Here are two similar exercise sets sheet 7 lecture 5 with adjusted numbers:

Implementation of DBMS

Exercise Sheet 7 *WS 2024 / 2025*

1) Suppose that blocks can hold either **four records**, **15 key-pointer pairs**, or **60 pointers**.

Questions:
a) We use the indirect bucket scheme. If each search-key value appears in **20 records**, how many blocks do we need to hold **6000 records** and its secondary index structure? How many blocks would we need if we did not use buckets but a key-pointer pair for each record?

b) We want to retrieve all records for a particular search key. In doing so, we sequentially scan the key-pointer pairs in the index from the beginning until we find the search key value we are looking for. We assume that records with the same search key never share a block. We also assume that buckets never extend across different blocks and that key-pointer pairs with the same key are all in the same block. How many blocks do you have to inspect on average for the two scenarios in **a**?

---

2) We would like to sort the tuples of a relation **R(X, Y, Z)**. Initially, the tuples are stored in secondary storage in a random order. At the end, they have to be in secondary storage sorted according to the values of attribute **X** (search key).

The following information is known about the relation:

- The relation **R** has **120,000 tuples**.
- The size of a block is **8192 bytes**. Blocks have a header of **80 bytes**.
- The sizes of attributes are **40 bytes** for **X**, **160 bytes** for **Y**, and **120 bytes** for **Z**. Records have a header of **32 bytes**.
- Each block holding **R** tuples is full of as many **R** tuples as possible. We use unspanned storage for the records and the key-pointer pairs in **c**.
- A record pointer is **10 bytes**.

Questions:
a) If we use the 2PMMS sorting algorithm with two passes, what is the minimum amount of main memory (in number of blocks) required?

b) What is the cost of the two-pass sorting algorithm in terms of the number of I/Os?

c) Consider the following variant of the sorting algorithm. Instead of sorting the entire tuples, we just sort the pairs <key, recordPointer> for each tuple. As in the conventional two-pass sorting algorithm, we sort chunks of <key, recordPointer> in main memory and write the chunks to disk. In the merge phase, <key, recordPointer> entries from different chunks are merged. The record pointers are used to recover the rest of the tuple (from the original copy of **R**) and write the sorted relation to the disk. What is the cost in terms of the number of I/Os?

Implementation of DBMS

Exercise Sheet 7, *WS 2024 / 2025*

1) Suppose that blocks can hold either **five records**, **12 key-pointer pairs**, or **40 pointers**.

Questions:

a) We use the indirect bucket scheme. If each search-key value appears in **15 records**, how many blocks do we need to hold **4500 records** and its secondary index structure? How many blocks would we need if we did not use buckets but a key-pointer pair for each record?

b) We want to retrieve all records for a particular search key. In doing so, we sequentially scan the key-pointer pairs in the index from the beginning until we find the search key value we are looking for. We assume that records with the same search key never share a block. We also assume that buckets never extend across different blocks and that key-pointer pairs with the same key are all in the same block. How many blocks do you have to inspect on average for the two scenarios in **a**?

---

2) We would like to sort the tuples of a relation **R(M, N, O)**. Initially, the tuples are stored in secondary storage in a random order. At the end, they have to be in secondary storage sorted according to the values of attribute **M** (search key).

The following information is known about the relation:

- The relation **R** has **80,000 tuples**.
- The size of a block is **4096 bytes**. Blocks have a header of **40 bytes**.
- The sizes of attributes are **50 bytes** for **M**, **180 bytes** for **N**, and **110 bytes** for **O**. Records have a header of **24 bytes**.
- Each block holding **R** tuples is full of as many **R** tuples as possible. We use unspanned storage for the records and the key-pointer pairs in **c**.
- A record pointer is **6 bytes**.

Questions:

a) If we use the 2PMMS sorting algorithm with two passes, what is the minimum amount of main memory (in number of blocks) required?

b) What is the cost of the two-pass sorting algorithm in terms of the number of I/Os?

c) Consider the following variant of the sorting algorithm. Instead of sorting the entire tuples, we just sort the pairs <key, recordPointer> for each tuple. As in the conventional two-pass sorting algorithm, we sort chunks of <key, recordPointer> in main memory and write the chunks to disk. In the merge phase, <key, recordPointer> entries from different chunks are merged. The record pointers are used to recover the rest of the tuple (from the original copy of **R**) and write the sorted relation to the disk. What is the cost in terms of the number of I/Os?

---

a) n >= 85

b) 29,092 I/Os

c) 96,522 I/Os

Similar New Exercise

Implementation of DBMS
Exercise Sheet 7

1) Suppose that blocks can hold either five records, 20 key-pointer pairs, or 80 pointers.

Questions:
a) We use the indirect bucket scheme. If each search-key value appears in 25 records, how many blocks do we need to hold 7500 records and its secondary index structure? How many blocks would we need if we did not use buckets but a key-pointer pair for each record?

b) We want to retrieve all records for a particular search key. In doing so, we sequentially scan the key-pointer pairs in the index from the beginning until we find the search key value we are looking for. We assume that records with the same search key never share a block. We also assume that buckets never extend across different blocks and that key-pointer pairs with the same key are all in the same block. How many blocks do you have to inspect on average for the two scenarios in a?

2) We would like to sort the tuples of a relation R(A,B,C). Initially, the tuples are stored in secondary storage in a random order. At the end, they have to be in secondary storage sorted according to the values of attribute A (search key).

