Dense Indexing

Definition

Dense indexing involves creating an index entry for every single search key (or record) in the database file. In a dense index, each key in the index file points to a unique record in the data file.

Example

Consider a database table of student records with the following attributes: `**StudentID**`, `**Name**`, and `**Grade**`.

Data File:

StudentID	Name	Grade
1	Alice	A
2	Bob	B
3	Charlie	A
4	David	C

Dense Index File:

StudentID	Pointer
1	→ Record 1
2	→ Record 2
3	→ Record 3
4	→ Record 4

In this example, the dense index file contains an entry for every `**StudentID**` in the data file, with each entry pointing to the corresponding record.

Sparse Indexing

Definition

Sparse indexing involves creating an index entry for only some of the search keys. In a sparse index, index entries are created for only a subset of records, typically the first record in each block of the data file. Sparse indexes are usually used with larger data files to reduce the size of the index.

Example

Using the same student records example but assuming the data file is organized into blocks of two records each:

Data File:

Block	StudentID	Name	Grade
1	1	Alice	A
	2	Bob	B
2	3	Charlie	A
	4	David	C

Sparse Index File:

Block	StudentID	Pointer
1	1	→ Block 1
2	3	→ Block 2

In this sparse index example, each index entry points to the first record in each block of the data file.

Both Possible

Block	StudentID	Name	Grade
1	1	Alice	A
	2	Bob	B
	3	Charlie	A
2	4	David	C
	5	Eve	B
	6	Frank	A

Dense Index Example:

StudentID	Pointer
1	→ Record 1 in Block 1
2	→ Record 2 in Block 1
3	→ Record 3 in Block 1
4	→ Record 1 in Block 2
5	→ Record 2 in Block 2
6	→ Record 3 in Block 2

Sparse Index Example:

Block	StudentID	Pointer
1	1	→ Block 1
2	4	→ Block 2

Ordered Database and Unordered Database

Primary Indexing(Ordered DB, Primary Key)

Definition

Primary indexing involves creating an index for the primary key of a data file. The primary key is a unique identifier for each record in a database. In primary indexing, the index file is typically sorted based on the primary key values, and each entry in the index points to a specific record or block in the data file.

Primary indexing can be dense or sparse, but it is usually sparse because the primary key is already unique and sorted.

Example

Consider a database table of student records with the following attributes: **StudentID**, **Name**, and **Grade**. The **StudentID** is the primary key.

Data File:

Block	StudentID	Name	Grade
1	1	Alice	A
	2	Bob	B
2	3	Charlie	A
	4	David	C
3	5	Eve	B
	6	Frank	A

Dense Primary Index

A dense primary index would include an index entry for every record in the data file.

Dense Primary Index File:

StudentID	Pointer
1	→ Record 1 in Block 1
2	→ Record 2 in Block 1
3	→ Record 1 in Block 2
4	→ Record 2 in Block 2
5	→ Record 1 in Block 3
6	→ Record 2 in Block 3

Sparse Primary Index

A sparse primary index would include an index entry for only the first record of each block in the data file. This reduces the size of the index file and can still efficiently locate records by using the index to narrow down the search to a specific block.

Sparse Primary Index File:

StudentID	Pointer
1	→ Block 1
3	→ Block 2
5	→ Block 3

Example Scenario with Detailed Steps

Suppose we want to find the record for **StudentID = 4**.

1. **Using Sparse Primary Index:**

    - Look up **StudentID = 4** in the sparse primary index.
    - The index entry **StudentID = 3** points to Block 2.
    - Search within Block 2:
        - Block 2 contains records with **StudentID = 3** and **StudentID = 4**.
        - Locate the specific record for **StudentID = 4**.

2. **Using Dense Primary Index:**

    - Look up **StudentID = 4** directly in the dense primary index.
    - The index entry **StudentID = 4** directly points to Record 2 in Block 2.

When to Use Primary Indexing

- **When records are frequently retrieved based on the primary key:** Primary indexing is highly efficient for direct lookups using the primary key.
- **When the dataset is relatively static:** Since the primary key is unique and sorted, updates and deletions can be managed efficiently.
- **When storage space for the index is a concern:** A sparse primary index can be used to minimize the storage overhead while still providing efficient access to records.

