf("The empty slot is at position: %d\n", emptySlot)

        nextNonEmpty := findNextNonEmpty(bitArray, 33)
```

```
        fmt.Printf("The next non-empty record from position 33 is at: %d\n",
nextNonEmpty)
}
```