The following information is known about the relation:

- The relation R has 150,000 tuples.
- The size of a block is 4,096 bytes. Blocks have a header of 64 bytes.
- The sizes of attributes are 20 bytes for A, 120 bytes for B, and 100 bytes for C. Records have a header of 16 bytes.
- Each block holding R tuples is full of as many R tuples as possible. We use unspanned storage for the records and the key-pointer pairs in c.
- A record pointer is 8 bytes.

Questions:
a) If we use the 2PMMS sorting algorithm with two passes, what is the minimum amount of main memory (in number of blocks) required?

b) What is the cost of the two-pass sorting algorithm in terms of the number of I/Os?

c) Consider the following variant of the sorting algorithm. Instead of sorting the entire tuples, we just sort the pairs <key,recordPointer> for each tuple. As in the conventional two-pass sorting algorithm, we sort chunks of <key,recordPointer> in main memory and write the chunks to disk. In the merge phase, <key,recordPointer> entries from different chunks are merged. The record pointers are used to recover the rest of the tuple (from the original copy of RRR) and write the sorted relation to the disk. What is the cost in terms of the number of I/Os?

Question 1a:

1. Suppose that blocks can hold either **4 records**, **12 key-pointer pairs**, or **60 pointers**.

   a) We use the indirect bucket scheme. If each search-key value appears in **15 records**, how many blocks do we need to hold **4500 records** and its secondary index structure? How many blocks would we need if we did not use buckets but a key-pointer pair for each record?
   b) We want to retrieve all records for a particular search key. In doing so, we sequentially scan the key-pointer pairs in the index from the beginning until we find the search key value we are looking for. We assume that records with the same search key never share a block. We also assume that buckets never extend across different blocks and that key-pointer pairs with the same key are all in the same block. How many blocks do you have to inspect on average, for the two scenarios in a)?

Question 1b:

2. Suppose that blocks can hold either **5 records**, **15 key-pointer pairs**, or **75 pointers**.

   a) We use the indirect bucket scheme. If each search-key value appears in **20 records**, how many blocks do we need to hold **6000 records** and its secondary index structure? How many blocks would we need if we did not use buckets but a key-pointer pair for each record?
   b) We want to retrieve all records for a particular search key. In doing so, we sequentially scan the key-pointer pairs in the index from the beginning until we find the search key value we are looking for. We assume that records with the same search key never share a block. We also assume that buckets never extend across different blocks and that key-pointer pairs with the same key are all in the same block. How many blocks do you have to inspect on average, for the two scenarios in a)?

Question 1c:

3. Suppose that blocks can hold either **6 records**, **18 key-pointer pairs**, or **90 pointers**.

   a) We use the indirect bucket scheme. If each search-key value appears in **25 records**, how many blocks do we need to hold **7500 records** and its secondary index structure? How many blocks would we need if we did not use buckets but a key-pointer pair for each record?
   b) We want to retrieve all records for a particular search key. In doing so, we sequentially scan the key-pointer pairs in the index from the beginning until we find the search key value we are looking for. We assume that records with the same search key never share a block. We also assume that buckets never extend across different blocks and that key-pointer pairs with the same key are all in the same block. How many blocks do you have to inspect on average, for the two scenarios in a)?

Question 2:

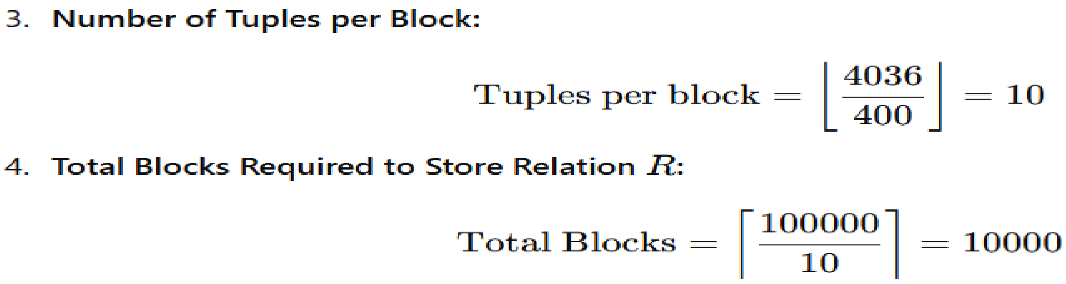
c) Consider the following variant of the sorting algorithm. Instead of sorting the entire tuples, we just sort the pairs for each tuple. As in the conventional two pass sorting algorithm, we sort chunks of in main memory and write the chunks to disk. In the merge phase, entries from different chunks are merged. The record pointers are used to recover the rest of the tuple (from the original copy of R) and write the sorted relation to the disk. What is the cost in terms of number of I/Os?

Detailed and Precise Explanation of the Solution

We need to sort the tuples of relation R(A,B,C) using the **Two-Pass Multiway Merge Sort (2PMMS)** algorithm. Given the problem constraints, we will analyze each part of the question step by step.

Step 1: Understanding the Given Data

1. **Relation Size and Tuple Structure:**
   - Each tuple consists of attributes A,B,CA, B, CA,B,C and a record header.
   - Sizes:
     - A=32 bytes
     - B=200 bytes
     - C=140 bytes
     - Record header = 28 bytes
   - **Total tuple size** = 28+32+200+140=400 bytes
2. **Block Size and Storage Constraints:**
   - Block size = **4096 bytes**
   - Block header = **60 bytes**
   - Available space per block = 4096−60=4036 bytes
   - Since we use **unspanned storage**, each block can only store complete tuples.



