UNIT IV

INDEXING AND HASHING

Indexing and Hashing: Basic Concepts, Ordered Indices, B+ tree Index Files, B-tree Index Files, Multiple-Key Access, Static Hashing, Dynamic Hashing, Comparison of Ordered Indexing and Hashing, Bitmap Indices.

Index Definition in SQL Transactions: Transaction Concepts, Transaction State, Implementation of Atomicity and Durability, Concurrent Executions, Serializability, Recoverability, Implementation of Isolation, Testing for Serializability

---

Basic Concepts: Indexing mechanisms used to speed up access to desired data. EX: author catalog in library

Search Key - attribute to set of attributes used to look up records in a file.

An **index file** consists of records (called **index entries**) of the form

Search key	pointer

Index files are typically much smaller than the original file. Two basic kinds of indices are:

1. **Ordered indices:** search keys are stored in sorted order
2. **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

Index Evaluation Metrics:

- Access types supported efficiently. EX:
  - Ex: records with a specified value in the attribute
  - or records with an attribute value falling in a specified range of values (EX: 10000 < *salary* < 40000)
- Access time
- Insertion time
- Deletion time
- Space overhead

Ordered Indices

In an **ordered index,** index entries are stored sorted on the search key value. EX: author catalog in library.

Primary index: In a sequentially ordered file, the index whose search key specifies the sequential order of the file. It is also called **clustering index.** Here search key of a primary index is usually but not necessarily the primary key.

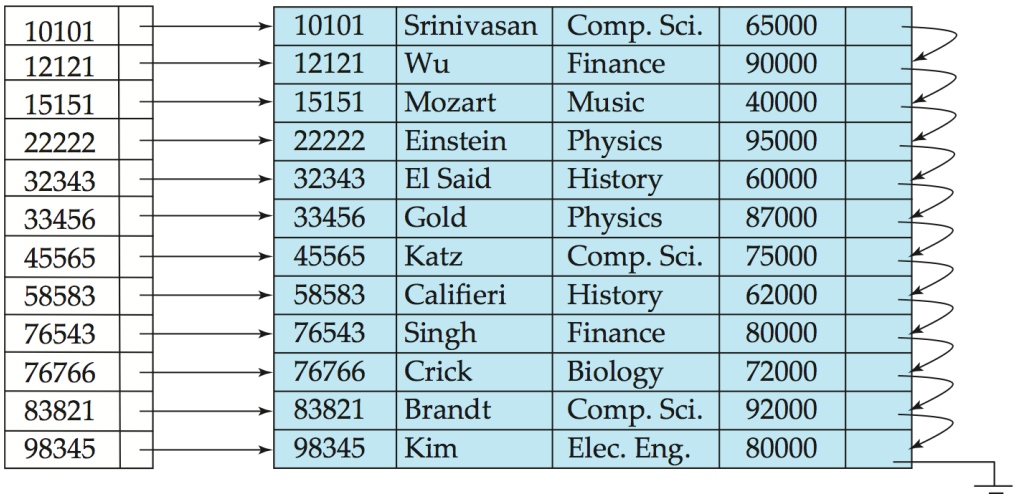
Secondary index: An index whose search key specifies an order different from the sequential order of the file. Also called **non-clustering index.**

Index-sequential file: ordered sequential file with a primary index.

