Ch-14 Indexing

Think of the index at the back of a book - sorted small

- **Ordered indices :** Based on sorted ordering of the values
- **Hash indices :** Based on a uniform distribution of values across a range of buckets(uses a hash function)

There is no such thing called best, situation dependent

FACTORS

- **Acess types :** Finding records with a specified attribute value or range that are supported efficiently
- **Acess time :** Time to find data set of items using techinque given
- **Insertion time :** T(Insert data at correct place) + T(Update index structure)
- **Deletion time :** T(Find item) + T(Update index structure)
- **Space overhead :** The additional space occupied by an index structure

---

- search key : set of attributes used to look up records in a file

Ordered Indices

- ordered index : stores search keys in sorted order
- clustering index/primary indices : search key defines sequential order of file(= **index sequential files**)
- nonclustering indices / secondary indices : search key specifies an order different from the sequential order of the file

Dense vs Sparse

- Dense Index : entry appears for everys earch key value(pointer to first data record and rest are sequentially stored)

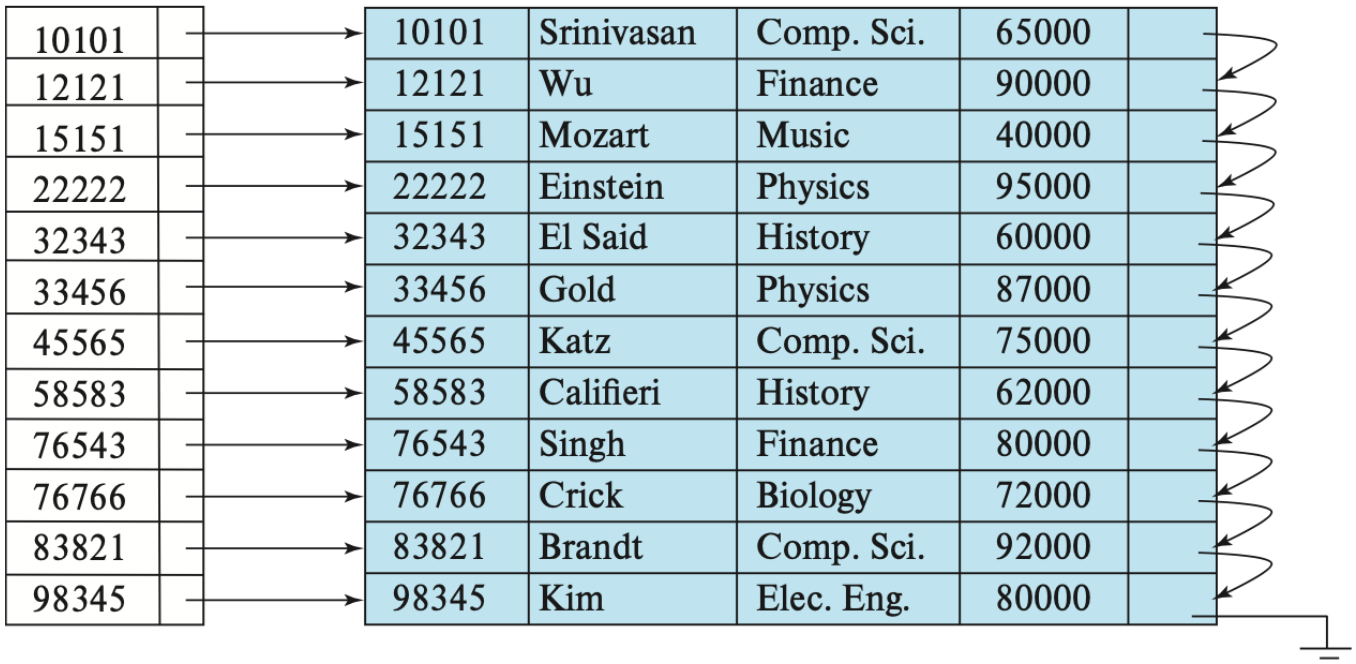


Figure 14.2 Dense index.

- Sparse Index : index entry appears only for some of the search key values. Find index entry with largest serach key value less than or equal to serach key value we are looking. Follow the pointers in the file until we find the desired record.

