Question 1

Consider a 3.5 inch diskette with 2 magnetic surfaces with 64 tracks per surface, rotating at 3600 rpm. It has a usable capacity of 2 megabytes ($2 \times 2^{20}$ bytes). Assume 20% of each track is used as overhead (gaps). Also, assume that the usable capacity is equally distributed among the tracks.

a. What is the burst bandwidth this disk can support?

b. What is the sustained bandwidth this disk can support?

c. What is the average rotational latency?

d. Assuming the average seek time is 16 ms, what is the average time to fetch a 2-kilobyte ($2 \times 2^{10}$ bytes) sector?

Question 2

Consider a disk with the following properties:

- There are four platters providing eight surfaces.
- There are $2^{13} = 8192$ tracks per surface.
- There are (on average) $2^8 = 256$ sectors per track.
There are $2^9 = 512$ bytes per sector.

The disk rotates at 3840 rpm.

The block size is $2^{12} = 4096$ bytes.

Assume 10% of each track is used as overhead.

The time it takes the head to move n tracks is $1 + n/500$ milliseconds.

Suppose that we know that the last I/O request accessed cylinder 3000. (Cylinders are numbered sequentially: 1, 2, ..., 8192.)

a. What is the expected (average) number of cylinders that will be traveled due to the very next I/O request to this disk?

b. What is the expected block access time for the next I/O, again given that the head is on cylinder 3000 initially?

**Question 3**

You are helping a client implement her medical application on your database system. Each patient record has 12 fields that always occur (e.g., name, patient number) and 38 fields that may or may not be relevant or known for a patient (e.g., number of children given birth to, cholesterol level). Assume that each of the optional fields is relevant or known for a particular patient with probability $p$. For all fields, values are stored in a fixed 15 bytes.

You are considering two options for storing each patient record:

(i) A **fixed format** record of size 50 * 15 bytes.

(ii) A **variable format** record where the 12 mandatory fields in a record are stored using 15 bytes each. Each optional field present in the record is stored in a **tagged format** with a 2-byte tag followed by the 15-byte attribute value.

a. What is the expected size of a record for each option? (Your answer may be a function of $p$.)

b. For what range of $p$ values is the fixed format option best?

c. Give two reasons why you may advise your client to go for the variable format option even when the value of $p$ in her application is large.

**Question 4**

Your friend Peter Parker, who is into a lot of web stuff, comes to you with a data management problem. Peter has about a billion web pages. He has extracted the data from these pages into the following three relations: **Words**, **Pages**, and **Links**. There are no duplicate entries in any of these relations.
Peter has given a distinct 8-byte identifier to each distinct word in the web pages. For example, the word “optimization”, which may appear in multiple pages, gets one distinct identifier. The relation `Words` has two attributes: `Word` and `WordID`, where `Word` is the string content of a word, and `WordID` is its 8-byte identifier.

Peter has given a distinct 8-byte identifier to each of the pages. The relation `Pages` contains two attributes: `WordID` and `PageID`, where `WordID` is the 8-byte identifier of a word, and `PageID` is the 8-byte identifier of a page that contains the word.

The relation `Links` contains two attributes: `SrcPageID` and `DestPageID`, where `SrcPageID` and `DestPageID` are 8-byte page identifiers such that the `SrcPageID` web page contains a link pointing to the `DestPageID` web page.

You need to answer THREE questions in FOUR scenarios. That is, this question is comprised of TWELVE questions in total. Note that there may not be a single correct answer for some of these questions. For all these questions, explain your reasoning clearly for whatever answer you give. Your will be graded on the correctness and clarity of your reasoning.

Here are the three questions (the four scenarios are described later):

Q1: If no indexes are allowed, what layout would you recommend to store the relations on disk?

Q2: What would be a good choice of indexes on each relation for the data layout you suggested for Q1 above?

Q3: If you have complete freedom in choosing the data layout and indexes, would your answer be the same as the one for Q2? If not, what layout and indexes would you recommend?

Here are the four scenarios:

**Scenario 1:** Peter’s database has the following properties:

- Only the `Words` and `Pages` relations are created.
- These relations are created all at once and are read-only, i.e., there are no updates.
- The only type of query that Peter wants to support on this data is one where given a word (e.g., “optimization”), you want to return the identifiers of all pages that contain that word. An example query in SQL:

```
Select P.PageID
From Words W, Pages P
Where W.Word = "optimization" and W.WordID = P.WordID
```

**Scenario 2:** Peter’s database has the following properties:

- Only the `Words` and `Pages` relations are created.
- There are insertions and deletions to the `Pages` relation. There are insertions, but no deletions, to the `Words` relation.
The only type of query that Peter wants to support on this data is the one in Scenario 1.

**Scenario 3:** Peter’s database has the following properties:

- All three relations are created.
- These relations are created all at once and are read-only, i.e., there are no updates.
- Peter wants to support two types of queries over this data. One query type is the one in Scenario 1. The other query type is one where given a word (e.g., “optimization”), you want to return the identifiers of all pages that link (point) to a page that contains that word. (It is okay to have duplicate page identifiers in the query result.) An example query in SQL:

```sql
Select L.SrcPageID
From Words W, Pages P, Links L
```

**Scenario 4:** Peter’s database has the following properties:

- All three relations are created.
- There are insertions and deletions to the `Pages` and `Links` relations. There are insertions, but no deletions, to the `Words` relation.
- Peter wants to support the two types of queries from Scenario 3 over this data.