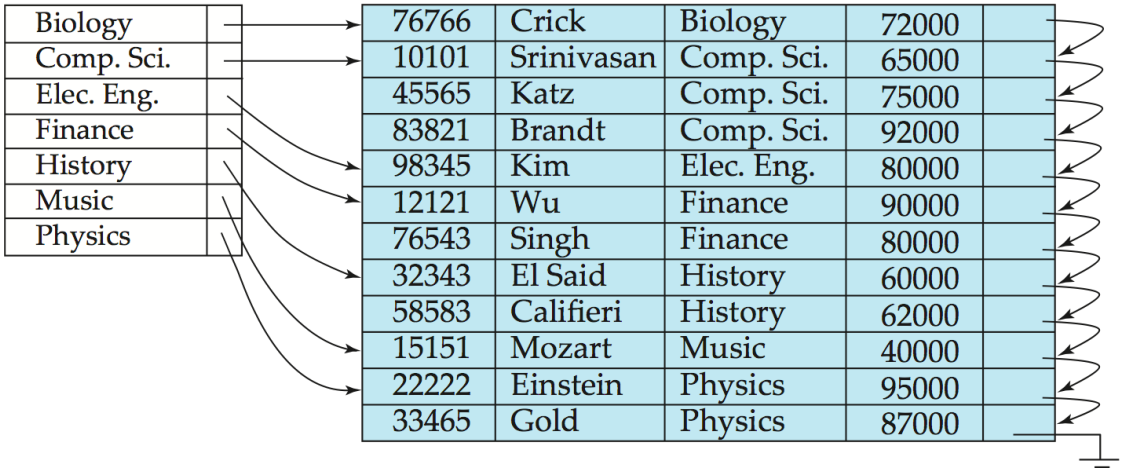
Dense Index Files

Dense index — Index record appears for every search-key value in the file.

EX: Index on *ID* attribute of *instructor* relation



Dense index on *dept_name*, with *instructor* file sorted on *dept_name*

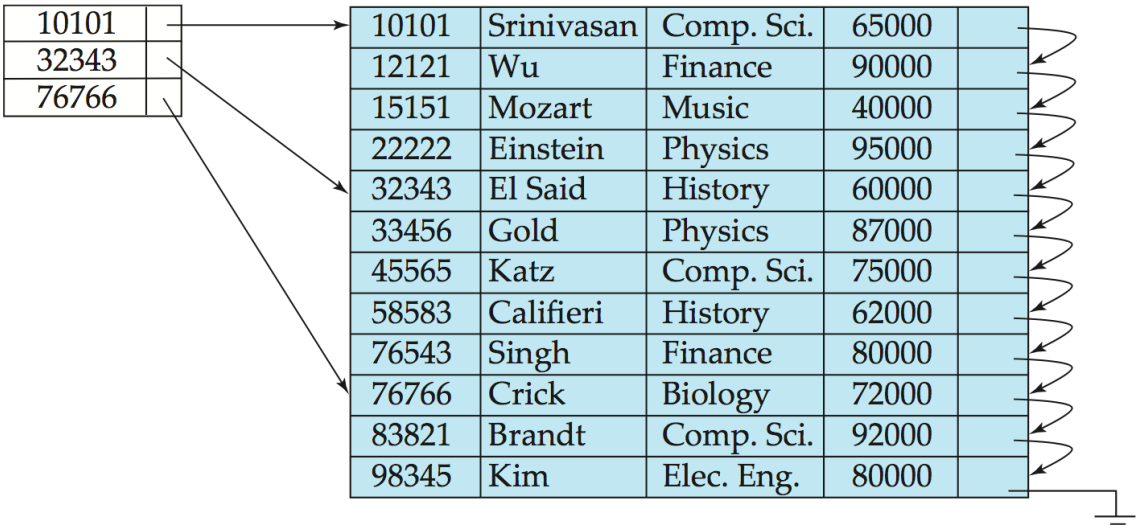


Sparse Index Files

Sparse Index: contains index records for only some search-key values.
It is Applicable when records are sequentially ordered on search-key

To locate a record with search-key value *K* we:

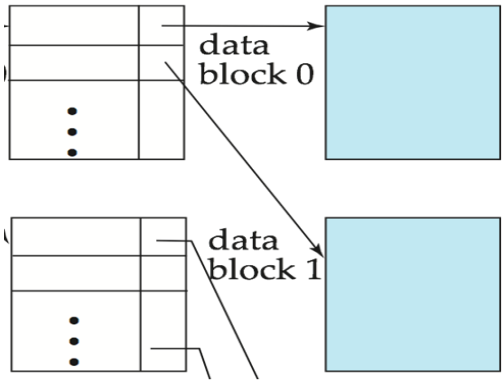
- Find index record with largest search-key value < *K*
- Search file sequentially starting at the record to which the index record points



Compared to dense indices:

- Less space and less maintenance overhead for insertions and deletions.
- Generally slower than dense index for locating records.