3. **Number of Tuples per Block:**

$$\text{Tuples per block} = \left\lfloor \frac{4036}{400} \right\rfloor = 10$$

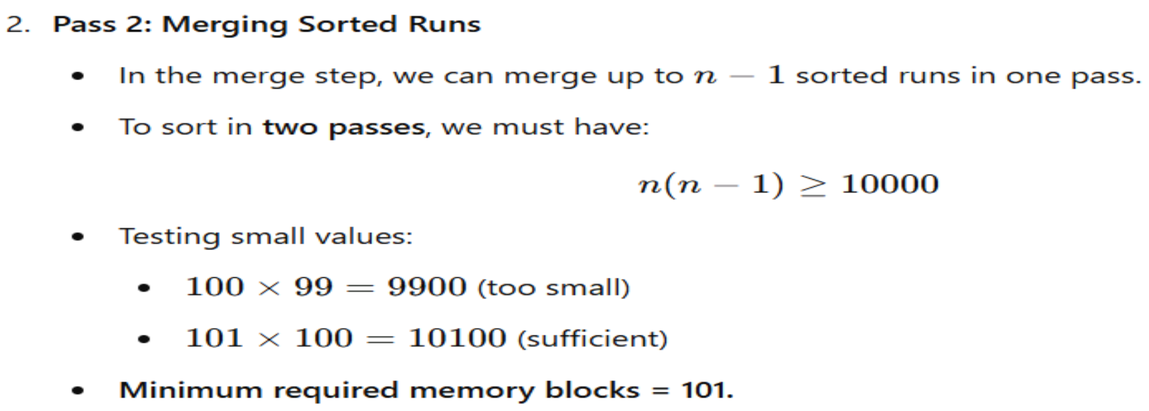
4. **Total Blocks Required to Store Relation R:**

$$\text{Total Blocks} = \left\lceil \frac{100000}{10} \right\rceil = 10000$$

Part (a): Minimum Memory Blocks for 2PMMS

The **Two-Pass Multiway Merge Sort (2PMMS)** sorts a relation in two passes:

1. **Pass 1: Sorting Initial Runs**
   - ○ With nnn memory blocks, we can sort nnn blocks at a time.
   - ○ This produces ⌈10000/n⌉ sorted runs.



2. **Pass 2: Merging Sorted Runs**
   - In the merge step, we can merge up to $n - 1$ sorted runs in one pass.
   - To sort in **two passes**, we must have:

$$n(n-1) \geq 10000$$

   - Testing small values:
     - $100 \times 99 = 9900$ (too small)
     - $101 \times 100 = 10100$ (sufficient)
   - **Minimum required memory blocks = 101.**

Part (b): Cost of 2PMMS in Terms of I/Os

The **I/O cost** includes:

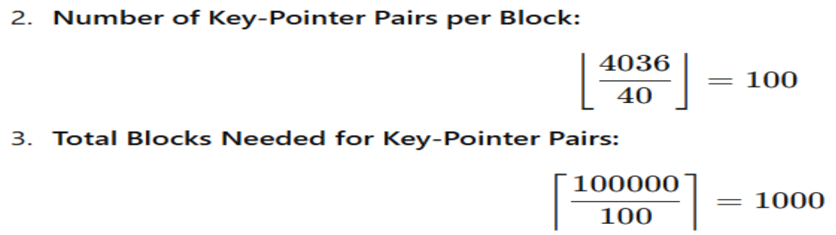
1. **Pass 1: Reading and Sorting the Blocks**
   - ○ Read all 10000 blocks: **10000 I/Os**
   - ○ Write back sorted runs: **10000 I/Os**
2. **Pass 2: Merging Sorted Runs and Writing Final Output**
   - ○ Read all 10000 blocks: **10000 I/Os**
   - ○ Write back the final sorted relation: **10000 I/Os**

Total I/Os =10000+10000+10000+10000=40000

Part (c): Sorting Key-Pointer Pairs Instead of Full Tuples

Instead of sorting the full tuples, we sort **(key, record pointer) pairs** and use the record pointers to retrieve full tuples later.

1. **Key-Pointer Pair Size:**
   - ○ Key A=32 bytes
   - ○ Record pointer = 8 bytes
   - ○ **Total size** = 32+8=40 bytes



2. **Number of Key-Pointer Pairs per Block:**

$$\left\lfloor \frac{4036}{40} \right\rfloor = 100$$

3. **Total Blocks Needed for Key-Pointer Pairs:**

$$\left\lceil \frac{100000}{100} \right\rceil = 1000$$

2. **Sorting Costs in Terms of I/Os:**
   - **Step 1:** Read relation R and construct key-pointer pairs → **10000 I/Os**
   - **Step 2:** Write the key-pointer sorted sublists to disk → **1000 I/Os**
   - **Step 3:** Read the sorted sublists back from disk → **1000 I/Os**
   - **Step 4:** Retrieve full tuples using pointers (one I/O per tuple) → **100000 I/Os**
   - **Step 5:** Write the final sorted relation back to disk → **10000 I/Os**

Total I/Os=10000+1000+1000+100000+10000=122000

Summary of Results

Part	Description	Total I/Os
(a)	Minimum memory blocks required for 2PMMS	**101**
(b)	Standard 2PMMS sorting I/O cost	**40000**
(c)	Sorting key-pointer pairs instead of full tuples	**122000**

Sorting just **key-pointer pairs** increases I/O cost significantly because retrieving full tuples after sorting adds **100000 I/Os**.

Key Takeaways

- **Sorting the full relation** using **2PMMS** is more efficient (40000 I/Os).
- **Sorting key-pointer pairs** may save memory but incurs **extra retrieval costs**, making it much more expensive (122000 I/Os).
- **More main memory blocks (higher n) reduce the number of sorted runs** and lower I/O costs.