**Question 5**

The following subquestions use the same conventions and notations used in Section 13.3 of the textbook. For each subquestion, state only the required answer; please do not include explanations.

A. Consider the portion of B-Tree shown in Figure 1. What is the permissible range of values for X and Y?

B. Consider the B-Tree shown in Figure 2. What is the maximum number of keys that can be inserted into the B-Tree without necessitating the addition of a new level?

C. What is the minimum number of key insertions that causes a new level to be introduced in the B-Tree of Figure 2? Give an example insertion sequence having the minimum number of keys that causes a new level to be added.

D. Consider the portion of B-Tree shown in Figure 3. Delete key 62 and update the B-Tree so that only the three nodes shown in Figure 3 are modified. Show the state of the three nodes after the deletion.

E. We start with an empty B-Tree (with a reasonably large n, around 100 say) and insert N (N ≫ n) keys in sorted order. What can you say about the space utilization of the resulting B-Tree?
We are given a large relation $R(A, B, \ldots)$ with 100 million tuples. Attribute $A$ is a primary key for the relation. The relation is static, meaning there are no updates, insertions, or deletions to the relation. The only operation over the relation is to search for tuples of $R$ having a specified value of attribute $A$. Since $A$ is a key, there can be 0 or 1 tuples in $R$ having the specified value. Our task is to perform this operation as fast as possible.

We do not have control over how tuples of $R$ are stored on the disk. In other words, our solution cannot make any assumptions about the storage of $R$. But each tuple of $R$ is stored in a contiguous region of a single disk block. Therefore, we can maintain pointers to tuples of $R$, if we chose to.

Values of attribute $A$ are drawn from an ordered domain. We are only permitted to perform the basic comparison operations over values of attribute $A$, i.e., if $a_1$ and $a_2$ are values of attribute $A$, then we can perform the boolean operations $(a_1 \text{ Op } a_2)$, where $\text{Op} \in \{<, >, =, \leq, \geq, \neq\}$. In particular, we are not permitted to compute hashes over attribute $A$ values. (We will later relax this requirement.)

We have access to a limited main memory (specified later), but nearly unlimited disk space.

**Question 6**  
**Points 25 = 10 + 10 + 5**
Leaf Level:

Figure 3: Part of a B-Tree with $n = 5$

(around a few tens of GB). The disk characteristics are as follows:

- A random seek takes 10 ms (milli-seconds) on average.
- Size of a disk block is 4096 bytes.
- The sustained transfer rate is 50 MB/sec. Therefore, once the head is over a block, we can read the block in about 0.08 ms. For simplicity, we ignore gaps: so reading two contiguous blocks takes $2 \times 0.08$, reading three takes $3 \times 0.08$, and so on.

Finally, we need 8 bytes to store a value of attribute $A$, and 8 bytes to store a disk-based pointer (for simplicity, we assume both record pointers and block pointers require the same 8 bytes).

Clearly, we need to build an auxiliary index-like structure to support the search operation. One obvious approach to the problem is to build one of the indexes that we have studied (e.g., B-Tree). But most of these are general purpose indexes, which may not be best for our specialized task.

A. Design an index structure that optimizes the search operation over $R$. Characterize its performance in terms of the time required to perform each search operation. For this part of the problem, design a solution that is completely disk-based, i.e., you are not allowed to use main memory to hold parts of the index.

B. Suppose you have access to 512 MB of main memory. You can now store a portion of your index in main memory to speed up search. Design an index for this case, and characterize its performance.

C. For case A (where the index is completely disk-based), suppose we remove the requirement that we cannot hash. (I.e., now you could build a hash-based index if you chose to.) Would your solution change? If so, describe your new index structure and characterize its performance.

For all the cases above, assume you have sufficient memory for performing I/O operations. In your solutions, clearly state all your assumptions. Some portion of credit for clear and concise descriptions.

**Question 7**

**Points 20 = 5 x 4**

The following subquestions use the same definitions, notations, and conventions used in Sections 13.4 of the textbook. For each subquestion, state only the required answer; please do not include explanations. Assume that all indexes are constructed over distinct keys (i.e., there are no duplicates),
and that the hash function ensures that there are no collisions (i.e., two keys hash to different hash values).

A. For an extensible hash table, what is the minimum number of insertions that can cause the size of the directory (array of pointers to blocks) to increase from 4 \((i = 2)\) to 16 \((i = 4)\). Assume each data block can contain 100 index keys.

B. Figure 4 conceptually represents different configurations of the directory in an extensible hash table. (The small squares represent hash data blocks.) For each of the five configurations \((I) – (V)\) indicate if the configuration can occur in a valid extensible hash table. For simplicity, assume no deletes.

![Figure 4: Configurations of the directory in a extensible hash table](image)

C. Figure 5 shows the contents of all the data buckets for five different instantiations of some hash-based index. For each of the five cases, indicate if the contents could have arisen in an extensible hash table.

D. For each case \((I) – (V)\) of Figure 5, indicate if the contents of the data buckets could have arisen in a linear hash table.
Figure 5: Contents of all data buckets in a hash-based index