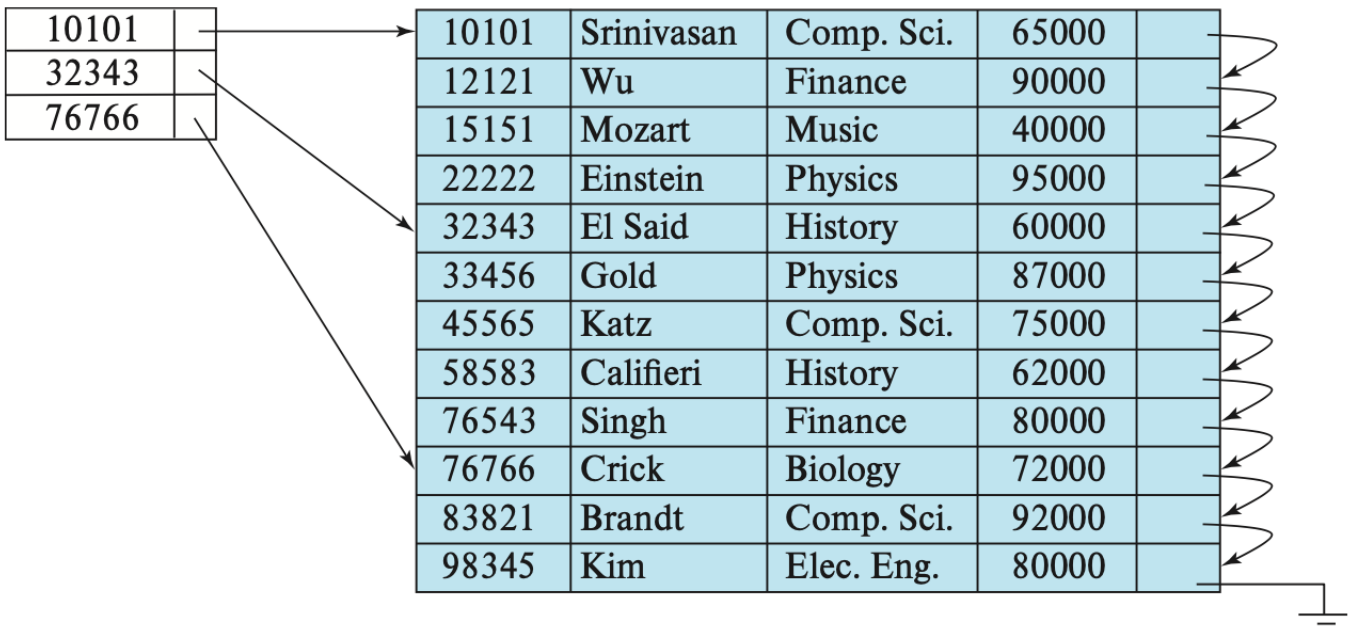


Figure 14.3 Sparse index.

There is a tradeoff between acess time and space overhead. Ideal for a block to have one sparse index. Time will take for block to be read from disk and brought to main memory and very fast to lookup all values once in main memory.

Multilevel Indices

If we use dense index for large records. Then more time even though you want to do a binary serach. So, create a sparse outer index of the inner indices. Recursively do this.