Key Features of These Questions:

1. **Block Structure:**
   - Each block can hold a fixed number of records, key-pointer pairs, or pointers.
2. **Indexing Schemes:**
   - The indirect bucket scheme is used in one scenario, while a direct key-pointer scheme is used in the other.
3. **Data Distribution:**
   - Each search-key value appears in a fixed number of records (e.g., 10, 15, 20, 25).
4. **Retrieval Process:**

- The questions involve calculating the number of blocks to inspect on average when retrieving all records for a given search key.

The exercises touch on several core concepts in the implementation of **Database Management Systems (DBMS)**. Here are detailed explanations of the concepts used in these questions, along with examples:

---

1. Indirect Bucket Scheme

Concept:

The **indirect bucket scheme** is a storage technique used for secondary indexes. Instead of storing one key-pointer pair for each record in the index, **buckets** group pointers for records with the same key. This reduces the size of the index, as multiple records with the same key share a single entry in the index.

Key Points:

- **Key-pointer pair index:** Each record has its own key-pointer pair in the index.
- **Bucketed index:** All records with the same key are grouped under one entry in the index, with a pointer to a block (bucket) that contains the pointers for the records.

Example:

Assume:

- A block can hold 3 records, 10 key-pointer pairs, or 50 pointers.
- There are 3000 records, and each key appears in 10 records.

Without Buckets:

- Each key-pointer pair takes a slot in the index.
- Total key-pointer pairs = Total records = 3000.
- Blocks required for key-pointer pairs:

$$\lceil \frac{3000}{10} \rceil = 300 \text{ blocks.}$$

With Buckets:

- Each unique key (appearing in 10 records) occupies **one key-pointer pair**, and a bucket holds pointers for those 10 records.
- Total unique keys = $\frac{3000}{10} = 300.$
- Blocks for key-pointer pairs = $\lceil \frac{300}{10} \rceil = 30.$
- Each bucket holds 50 pointers.
  Buckets required = $\lceil \frac{10 \times 300}{50} \rceil = 60 \text{ blocks.}$
- Total blocks: $30 + 60 = 90.$

Using buckets reduces the total storage cost from **300 blocks** to **90 blocks**.

2. Sorting Algorithms and 2PMMS (Two-Phase Multi-Way Merge Sort)

Concept:

Sorting is essential for tasks like creating sorted indexes or answering range queries efficiently. **Two-Phase Multi-Way Merge Sort (2PMMS)** is commonly used to sort large datasets that do not fit into memory.

Two Phases:

1. **Phase 1 (Partitioning):**
   - Divide the data into chunks that fit into memory.
   - Sort each chunk in memory and write it back to disk as a sorted "run."
2. **Phase 2 (Merging):**
   - Merge sorted runs in memory. Each merge reduces the number of runs until only one sorted run remains.

Example:

Assume:

- Relation **R** has 100,000 tuples.
- Block size = 4096 bytes.
- Tuple size = Header (28 bytes) + A (32 bytes) + B (200 bytes) + C (140 bytes).
  Total tuple size = $28 + 32 + 200 + 140 = 400$ bytes.
- Each block can hold:
$$\left\lfloor \frac{4096 - 60}{400} \right\rfloor = 10 \text{ tuples.}$$
- Total blocks for **R**:
$$\lceil \frac{100,000}{10} \rceil = 10,000 \text{ blocks.}$$

Phase 1:

- If memory can hold M blocks, we process M blocks at a time.
  Number of sorted runs:
$$\lceil \frac{\text{Total blocks}}{\text{Memory blocks}} \rceil.$$

Phase 2:

- Merge all runs.
  I/O cost for Phase 1 + Phase 2 = $2 \times (\text{Total blocks}) \times (\text{Number of passes})$.

Variant:

Sorting only `<key, recordPointer>` pairs reduces the size of data processed.

- Pair size = Key (32 bytes) + Pointer (8 bytes) = **40 bytes**.
- Each block can hold:
  $$\left\lfloor \frac{4096 - 60}{40} \right\rfloor = 100 \text{ pairs.}$$
- Total blocks for key-pointer pairs:
  $$\lceil \frac{100,000}{100} \rceil = 1,000 \text{ blocks.}$$

Sorting **key-pointer pairs** requires fewer blocks and thus fewer I/Os compared to sorting full tuples.

3. Index Types: Dense vs Sparse Index

Concept:

Indexes are data structures that speed up query processing by reducing the amount of data to be searched.

Dense Index:

- Contains a key-pointer pair for every record in the data file.
- Advantage: Fast lookup for any record.
- Disadvantage: Larger index size.

Sparse Index:

- Contains key-pointer pairs only for the first record in each block.
- Advantage: Smaller index size.
- Disadvantage: Slower lookups since additional I/O is required to find the exact record.

Example:

Assume:

- Blocks hold 10 records.
- Data file contains 10^6 records.

Dense Index:

- Total key-pointer pairs = 10^6.
- Blocks for key-pointer pairs:

$$\lceil \frac{10^6}{50} \rceil = 20,000 \text{ blocks.}$$

Sparse Index:

- Total blocks in the data file:

$$\lceil \frac{10^6}{10} \rceil = 100,000 \text{ blocks.}$$

- Key-pointer pairs = Number of data blocks = 100,000.
- Blocks for key-pointer pairs:

$$\lceil \frac{100,000}{50} \rceil = 2,000 \text{ blocks.}$$

Sparse indexes are much smaller, but they may require additional I/O to find the exact record.

4. Disk I/O for Record Retrieval

Concept:

Disk I/O is a measure of the number of disk operations required to retrieve data. The goal is to minimize disk I/O for efficient performance.

