Here is a detailed explanation of the solutions and the underlying concepts.

Q.1: Understanding the Problem

We are dealing with **secondary indexing** in a database where:

- Each **block** can hold either:
  - **3 records** (data blocks)
  - **10 key-pointer pairs** (index blocks)
  - **50 pointers** (bucket blocks)
- We have **3000 records** in total, and each **search key appears in 10 records**.
- Two indexing schemes are considered:

1. **Indirect bucket scheme**: Uses an extra level of indirection (bucket blocks).
2. **Direct indexing**: Each key-pointer pair points directly to the record.

Part (a): Calculating Storage Requirements

We need to determine how many blocks are required for the data file and the index structure.

Indirect Bucket Scheme

1. **Storing the Data File**
   - Since **each block holds 3 records**, the number of blocks required: $\lceil 3000/3 \rceil = 1000$ blocks
2. **Storing the Index (Key-Pointer Pairs)**
   - There are **300 distinct keys**, each pointing to a bucket.
   - Each **block holds 10 key-pointer pairs**, so we need: $\lceil 300/10 \rceil = 30$ blocks
3. **Storing the Buckets**
   - Each search key appears in **10 records**, so we need **3000 pointers** in total.
   - Each **bucket block holds 50 pointers**, so we need: $\lceil 3000/50 \rceil = 60$ blocks
4. **Total Blocks**
   - **Index blocks (30) + Bucket blocks (60) + Data blocks (1000) = 1090 blocks**

Direct Indexing (No Buckets)

1. **Storing the Data File**
   - Same as before: **1000 blocks**.
2. **Storing the Index (Key-Pointer Pairs)**

   - Instead of using **buckets**, we store **one key-pointer pair for each of the 3000 records**.
   - Since each block holds **10 key-pointer pairs**, we need: ⌈3000/10⌉=300 blocks
3. **Total Blocks**
   - **Index blocks (300) + Data blocks (1000) = 1300 blocks**

Comparison

Scheme	Data Blocks	Index Blocks	Bucket Blocks	Total Blocks
Indirect Buckets	1000	30	60	**1090**
Direct Indexing	1000	300	0	**1300**

Conclusion: The **indirect bucket scheme** is more space-efficient than direct indexing.

Part (b): Searching for a Specific Search Key

Now, we analyze how many blocks must be inspected **on average** to retrieve all records for a given search key.

Indirect Bucket Scheme

1. **Searching the Key in Index Blocks**
   - There are **30 index blocks** containing key-pointer pairs.
   - Since the key could be anywhere, we assume a uniform distribution.
   - On average, we inspect: (1+30)/2=15.5 blocks
2. **Accessing the Corresponding Bucket Block**
   - The pointer from the index block points to **exactly 1 bucket block**.
3. **Retrieving Records from Data Blocks**
   - Each search key corresponds to **10 records**, each in a different block.
   - So, we inspect **10 data blocks**.
4. **Total Blocks Inspected**

15.5+1+10=26.5 blocks

Direct Indexing Scheme

1. **Searching the Key in Index Blocks**
   - There are **300 index blocks**.
   - On average, we inspect: (1+300)/2=150.5 blocks

2. **Retrieving Records from Data Blocks**
    - Since we store **one key-pointer pair per record**, we directly retrieve the **10 data blocks**.
3. **Total Blocks Inspected**

150.5+10=160.5 blocks

Comparison

Scheme	Avg. Index Blocks Inspected	Bucket Blocks	Data Blocks	Total Blocks
Indirect Buckets	**15.5**	1	10	**26.5**
Direct Indexing	**150.5**	0	10	**160.5**

Conclusion:
The **indirect bucket scheme** is **much faster** than direct indexing because it reduces the number of index blocks that need to be inspected.

Key Concepts Explained

1. Block Storage

- **Records per block:** Determines how many records can fit into a storage block.
- **Key-pointer pairs per block:** Determines how many search keys and their corresponding pointers fit into an index block.
- **Pointers per bucket block:** Determines how many record pointers a bucket can hold.

2. Secondary Indexing

- **Secondary indexes** allow searching by a non-primary attribute.
- **Direct indexing:** Each key-pointer pair directly references a record.
- **Indirect bucket scheme:** Uses an extra level (buckets) to group records with the same key.

3. Search Efficiency

- **Sequential search in index blocks:** Average blocks searched = (1+N)/2
- **Buckets improve efficiency** by reducing the number of index blocks needed.

Two Similar Questions

Question 1: Indexing with Different Block Capacities

Suppose that blocks can hold: **4 records**,**15 key-pointer pairs**, **60 pointers** in a bucket.

A **database has 4000 records**, and each search key appears in **20 records**.
Calculate the storage and average block inspections for:

1. **Indirect bucket scheme**
2. **Direct indexing**

Question 2: Search Performance for Primary Indexing

Suppose we have **5000 records** indexed by a **primary index** where:

- Each **index block** stores **20 key-pointer pairs**.
- Each **data block** holds **5 records**.

We want to retrieve **one record** by searching for its key.