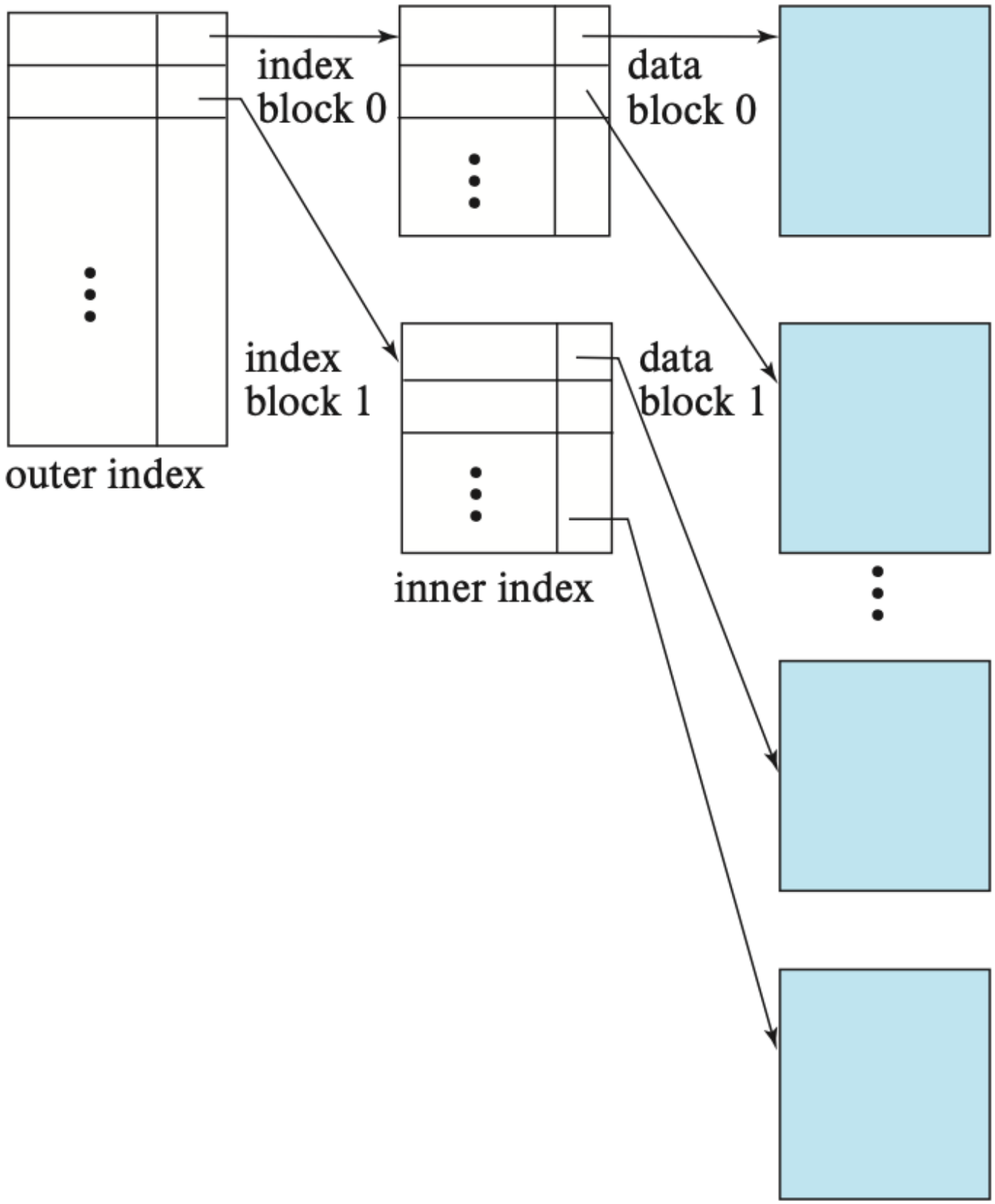


Figure 14.5 Two-level sparse index.

Index Update

Update = deletion + inserting modified data

Insertion

- Dense Indices :
  - If search key value not there, insert it
  -
    - If index entry stores pointers with same search key, add new one
    - Index entry points to first record and new one placed after others
- Sparse Indices :
  - If new block is created, insert first search key value into index.
  -
    - If new record has least search key in its block, update index entry
    - No changes to index

Deletion

- Dense indices :
  - If it is only record, delete index entry.
  -
    - If index entry stores pointers with same search key, remove the record
    - Delete this record and make pointer to point next one
- Sparse indices:
  - If index does not have serach key value chill
  -
    - If deleted one was the only one with its search key, the system replaces the corresponding index record with an index record for the next search key and if it already has index entry then delete the current one
    - change the pointer to point to the next record with same search key value

Secondary Indices

did not understand

composite search key : multi attribute

B+ Tree Index Files

Index-sequential file organization performance degrades as the file grows, both for index lookups and for sequential scans through the data

- Path from root to leaf same

- Root has between 2 and n children
- Non Leaf node have children from range
- $$\lceil n/2 \rceil, n$$

Structure of B+ tree

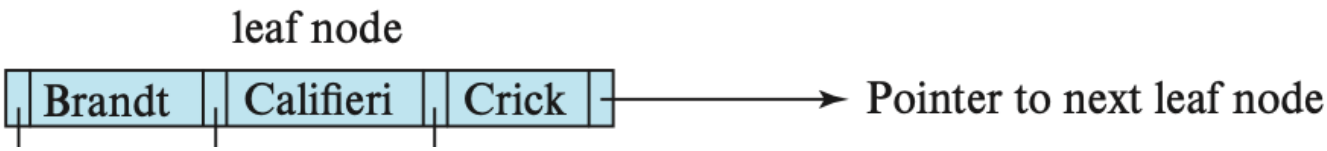
A typical node contians n-1 search key values(sorted) and n pointers

Figure 14.7 Typical node of a B^+-tree.