Benefits and Limitations

Benefits:

- **Fast search and retrieval:** Primary indexing allows quick access to records based on the primary key.
- **Efficient sorting:** Since the index is sorted based on the primary key, it supports efficient range queries.

Limitations:

- **Update overhead:** Inserting, updating, or deleting records can be more complex and require maintaining the index order.
- **Storage requirements:** Although a sparse index reduces storage needs, maintaining any index still incurs some storage overhead.

Summary

Primary indexing, whether dense or sparse, provides a structured way to quickly access records using the primary key. The choice between dense and sparse indexing depends on the specific use case, balancing the need for speed and storage efficiency.

Clustering Indexing(Ordered DB, Non-Key)

Clustering indexing is used when records in a database table are physically ordered based on a non-primary key (clustering key) which may not be unique. A clustering index is created on this non-primary key to improve query performance, especially for range queries. In clustering indexing, the data file itself is ordered based on the clustering key, and the index points to blocks of records rather than individual records.

Consider a database table of student records with the following attributes: **StudentID**, **Name**, **Grade**, and **Department**. Suppose we frequently query the table based on the **Department** attribute, which is not unique but groups records logically.

Data File Ordered by Department:

Block	StudentID	Name	Grade	Department
1	4	David	C	Computer Science
	1	Alice	A	Computer Science
2	5	Eve	B	Mathematics
	2	Bob	B	Mathematics
3	6	Frank	A	Physics
	3	Charlie	A	Physics

Clustering Index File:

Department	Pointer
Computer Science	→ Block 1
Mathematics	→ Block 2
Physics	→ Block 3

Detailed Example Scenario

Suppose we want to find all students in the "Mathematics" department.

1. **Using Clustering Index:**
    - Look up "Mathematics" in the clustering index.
    - The index entry for "Mathematics" points to Block 2.
    - Retrieve all records in Block 2:
        - Block 2 contains records for **StudentID = 5** (Eve) and **StudentID = 2** (Bob), both in the Mathematics department.

When to Use Clustering Indexing

- **When queries involve range searches or groupings:** Clustering indexing is particularly useful for queries that request a range of values or group records based on the clustering key.
- **When the order of records matters:** Since the data file is physically ordered based on the clustering key, accessing a range of records is efficient.
- **When the clustering key is not unique:** Clustering indexing handles non-unique keys well, grouping all relevant records together in contiguous storage blocks.

Benefits:

- **Improved range query performance:** Clustering indexing speeds up queries that access a range of records based on the clustering key.
- **Efficient storage of related records:** Grouping records together based on the clustering key reduces the need for extensive searching and improves data locality.

Limitations:

- **Update overhead:** Inserting new records or updating existing ones can be more complex, as maintaining the physical order of records is necessary.
- **Storage requirements:** Clustering indexing might require additional storage to keep track of the blocks of records.

Summary

Clustering indexing provides an efficient way to handle queries involving non-primary keys that group records logically. It optimizes range queries and enhances the performance of searches on the clustering key by organizing the data file based on that key. This method balances the need for fast retrieval with the complexity of maintaining order during updates and insertions.

Secondary Indexing(Unordered DB, Non-Key)

Definition

Secondary indexing involves creating an index on a non-primary key attribute, which is not used to determine the physical order of records in the data file. Unlike primary and clustering indexes, secondary indexes can be created on attributes that are not unique and can have duplicate values. Secondary indexes improve query performance for searches involving non-primary attributes.

Types of Secondary Indexes

1. **Dense Secondary Index:** Contains an index entry for every record in the data file.
2. **Sparse Secondary Index:** Contains an index entry for only some records, usually the first record of each block.

Example

Consider a database table of student records with the following attributes: **StudentID**, **Name**, **Grade**, and **Department**. Suppose we want to create a secondary index on the **Grade** attribute, which can have duplicate values.

Data File:

StudentID	Name	Grade	Department
1	Alice	A	Computer Science
2	Bob	B	Mathematics
3	Charlie	A	Physics
4	David	C	Computer Science
5	Eve	B	Mathematics
6	Frank	A	Physics

Dense Secondary Index

A dense secondary index on **Grade** would have an index entry for every record, with each entry pointing to the corresponding record in the data file.