Good tradeoff: Sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
  - Example 1: In the *instructor* relation stored sequentially by ID, we may want to find all instructors in a particular department
  - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values
- We can have a secondary index with an index record for each search-key value

Secondary Indices Example

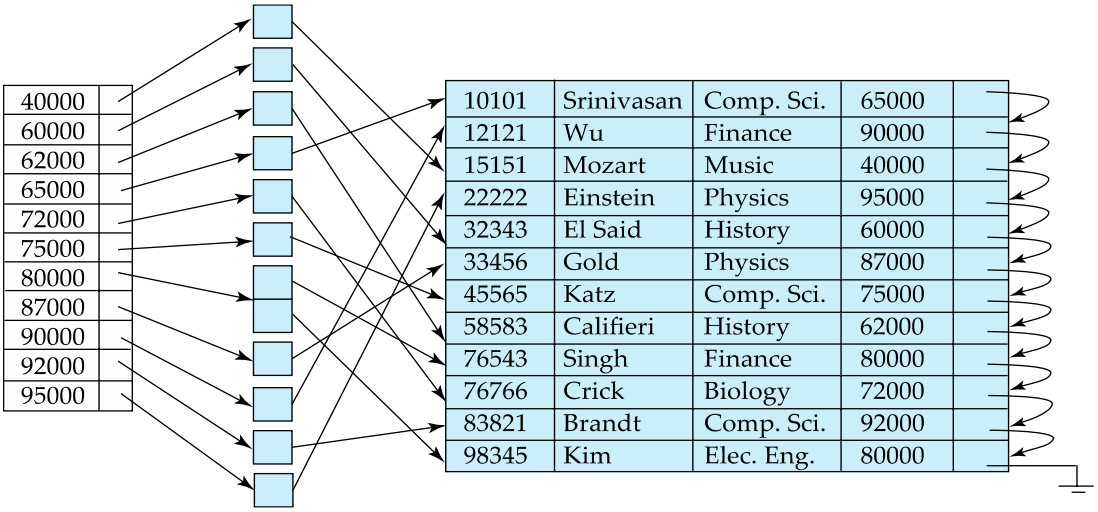


Fig: Secondary index on *salary* field of *instructor*

Index record points to a bucket that contains pointers to all the actual records with that particular search-key value. And Secondary indices have to be dense.

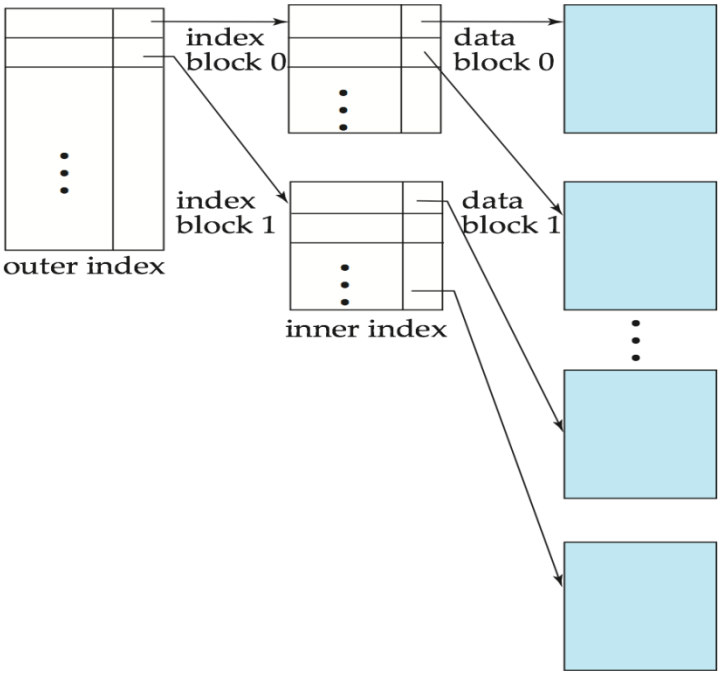
Primary and Secondary Indices offer substantial benefits when searching for records. But updating indices imposes overhead on database modification. When a file is modified, every index on the file must be updated.