Leaf node :

Pi points to file record with search key value Ki. Pn points to next leaf node. It should conatain atleast following values

$$\lceil (n-1)/2 \rceil$$



Fanout : Number of pointers in a node

Internal Nodes :

Pointer points to other nodes

They must have atleast following number of pointers

$$\lceil n/2 \rceil$$

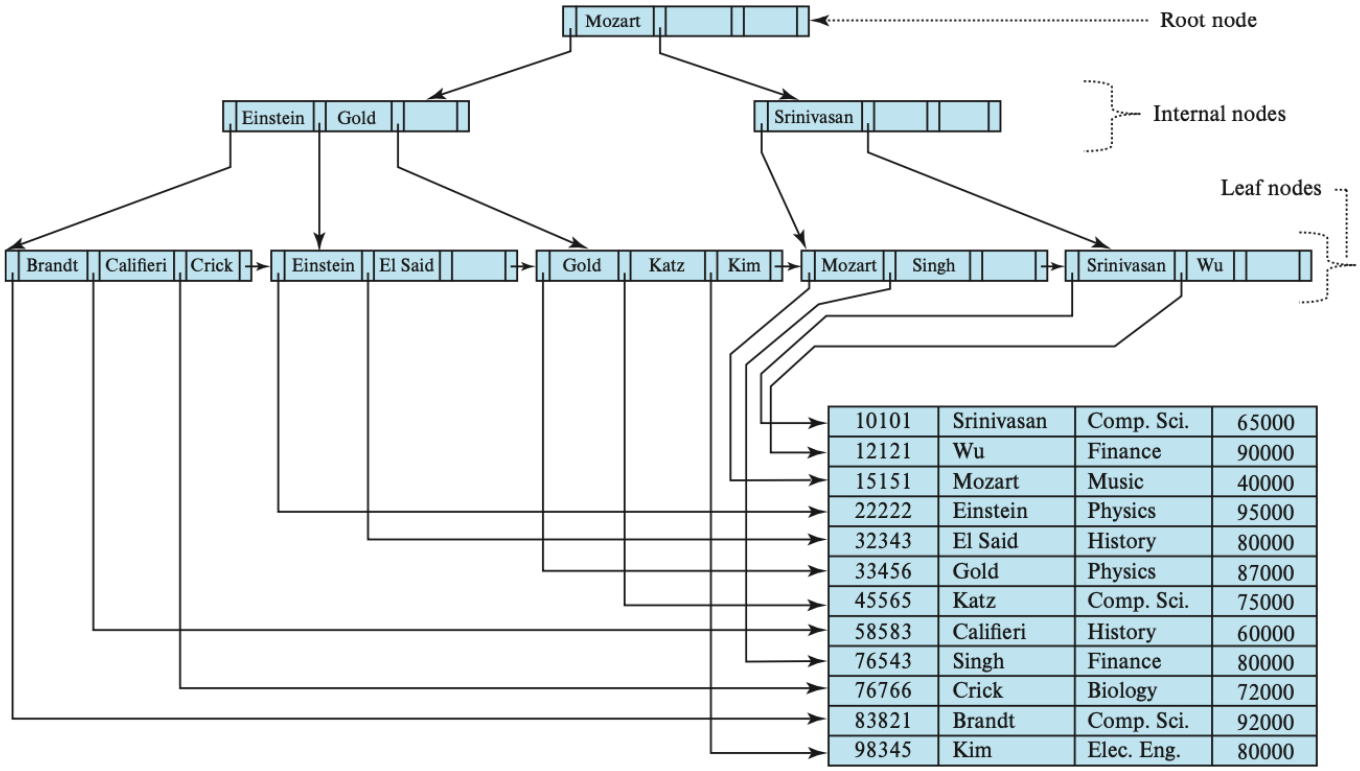


Figure 14.9 B+-tree for *instructor* file ($n = 4$).

- P1 points to subtree that contains search key values less than K1
- For a node with m pointers P2,P3...Pm-1 pointers point to subtree that contains search key values less than Ki and greater than or equal to Ki-1.
- Pm points to part of subtree that contains key values greater than or equal to Km-1

For non unique search keys, append a primary key and use it as search key.

Queries

find(v) : smallest i

- While(!leaf)
  - IF Ki>=v.
    - if Ki = v, go to Pi+1
    - else go to Pi.
  - Else go to Pm
- Check in leaf

findrange(lb,ub):

- While(!leaf)
  - smallest i, lb <= Ki
    - if none go to Pm

        - else if(lb==Ki) go to Pi+1
        - else go to Pi
- i=least value such that lb<=Ki
- if no i then i = 1 + number of keys in C(node)
- While(!flag)
    - n = number of keys in C
    - if i<n & Ki<ub , add Pi to set, i++
    - else if i> n & Pn+1!=null, C = C.Pn+1, i = 1
    - else flag = true

Querying cost = ⌈log⌈n/2⌉(N)⌉
We choose node size same as size of disk block(4 KB)
Search key 12 bytes
disk pointer 8 bytes
n =4000/(12+8) = 200
Search key 32 bytes
n = 4000/(32+8) = 100

Range queries have an additional cost, after traversing down to the leaf level: all the pointers in the given range must be retrieved. These pointers are in consecutive leaf nodes; thus, if M such pointers are retrieved, at most ⌈M/(n/2)⌉ + 1 leaf nodes need to be accessed to retrieve the pointers (since each leaf node has at least n/2 pointers, but even two pointers may be split across two pages). To this cost, we need to add the cost of accessing the actual records. For secondary indices, each such record may be on a different block, which could result in M random I/O operations in the worst case. For clustered indices, these records would be in consecutive blocks, with each block containing multiple records, resulting in a significantly lower cost.