Dense Secondary Index File on Grade:

Grade	StudentID	Pointer
A	1	→ Record 1
A	3	→ Record 3
A	6	→ Record 6
B	2	→ Record 2
B	5	→ Record 5
C	4	→ Record 4

Sparse Secondary Index

A sparse secondary index on **Grade** would have fewer entries, typically pointing to the first occurrence of each grade in a block. This example is less common for secondary indexes because the attribute values can be duplicated and spread across the file.

Sparse Secondary Index File on Grade:

Grade	Block	Pointer
A	1	→ Block 1
B	2	→ Block 2
C	3	→ Block 3

When to Use Secondary Indexing

- **When queries frequently involve non-primary key attributes:** Secondary indexing is beneficial for improving performance on searches involving attributes other than the primary key.
- **When attributes have duplicate values:** Secondary indexes are suitable for attributes with duplicate values, such as **Grade** in the student records example.
- **When the data file is large and direct search is inefficient:** Secondary indexes can significantly reduce the search time by providing quick access paths to the records.

Benefits and Limitations

Benefits:

- **Improved query performance:** Secondary indexing speeds up searches involving non-primary key attributes.
- **Flexibility:** Allows indexing on multiple attributes, enabling efficient querying on various fields.

Limitations:

- **Update overhead:** Maintaining secondary indexes can be complex and resource-intensive, especially when records are inserted, updated, or deleted.
- **Storage requirements:** Secondary indexes require additional storage space, which can be substantial if multiple secondary indexes are created.

Example Scenario with Detailed Steps

Suppose we want to find all students with a **Grade = A**.

1. **Using Dense Secondary Index:**
    - Look up **Grade = A** in the dense secondary index.
    - Retrieve the pointers to the records with **StudentID = 1**, **StudentID = 3**, and **StudentID = 6**.
    - Access the records in the data file:
        - Record for **StudentID = 1**: Alice, Grade A, Computer Science
        - Record for **StudentID = 3**: Charlie, Grade A, Physics
        - Record for **StudentID = 6**: Frank, Grade A, Physics

Summary

Secondary indexing enhances the performance of queries involving non-primary key attributes by providing additional access paths to the data. While it improves search efficiency, it also introduces complexities in maintaining the indexes and requires extra storage space. Secondary indexes are especially useful when queries often involve attributes with duplicate values or when multiple attributes need to be indexed for flexible querying.

B-Tree Indexing

Definition

B-Tree indexing is a widely used indexing method that maintains sorted data in a balanced tree structure. Each node in a B-Tree contains multiple keys and child pointers, ensuring that the tree remains balanced and operations such as search, insert, and delete are efficient. The height of the B-Tree grows logarithmically with the number of records, providing efficient access to data.

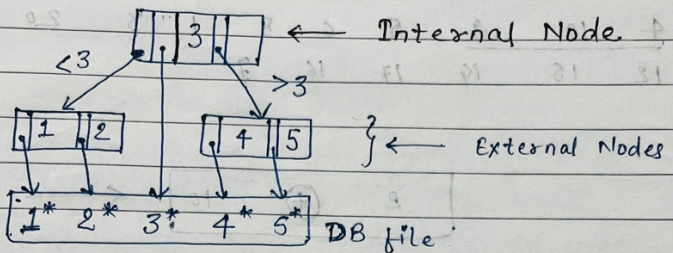
Structure of a B-Tree

A B-Tree of order m (also known as a m-ary B-Tree) has the following properties:

- Each node can have at most $m - 1$ keys and m child pointers.
- Each internal node (except the root) has at least $\lceil \frac{m}{2} \rceil$ children.
- The root has at least two children if it is not a leaf.
- All leaf nodes are at the same level, ensuring the tree remains balanced.
- Keys within a node are sorted, and the keys in the subtrees are ordered.