Scenarios:

a) **Sequential Scan:**

- Search the data file from the beginning until the record is found.
- Average I/Os = Half the total blocks in the data file.
  For 10^6 records (100,000 blocks), I/Os = $100,000/2 = 50,000$.

b) **Using Dense Index:**

- Scan the index sequentially until the key is found, then read the data block.
  Index I/Os = Half the index blocks.
  Data file I/Os = 1.
  Total I/Os = Index I/Os + 1.

c) **Using Multi-Level Index:**

- For each level, one block is read to locate the pointer to the next level.
  Final level points to the data block.
  Total I/Os = Number of index levels + 1.

Example:

Assume:

- Dense index has 20,000 blocks.
- Sparse index has 2,000 blocks.

Dense Index:

- Multi-level structure: 20,000 → 400 → 8 → 1.

  Total levels = 4.

  Total I/Os = $4 + 1 = 5$.

Sparse Index:

- Multi-level structure: 2,000 → 40 → 2 → 1.

  Total levels = 3.

  Total I/Os = $3 + 1 = 4$.

Sparse indexes require fewer I/Os due to fewer levels.

Sorting Algorithms and 2PMMS (Two-Phase Multi-Way Merge Sort)

Sorting is one of the fundamental operations in database systems, enabling efficient retrieval, processing, and organization of data. When the size of the data exceeds the available main memory, external sorting algorithms like the **Two-Phase Multi-Way Merge Sort (2PMMS)** are used.

---

1. Two-Phase Multi-Way Merge Sort (2PMMS)

2PMMS is an **external sorting algorithm** designed for large datasets stored in secondary storage (e.g., disks). The algorithm operates in two distinct phases: **partitioning (sorting in memory)** and **merging sorted runs**.

Assumptions

- Data is too large to fit entirely in main memory.
- Data is stored in blocks on secondary storage.
- Disk I/O is the primary cost, so the algorithm is optimized to minimize read/write operations.

2. How It Works

Phase 1: Partitioning (Sorting Runs)

1. **Load data chunks into memory**:
   Read as many blocks of data as the main memory can hold.
   Example: If memory holds 10 blocks and the dataset has 1000 blocks, process the data in chunks of 10 blocks at a time.
2. **Sort each chunk in memory**:
   Use an efficient in-memory sorting algorithm (e.g., quicksort or heapsort).
3. **Write sorted runs back to disk**:
   Write each sorted chunk (called a "run") back to secondary storage.

   Example: For a dataset of 1000 blocks, with memory holding 10 blocks at a time, the first pass creates $\frac{1000}{10} = 100$ sorted runs, each of 10 blocks.

Phase 2: Merging Sorted Runs

1. **Load portions of runs into memory**:
   Use a priority queue or heap to merge runs. If there are k runs, the algorithm merges k-way.
   The memory holds:
   - k blocks (one from each run to be merged).
   - A buffer to write the merged output.

2. **Merge runs into larger sorted runs**:
   Compare the smallest elements across the k runs and write the smallest to the output buffer.
   Repeat until all runs are merged into a single sorted output.

   Example:

   - First merge: 100 runs → 10 larger runs (each of 100 blocks).
   - Second merge: 10 larger runs → 1 final sorted run (1000 blocks).

3. Cost of 2PMMS

The **cost** of 2PMMS is measured in terms of disk I/O operations (reads and writes).

- Each block is read and written in **Phase 1** (1 read + 1 write).
- Each block is read and written in **Phase 2** during the merge steps.

If there are N blocks in total, and memory can hold M blocks:

- **Number of passes**:

$$1 + \lceil \log_M \lceil \frac{N}{M} \rceil \rceil$$

  - The first pass creates sorted runs.
  - Subsequent passes merge runs until one sorted run remains.

- **Total I/Os**:

$$2N \times (\text{Number of passes}).$$

4. Example of 2PMMS

Dataset:

- Relation $R(A, B, C)$ with $100,000$ tuples.
- Tuple size = 400 bytes.
- Block size = 4096 bytes (header = 60 bytes).

Steps:

1. **Phase 1: Sorting Runs**
   - Each block holds:
$$\left\lfloor \frac{4096 - 60}{400} \right\rfloor = 10 \text{ tuples per block.}$$
   - Total blocks for $100,000$ tuples:
$$\lceil \frac{100,000}{10} \rceil = 10,000 \text{ blocks.}$$
   - If memory holds $M = 100$ blocks, the first pass creates:
$$\lceil \frac{10,000}{100} \rceil = 100 \text{ sorted runs.}$$

2. **Phase 2: Merging Runs**
   - Merge $M - 1 = 99$ runs at a time (1 block reserved for output buffer).
   - First merge: 100 runs → $\lceil \frac{100}{99} \rceil = 2 \text{ larger runs.}$
   - Second merge: 2 larger runs → 1 final sorted run.
3. **Total I/Os**
   - Phase 1: $2 \times 10,000 = 20,000$ I/Os.
   - Phase 2 (2 passes):
   $$2 \times 10,000 \times 2 = 40,000 \text{ I/Os.}$$
   - Total I/Os: $20,000 + 40,000 = 60,000$.

5. Variants of 2PMMS

Sorting Key-Pointer Pairs Instead of Full Tuples

- Instead of sorting full tuples, only sort `<key, recordPointer>` pairs.
- This reduces the amount of data processed.

Example:

- Key size = 32 bytes, pointer size = 8 bytes → Pair size = 40 bytes.
- Each block can hold:

$$\left\lfloor \frac{4096 - 60}{40} \right\rfloor = 100 \text{ pairs.}$$

- Total blocks for $100,000$ tuples:

$$\lceil \frac{100,000}{100} \rceil = 1,000 \text{ blocks.}$$

- Sorting 1,000 blocks instead of 10,000 blocks greatly reduces I/O costs.