Update

Insertion : Go to where find(v) takes and insert there
Deletion : Go and find the correct one, delete it and move all the entries to left by one position in that leaf node
Inserting Adams -> Brandt,Califeri and Crick node is divided into two nodes Adams,Brandt and Califieri, Crick. Now the we have to point to this new node and hence add one more entry to the

parent node i.e. Califeri

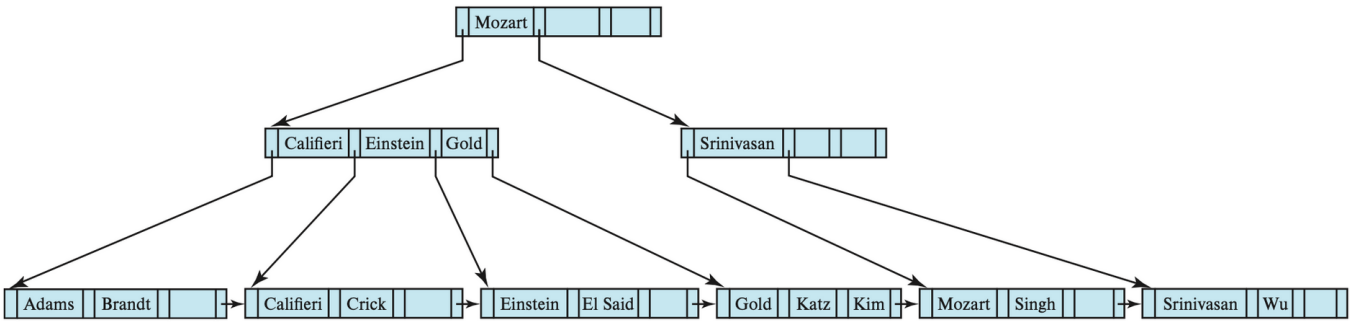


Figure 14.14 Insertion of "Adams" into the B+-tree of Figure 14.9.

Now inserting Lamport, we see that Gold,Katz, Kim is full. So we split into Gold,Katz and Kim, Lamport. Now parent node does not have space for new entry, so it is split and we have Califeri, Einstein and Gold, Lamport. For Kim the left pointer can point to the node with Gold and the right pointer to the Kim. So no need of Gold in the split nodes. As the parent node is split, we have to add to the root node. Hence we should add Gold. Now left pointer of Gold points to the start the right points to Kim node and right of Mozart to Srinivasan.

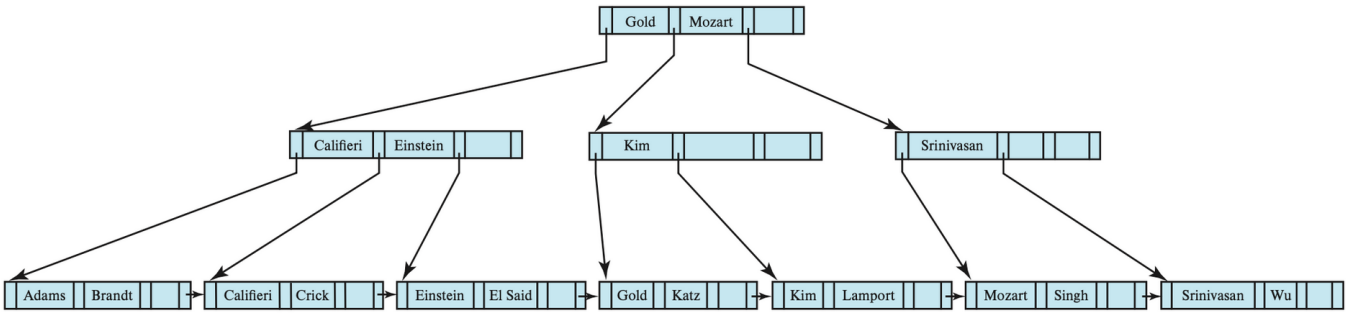


Figure 14.15 Insertion of "Lamport" into the B+-tree of Figure 14.14.

14.16 14.17 14.21 are left for now will be writing them soon!!