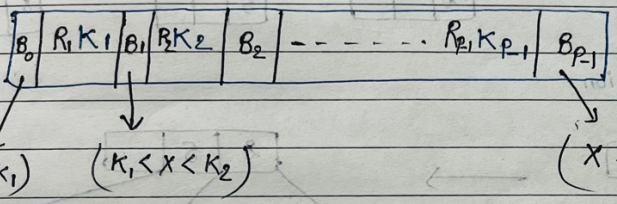
(DYNAMIC INDEXING) →

B-TREE Indexing → (Order P : Max #f BP in Block)

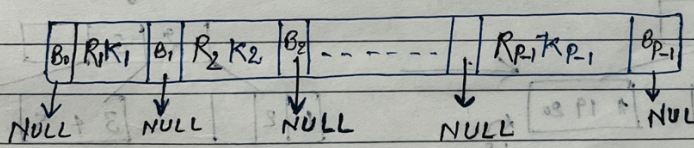


Definition

(i) Structure of Internal Node :→ { P B_p, (P-1) Keys, (P-1) R_p }



(ii) Structure of External Nodes



wastage of space here for maintaining the Block pointer.

(iii) Every Internal Node Except Root Should be at least $\lceil P/2 \rceil$ Block pointers at most P Block pointers

(iv) Root Can be atleast 2 Bp at most P Block pointers

(v) Every leaf (node) should be at same level.

B+ Tree Indexing

Definition

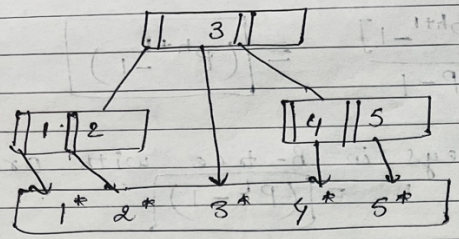
A B+ Tree is an extension of the B-Tree data structure that maintains sorted data in a balanced tree format. Unlike B-Trees, in B+ Trees, all values are stored at the leaf level and internal nodes only store keys for routing. This design enhances the efficiency of range queries and sequential access.

Structure of a B+ Tree

A B+ Tree of order m has the following properties:

- Each internal node (except the root) has between $\lceil \frac{m}{2} \rceil$ and m children.
- The root has at least two children if it is not a leaf.
- Each leaf node contains between $\lceil \frac{m}{2} \rceil - 1$ and $m - 1$ keys.
- All keys are stored in leaf nodes.
- Leaf nodes are linked to form a linked list, allowing for efficient range queries and sequential access.
- Internal nodes store keys and pointers to child nodes, facilitating search and traversal.

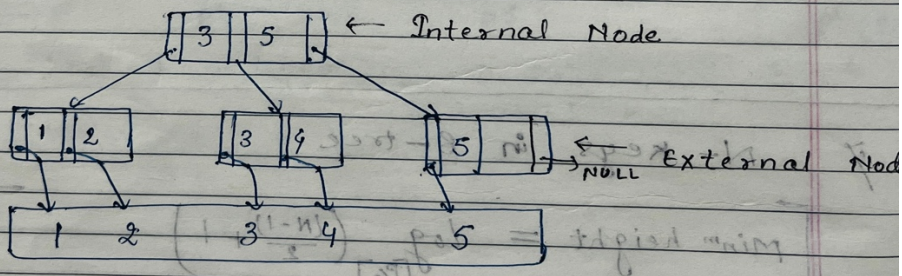
B- Tree



Adv
- * Random Access. $O(\log_p N)$ Blocks
- * Range Queries

Dis
- * Not Best Suitable for Sequential Access.

B^+- TREE



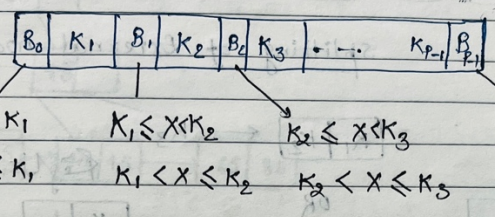
→ Every Key should be Leaflevel

* Random Access.
* Range Queries.
* Sequential Access.

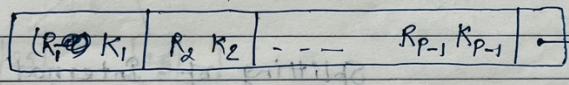
Definition :→ (ORDER p : Max # Blocks Pointer in Node)

∴ NO Record Pointer in B^+- tree.
for Internal nodes

[1] Internal Node :→ (p-1) Keys



[2] External Node :— (Max (p-1) Key)



{(p-1) Key + (p-1) R_p + 1 B_p}