6. Applications of 2PMMS

- **Sorting Large Relations**: For creating sorted indexes.
- **Joins**: When relations are sorted for merge joins.
- **Deduplication**: Sorting helps identify duplicates efficiently.
- **Grouping**: Sorting assists in grouping data for aggregate operations.

---

7. Real-World Example

Scenario:

A company's sales database contains 1 billion sales records, and they want to sort them by date.

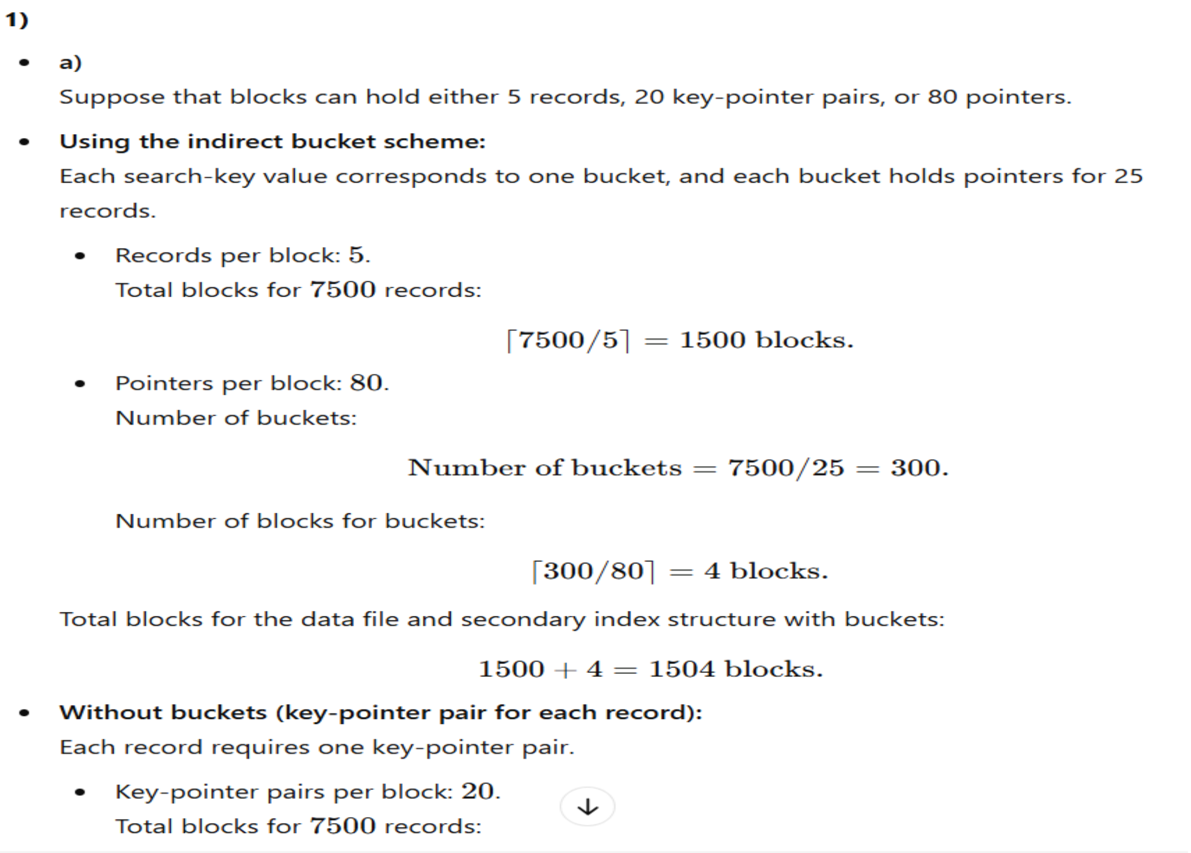
- **Assumptions**:
  - Each record is 500 bytes.
  - Block size = 4 KB.
  - Memory can hold 100 MB = 25,600 blocks.

- **Steps**:
  1. Phase 1: Sort $\frac{1B\times 500}{4K} = 125M \text{ blocks}$ in chunks of 25,600 blocks → 4,882 runs.
  2. Phase 2: Merge 25,599 runs per pass.
     Pass 1: 4,882 runs → $\lceil \frac{4882}{25599} \rceil = 191$ runs.
     Pass 2: 191 runs → 1 final run.
- **I/O Cost**:
  Phase 1: $2 \times 125M = 250M \text{ I/Os}$.
  Phase 2: $2 \times 125M \times 2 = 500M \text{ I/Os}$.
  Total: 750M I/Os.

Key Takeaways

- **2PMMS is efficient** for external sorting because it minimizes disk I/O.
- **Chunk size matters**: Larger memory chunks result in fewer sorted runs and fewer merge passes.
- Sorting key-pointer pairs instead of full records can **save storage and processing costs**.