Complexity of Update :
Insertion : log⌈n/2⌉(N)
Deletion : log⌈n/2⌉(N)

If nodes are filled in sorted order, only half will be filled

Nonunique Search Key

We add an extra attribute called a uniquifier attribute.
We have two choices, donot add uniquifier and rather store the values with same key values in a bucket which will have issues during the deletion(O(N)) whereas adding uniquifier although increases storage will be same as unique B+ tree.

B+ trees extensions

B+ tree file organization

In secondary indices, in place of pointers to the indexed records, we store the values of the primary-index search-key attributes. For example, suppose we have a primary index on the attribute ID of relation instructor; then a secondary index on dept name would store with each department name a list of instructor's ID values of the corresponding records, instead of storing pointers to the records. Relocation of records because of leaf-node splits then does not require any update on any such secondary index. However, locating a record using the secondary index now requires two steps: First we use the secondary index to find the primary-index search-key values, and then we use the primary index to find the corresponding records.
This approach thus greatly reduces the cost of index update due to file reorganiza- tion, although it increases the cost of accessing data using a secondary index.

Indexing Strings

The fanout of nodes can be increased by using a technique called prefix compres- sion. With prefix compression, we do not store the entire search key value at nonleaf nodes. We only store a prefix of each search key value that is sufficient to distinguish between the key values in the subtrees that it separates. For example, if we had an index on names, the key value at a nonleaf node could be a prefix of a name; it may suffice to store "Silb" at a nonleaf node, instead of the full "Silberschatz" if the closest values in the two subtrees that it separates are, say, "Silas" and "Silver" respectively.

Bulk Loading of B+ tree Indices

Insertion of a large number of entries at a time into an index is referred to as bulk loading of the index. An efficient way to perform bulk loading of an index is as follows: First, create a temporary file containing index entries for the relation, then sort the file on the search key of the index being constructed, and finally scan the sorted file and insert the entries into the index

In bottom-up B+- tree construction, after sorting the entries as we just described, we break up the sorted entries into blocks, keeping as many entries in a block as can fit in the block; the resulting blocks form the leaf level of the B+-tree. The minimum value in each block, along with the pointer to the block, is used to create entries in the next level of the B+- tree, pointing to the leaf blocks. Each further level of the tree is similarly constructed using the minimum values associated with each node one level below, until the root is created.

B-Tree Index Files

B-tree eliminates the redundant storage of search-key values

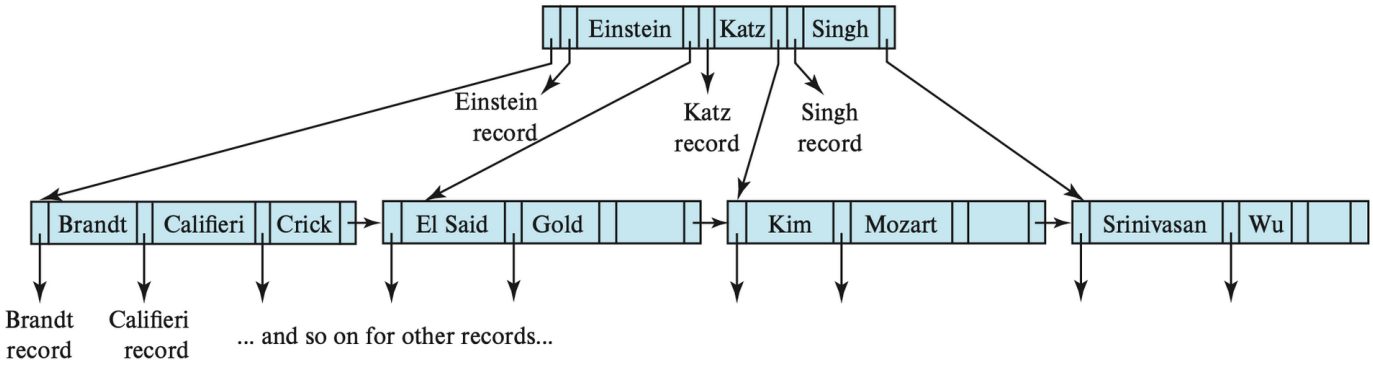


Figure 14.23 B-tree equivalent of B+-tree in Figure 14.9.