- ✓ Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive.
- ✓ Each record access may fetch a new block from disk
- ✓ Block fetch requires about 5 to 10 micro seconds, versus about 100 nanoseconds for memory access

Multilevel Index: If primary index does not fit in memory, access becomes expensive.
Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.

- outer index – a sparse index of primary index
- inner index – the primary index file

If even outer index is too large to fit in main memory, yet another level of index can be created, and so on. Indices at all levels must be updated on insertion or deletion from the file.



Index Update: Record Deletion

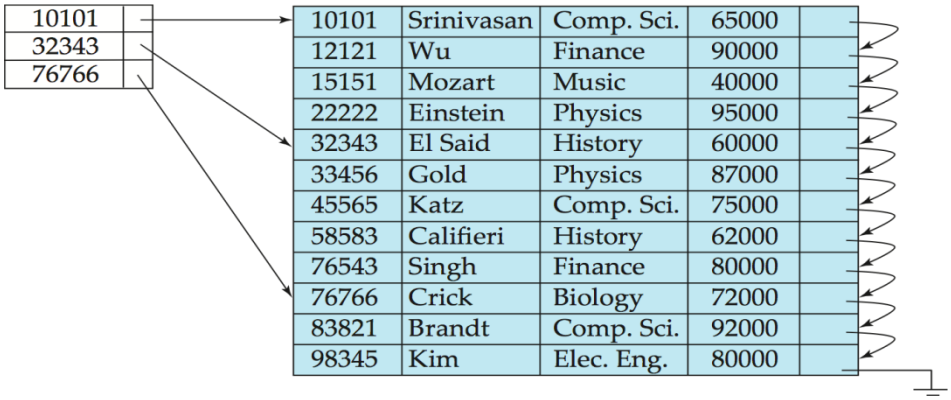
If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

Single-level index deletion:

Dense indices – Deletion of search-key: similar to file record deletion.

Sparse indices –

- If deleted key value exists in the index, the value is replaced by the next search-key value in the file (in search-key order).
- If the next search-key value already has an index entry, the entry is deleted instead of being replaced.



Index Update: Record Insertion

Single-level index insertion: Perform a lookup using the key value from inserted record

Dense indices – if the search-key value does not appear in the index, insert it.

Sparse indices – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.

If a new block is created, the first search-key value appearing in the new block is inserted into the index.

- Multilevel insertion (as well as deletion) algorithms are simple extensions of the single-level algorithms

B+ Tree Index Files

B+ tree indices are an alternative to indexed-sequential files.

Disadvantage of indexed-sequential files:

- Performance degrades as file grows, since many overflow blocks get created.
- Periodic reorganization of entire file is required.

Advantage of B+ tree index files:

- Automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
- Reorganization of entire file is not required to maintain performance.

(Minor) Disadvantage of B+ trees:

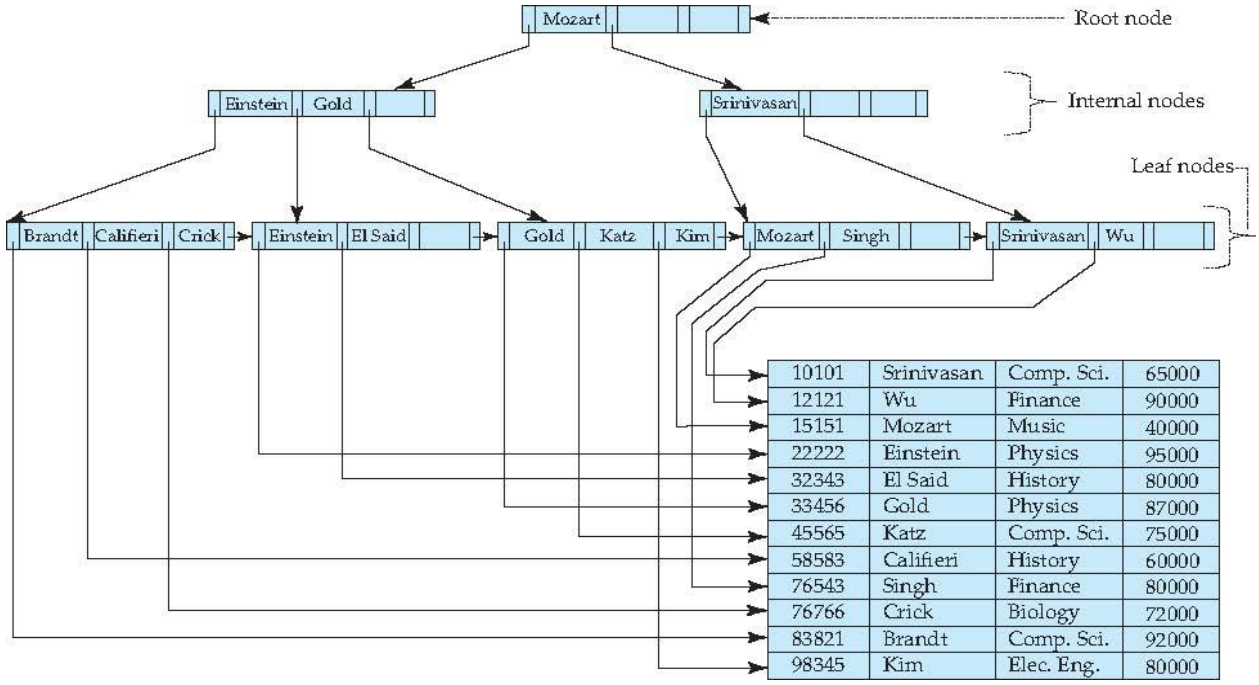
- Extra insertion and deletion overhead, space overhead.

Advantages of B+ trees outweigh disadvantages

- B+ trees are used extensively

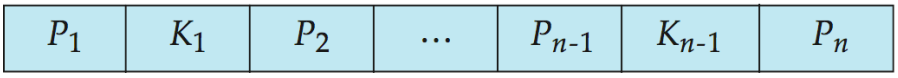
B+ tree is a rooted tree satisfying the following properties:

1. All paths from root to leaf are of the same length
2. Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
3. A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
4. Special cases:
   a) If the root is not a leaf, it has at least 2 children.
   b) If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values.



B+ Tree Node Structure

Typical node



- K_i are the search-key values
- P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

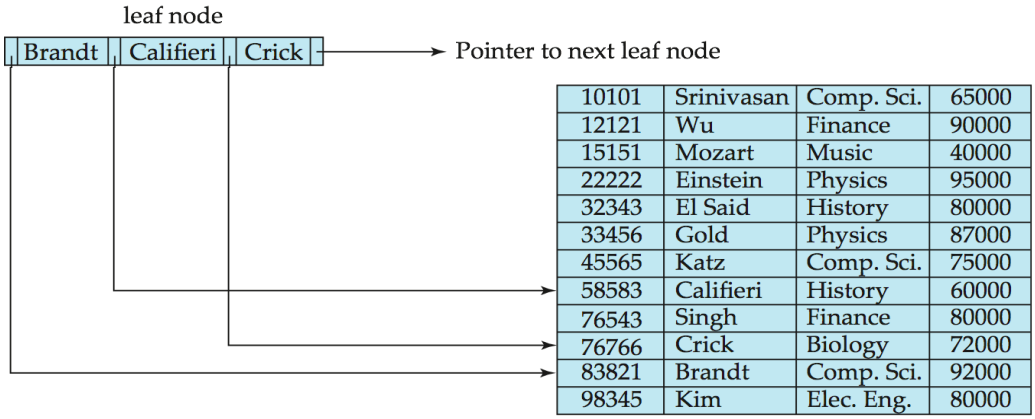
The search-keys in a node are ordered

$$K1 < K2 < K3 < \ldots < Kn-1$$

Leaf Nodes in B+ Trees

Properties of a leaf node:

1) For $i = 1, 2, \ldots, n-1$, pointer P_i either points to a file record with search-key value K_i, or to a bucket of pointers to file records, each record having search-key value K_i. Only need bucket structure if search-key does not form a primary key.
2) If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than L_j's search-key values
3) P_n points to next leaf node in search-key order