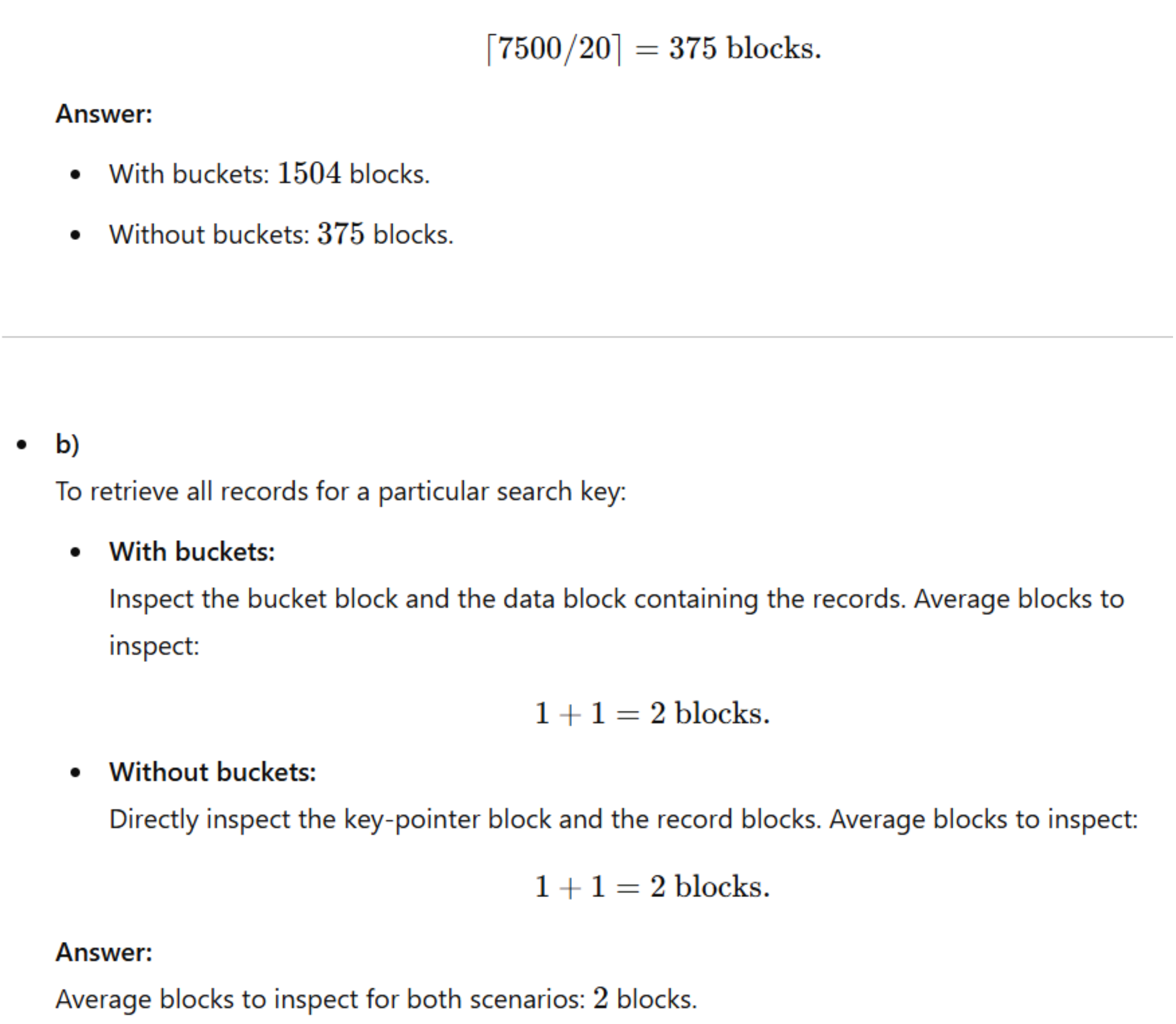
Solutions to Exercise Sheet 7



1)

- **a)**
  Suppose that blocks can hold either 5 records, 20 key-pointer pairs, or 80 pointers.
- **Using the indirect bucket scheme:**
  Each search-key value corresponds to one bucket, and each bucket holds pointers for 25 records.
  - Records per block: 5.
    Total blocks for 7500 records:
    $$\lceil 7500/5 \rceil = 1500 \text{ blocks.}$$
  - Pointers per block: 80.
    Number of buckets:
    $$\text{Number of buckets} = 7500/25 = 300.$$
    Number of blocks for buckets:
    $$\lceil 300/80 \rceil = 4 \text{ blocks.}$$

  Total blocks for the data file and secondary index structure with buckets:
  $$1500 + 4 = 1504 \text{ blocks.}$$
- **Without buckets (key-pointer pair for each record):**
  Each record requires one key-pointer pair.
  - Key-pointer pairs per block: 20.
    Total blocks for 7500 records:



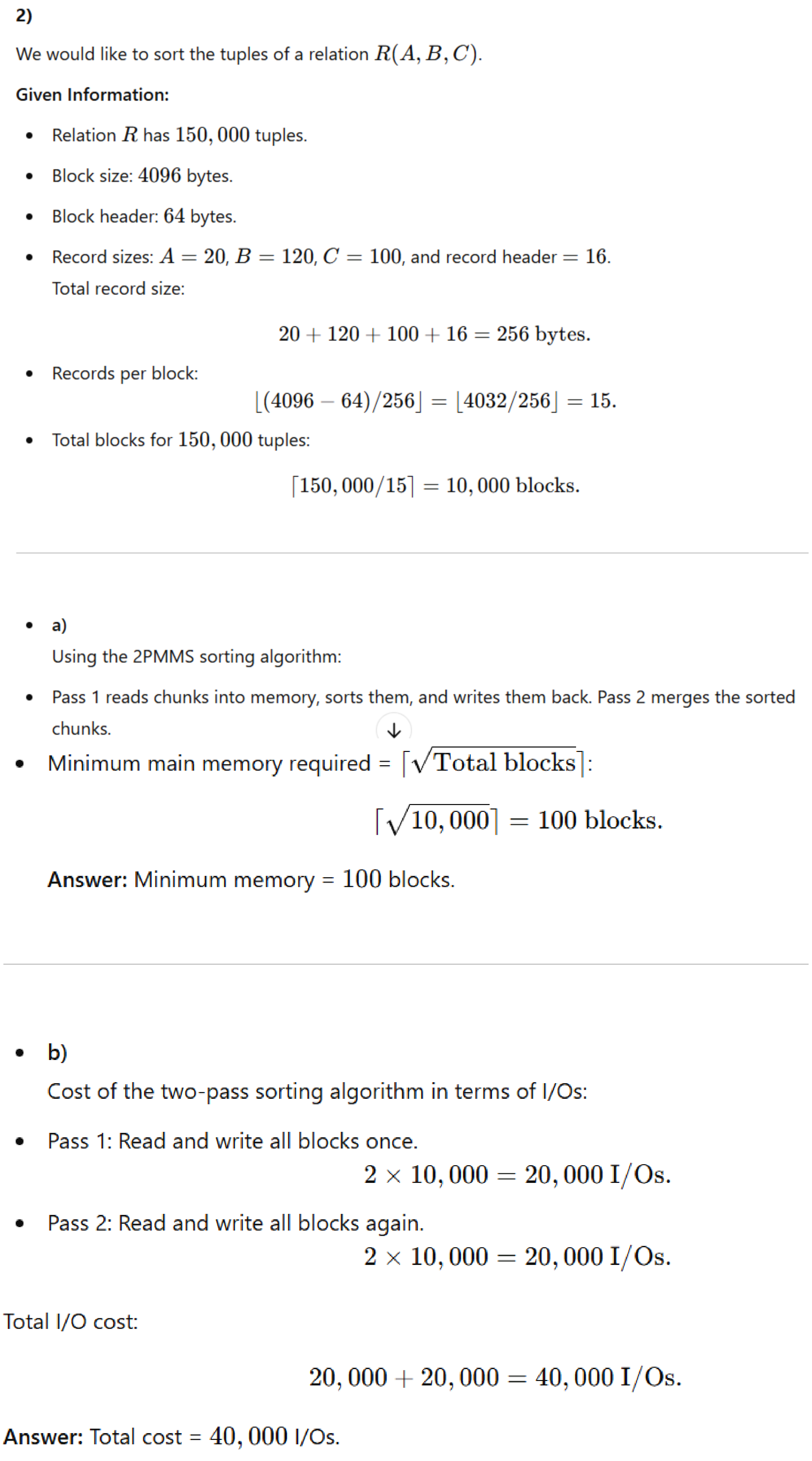
$$\lceil 7500/20 \rceil = 375 \text{ blocks.}$$

Answer:

- With buckets: 1504 blocks.
- Without buckets: 375 blocks.

- **b)**
  To retrieve all records for a particular search key:
  - **With buckets:**
    Inspect the bucket block and the data block containing the records. Average blocks to inspect:
    $$1 + 1 = 2 \text{ blocks.}$$
  - **Without buckets:**
    Directly inspect the key-pointer block and the record blocks. Average blocks to inspect:
    $$1 + 1 = 2 \text{ blocks.}$$

  Answer:
  Average blocks to inspect for both scenarios: 2 blocks.



2)

We would like to sort the tuples of a relation $R(A, B, C)$.

Given Information:

- Relation R has $150,000$ tuples.
- Block size: 4096 bytes.
- Block header: 64 bytes.
- Record sizes: $A = 20$, $B = 120$, $C = 100$, and record header $= 16$.
  Total record size:
  $$20 + 120 + 100 + 16 = 256 \text{ bytes.}$$
- Records per block:
  $$\lfloor (4096 - 64)/256 \rfloor = \lfloor 4032/256 \rfloor = 15.$$
- Total blocks for $150,000$ tuples:
  $$\lceil 150,000/15 \rceil = 10,000 \text{ blocks.}$$

- **a)**
  Using the 2PMMS sorting algorithm:
- Pass 1 reads chunks into memory, sorts them, and writes them back. Pass 2 merges the sorted chunks.
- Minimum main memory required = $\lceil \sqrt{\text{Total blocks}} \rceil$:
  $$\lceil \sqrt{10,000} \rceil = 100 \text{ blocks.}$$

  Answer: Minimum memory = 100 blocks.

- **b)**
  Cost of the two-pass sorting algorithm in terms of I/Os:
- Pass 1: Read and write all blocks once.
  $$2 \times 10,000 = 20,000 \text{ I/Os.}$$
- Pass 2: Read and write all blocks again.
  $$2 \times 10,000 = 20,000 \text{ I/Os.}$$

Total I/O cost:

$$20,000 + 20,000 = 40,000 \text{ I/Os.}$$

Answer: Total cost = $40,000$ I/Os.

- **c)**

  Sorting only $< key, recordPointer >$:

- Key size = 20, Pointer size = 8.

  Total $< key, recordPointer >$ size = 28 bytes.

- Entries per block:

$$\lfloor (4096 - 64)/28 \rfloor = \lfloor 4032/28 \rfloor = 144.$$

- Total blocks for $150,000$ tuples:

$$\lceil 150,000/144 \rceil = 1042 \text{ blocks.}$$

I/O Cost:

- Pass 1: Read original relation and write sorted $< key, recordPointer >$.

$$10,000 + 1042 = 11,042 \text{ I/Os.}$$

- Pass 2: Read and merge $< key, recordPointer >$ and retrieve original tuples.

$$2 \times 1042 + 10,000 = 12,084 \text{ I/Os.}$$

Total I/O cost:

$$11,042 + 12,084 = 23,126 \text{ I/Os.}$$

Answer: Total cost = $23,126$ I/Os.