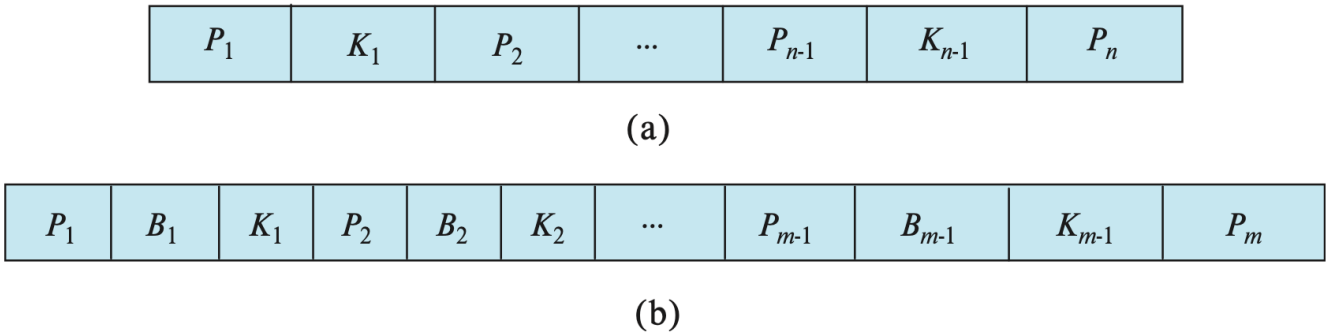


Figure 14.24 Typical nodes of a B-tree. (a) Leaf node. (b) Nonleaf node.

Indexing on Flash Storage

Several extensions and alternatives to B+-trees have been proposed for flash stor- age, with a focus on reducing the number of erase operations that result due to page rewrites. One approach is to add buffers to internal nodes of B+-trees and record up- dates temporarily in buffers at higher levels, pushing the updates down to lower levels lazily. The key idea is that when a page is updated, multiple updates are applied to- gether, reducing the number of page writes per update. Another approach creates mul- tiple trees and merges them; the log-structured merge tree and its variants are based on this idea.

Hash Indices

bucket : unit of storage that can store one or more records
For in-memory hash indices, a bucket could be a linked list of index entries or records
For disk-based indices, a bucket would be a linked list of disk blocks

In a Hash file organization, instead of record pointers, buckets store the actual records

A hash function h is a function from K to B. Let h denote a hash function. With in-memory hash indices, the set of buckets is simply an array of pointers, with the ith bucket at offset i. Each pointer stores the head of a linked list containing the entries in that bucket.

Insertion : We add the index entry for the record to the list at offset i. Hash indexing using overflow chaining is also called closed addressing

Range queries are not supported. If the bucket does not have enough space, a bucket overflow is said to occur. We handle bucket overflow by using overflow buckets.