1. **How many index blocks need to be searched (B+ tree structure assumed)?**
2. **How many data blocks are accessed?**
3. **Compare the performance with a secondary index.**

Final Thoughts

- **Indirect bucket indexing** reduces space and improves search efficiency.
- **Direct indexing** is simpler but less efficient in large datasets.
- **Block-based storage structure** impacts both **space and performance**.

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:**

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

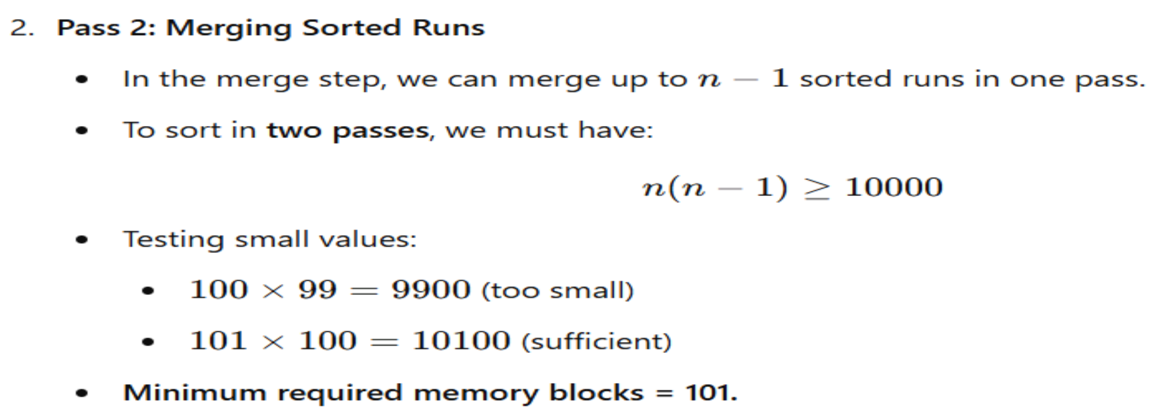
4. **Total Blocks Required to Store Relation R:**

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

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 100001000010000 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=32A = 32A=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 RRR 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 nnn) reduce the number of sorted runs** and lower I/O costs.

Exercise 13.2.5 Sol

Exercise 13.2.5 : On the assumptions of Exercise Sheet 6 Q2 what is the average number of disk I/Os to find and retrieve the ten records with a given search-key value both with and without the bucket structure? Assume nothing is in memory to begin, but it is possible to locate index or bucket blocks without incurring additional I/O's beyond what is needed to retrieve these blocks into memory.

(Note from Q6: Blocks can hold either 3 records, 10 key-pointer pairs, or 50 pointers. We use the indirect bucket scheme. We have 3000 records and each search-key value appears in 10 records.)

1. With Buckets : 1 (for index) + 1(for bucket) + 10 (for records)
2. Without Buckets : 1 (for index) + 10 (records)

   Since index is sorted and one block can hold 10 key-pointers and exactly ten records share the same key we will need only 1 I/O.

Solution to Exercise 13.2.5

Given Information

- **3000 records** total.
- **Each search-key value appears in 10 records**.
- **Block capacities**:
  - **3 records per block**
  - **10 key-pointer pairs per block** (for the index)
  - **50 pointers per block** (for the bucket structure)
- **We use an indirect bucket scheme**.

Solution

We need to find the **average number of disk I/Os** to retrieve **10 records** with the same search-key value, both **with and without buckets**.

Case 1: With Buckets

1. **Retrieve the index block → 1 I/O**
2. **Retrieve the bucket block → 1 I/O**
3. **Retrieve the 10 records → 10 I/Os** (since each record is accessed separately)

Thus, the **total number of disk I/Os = 1 + 1 + 10 = 12**

Case 2: Without Buckets

1. **Retrieve the index block → 1 I/O**
2. **Retrieve the 10 records → 10 I/Os**

Thus, the **total number of disk I/Os = 1 + 10 = 11**

Why Do We Need Fewer I/Os Without Buckets?

- Since the index is sorted and **one index block can hold 10 key-pointer pairs**, we can find **all 10 records directly from the index** in one lookup.
- This eliminates the need for an intermediate **bucket block**, saving **one I/O**.

Two More Similar Questions for Practice

1) Average I/O Cost for Searching Records with Different Indexing

Consider a database with **6000 records**, where each search-key appears in **12 records**.

- Blocks can hold:
  - **4 records per block**
  - **12 key-pointer pairs per index block**
  - **60 pointers per bucket block**
- We use an **indirect bucket scheme**.

Questions:

a) Compute the **average number of I/Os** to find **all 12 records with a given key with buckets**.
b) Compute the **average number of I/Os without buckets**.
c) If the database increases to **12000 records**, how does the I/O cost change?

---

2) Comparison of Clustered vs. Unclustered Index I/O Cost

A database has **5000 records**, and **each search key appears in 8 records**.

- Blocks can store:
  - **5 records per block**
  - **15 key-pointer pairs per index block**
  - **40 pointers per bucket block**

Questions:

a) Compute the **I/O cost using a clustered index** (where records with the same key are stored together).
b) Compute the **I/O cost using an unclustered index**.
c) How does the performance change if the block size is doubled?