1. What is an Index?

- An **index** is a data structure that helps **locate data quickly** based on a search key.
- Works like a book index — lets you jump directly to the relevant location instead of scanning everything.
- Used in DBMS to find records with a given attribute value.
- Must be **created** before use (usually by the DB designer).
- Comes with a **maintenance cost** (updates when data changes).
- Types:
    - **Primary index** – based on a sorted primary key.
    - **Secondary index** – based on a non-primary attribute.
    - **Dense vs. Sparse** indexes.

---

2. Sequential Files

- **Definition:** Records are stored **sorted by the search key**.
- Benefits: Speeds up queries on that key; supports range scans.
- Records are stored in **blocks**, which are:
    - Physically contiguous (next to each other on disk)
    - Often **chained** for navigation.
- **Insertion:**
    1. Place new record in correct block (if space is available).
    2. If full:
        - Insert a new block in the correct position (if possible).
        - Or create an **overflow block** linked to the original.

•1. Create new block; insert into proper order if possible (what if blocks are consecutive around a track for efficiency?).

This line from your notes is talking about an **insertion strategy** for **sequential files** when a block is full**.

Context

In a **sequential file** (records sorted by search key), blocks on disk are often stored **contiguously** to minimize disk head movement (important for HDDs).

When you insert a new record into the sequence:

1. **Find the correct block** based on the key.
2. If that block has **space**, insert in order.
3. If **block is full**:
    - **Option A:** Create a **new block** and place it in the correct sequence **physically** (between its neighbors) if possible.
        - If the blocks are stored consecutively on a track, this may break the contiguity — could hurt sequential read performance.
        - If space exists on the track (e.g., unallocated sectors), inserting the block there keeps efficiency high.
    - **Option B:** If physical insertion is impossible without reorganizing the file, create an **overflow block** and link it to the full block (keeps sequential order logically, but not physically).
    - **Option C:** Reorganize the file to restore full contiguity (expensive).

Why the note says "what if blocks are consecutive around a track?"

- Because in that case, inserting a new block **between** them might require shifting many blocks or breaking the efficient sequential layout.
- DBMS designers must choose between:
  - Keeping physical layout perfect (requires big data movement).
  - Allowing logical order only (and using overflow blocks).

---

3. Dense Index

- **Pointer to every record** in the file, sorted by the search key.
- Pros:
  - Smaller than the full dataset (records may be large, key-pointer pairs small).
  - Index may fit in memory → faster than searching the file.
  - Can check record existence without reading the data file.
- Used when **fast exact match** is important.

---

4. Sparse Index

- **Pointer to only some records**, typically **first record of each block**.
- Pros:
  - Saves index space → can store more of it in memory.
- Cons:
  - To check for existence, must access the data file.

---

5. Dense vs. Sparse

Dense	Sparse
Pointer for every record	Pointer for only some records
Can test existence without accessing file	Need to read the data file
Larger index size	Smaller index size

---

6. Multiple Levels of Index

- A **second-level index** is an index built **on top of another index**.
- Often **sparse** at higher levels → likely to fit in main memory.
- Goal: Reduce disk I/O.
- **Dense higher-level indexes** are not useful (wasteful).

Example of 2-Level Indexing

- Level 1: Index points to blocks in the data file.
- Level 2: Index points to blocks in the Level 1 index.
- Level 2 may fit entirely in memory → very fast lookups.

Summary Table

Concept	Key Idea
Index	Fast lookup structure for records using a search key
Sequential File	Data stored in sorted order of search key
Dense Index	Entry for every record
Sparse Index	Entry for first record in each block
Multi-level Index	Index built on top of another index to speed up lookups

---

Alright — here are 💉 **practice questions + solutions** for the **Index Structures** lecture.

Practice Questions

Q1 — Conceptual

Explain the difference between a **dense** index and a **sparse** index. Which one allows checking record existence without reading the data file?

Q2 — Applied

We have a data file with 200,000 records stored in blocks of 100 records each.

1. How many index entries will a **dense** index have?
2. How many index entries will a **sparse** index have if it stores one entry per block?

Q3 — Multi-level Index

A first-level sparse index has 4,000 entries, and each block can hold 200 entries.

1. How many blocks does the first-level index occupy?
2. How many entries will the second-level sparse index have?

Q4 — Sequential File Insertion

We store a sequential file sorted by customer ID. Block 12 is full, and a new record with a key between two existing keys in Block 12 arrives. List the possible insertion strategies.

Q5 — Memory Fit

Why is it beneficial for higher-level indexes to be sparse rather than dense?

✅ Solutions

A1.

- **Dense Index:** Has one entry for *every* record. Can check existence without accessing the data file.
- **Sparse Index:** Has one entry for *some* records (usually first in each block). Must read data file to confirm existence.
  Answer: Dense index allows checking existence without reading the data file.

A2.

1. Dense index → **200,000** entries (one per record).
2. Sparse index → **2,000** entries (one per block: 200,000 / 100).

A3.

1. First-level index blocks needed = 4,000 ÷ 200 = **20 blocks**.
2. Second-level sparse index → One entry per block in first-level index = **20 entries**.

A4.
Possible strategies:

1. Shift records within Block 12 if space exists (rare in full block).
2. Create an **overflow block** linked from Block 12.
3. Split Block 12 into two blocks (reallocate records).

A5.
Because sparse higher-level indexes take less space → more likely to fit entirely in memory → fewer disk I/Os → faster lookups.

Index Structures – Cheat Sheet

---

1. What is an Index?

- **Definition:** A data structure that speeds up record retrieval using a *search key*.
- **Analogy:** Book index → lets you jump directly to the relevant page.
- **Trade-off:** Faster lookups vs. extra storage and update cost.

---

2. Sequential Files

- Records stored **sorted** by the search key.
- Stored in **blocks** on disk.
- Supports **range queries**.
- **Insertion strategies:**
    1. Insert in place (if space).
    2. Overflow block (linked list).
    3. Split block.

---

3. Dense Index

- **One index entry per record**.
- Sorted by search key.

- **Pros:**
  - Can check if a record exists without reading the data file.
  - Smaller than full dataset → may fit in memory.
- **Cons:** Larger than sparse index.

---

4. Sparse Index

- **One index entry per block** (usually first record in each block).
- **Pros:** Smaller → may fit more in memory.
- **Cons:** Need to read data file to confirm record existence.

Dense vs. Sparse Table:

Feature	Dense	Sparse
Entries	1 per record	1 per block
File Access Needed?	No (for existence check)	Yes
Size	Larger	Smaller

---

5. Multi-level Index

- **Index on top of another index**.
- Higher-level indexes are **sparse** → smaller, may fit in RAM.
- Speeds up search:
  - Level 1: Index → data blocks.
  - Level 2: Index → Level 1 blocks.
- Example:
  - Level 1: 4,000 entries, 200/block → 20 blocks.
  - Level 2: 20 entries → fits in 1 block in memory.

---

6. When to Use

- **Dense:** Fast exact match; small dataset.
- **Sparse:** Large datasets; memory constraints; range queries.
- **Multi-level:** Very large datasets where even the index doesn't fit in memory.

Formula Reminders:

- #Sparse entries = #Blocks in data file.
- #Blocks for index = ⌈ #Entries ÷ EntriesPerBlock ⌉.
- Multi-level index stops when the top level fits in memory.