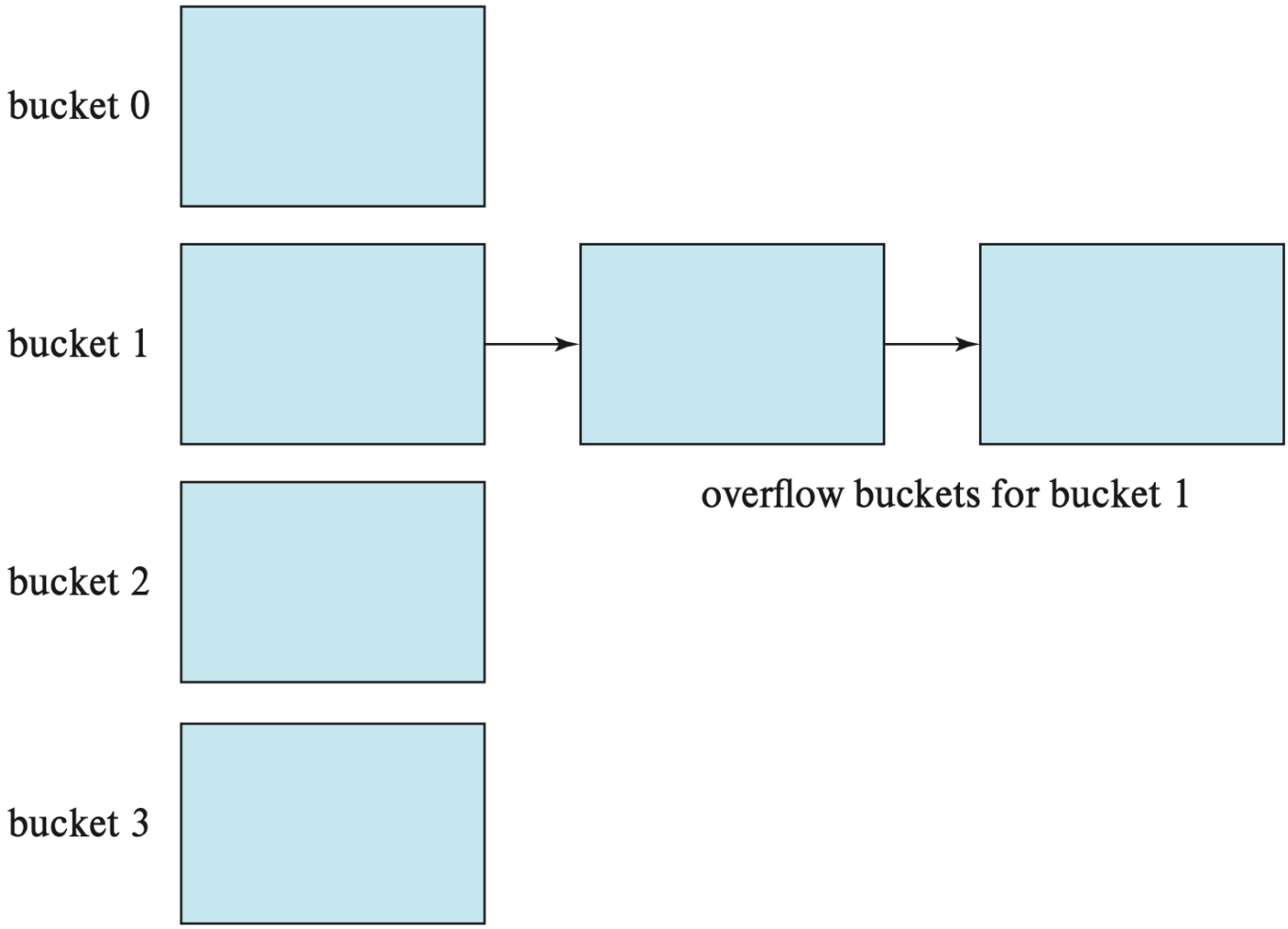


Figure 14.25 Overflow chaining in a disk-based hash structure.

nr = No of records
fr = No of records per bucket
d = fudge factor = 0.2(nearby) => 20% empty
Number of buckets = (nr/fr)*(1+d)

Number of buckets is fixed when the index is created, is called static hashing.

Multiple Key Acess

Covering Indices

will write later

LSM Trees

in memory - L0
on disk - L1,L2,L3 ... Lk

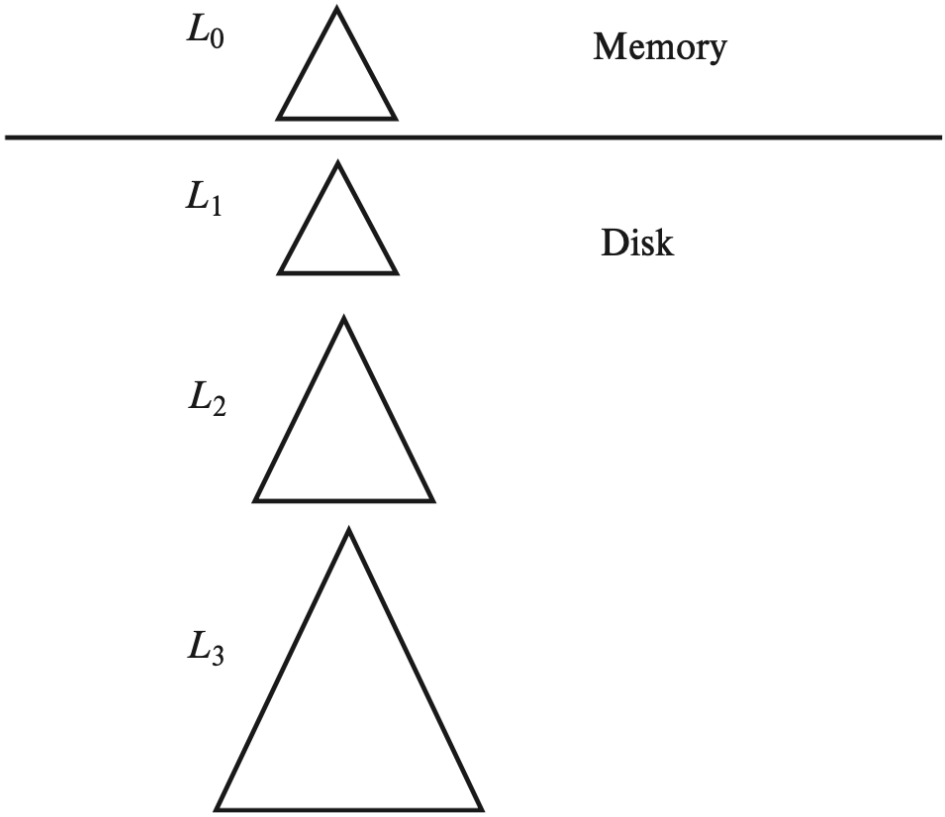


Figure 14.26 Log-structured merge tree with three levels.