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

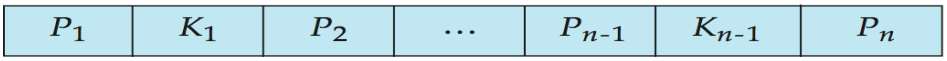
Non-Leaf Nodes in B+ Trees

Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:

a) All the search-keys in the sub tree to which $P1$ points are less than K_1

b) For $2 \le i \le n - 1$, all the search-keys in the sub tree to which P_i points have values greater than or equal to K_{i-1} and less than K_i

c) All search-keys in the sub tree to which P_n points have values greater than or equal to K_{n-1}



Example of a B+ tree

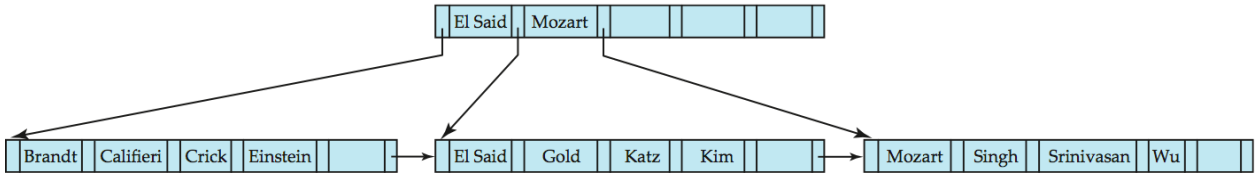


Fig: B+ tree for *instructor* file ($n = 6$)

- Leaf nodes must have between 3 and 4 values ($\lceil (n-1)/2 \rceil$ and $n - 1$, with $n = 6$).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil (n/2 \rceil$ and n with $n = 6$).
- Root must have at least 2 children.

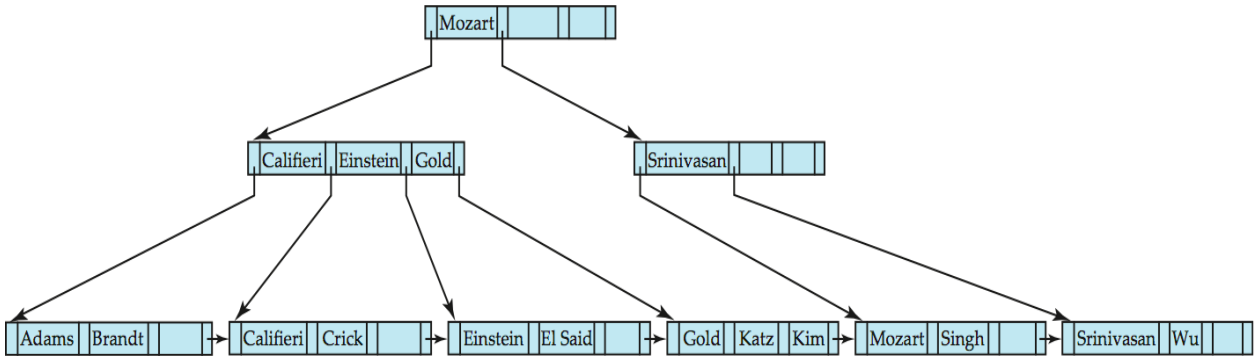
Observations about B+ trees

1. Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
2. The non-leaf levels of the B+ tree form a hierarchy of sparse indices.
3. The B+ tree contains a relatively small number of levels
   i. Level below root has at least $2 * \lceil n/2 \rceil$ values
   ii. Next level has at least $2 * \lceil n/2 \rceil * \lceil n/2 \rceil$ values
   a. If there are K search-key values in the file, tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
   b. Thus searches can be conducted efficiently.
4. Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time.

Queries on B+ trees

- Find record with search-key value *V.*
    1. *C=root*
    2. While C is not a leaf node {
        1. Let *i* be least value s.t. $V \leq K_i$.
        2. If no such exists, set *C = last non-null pointer in C*
        3. Else { if (V= K_i) Set C = P_{i+1} else set *C* = P_i}

        }
    1. Let *i* be least value s.t. $K_i = V$
    2. If there is such a value *i,* follow pointer P_i to the desired record.
    3. Else no record with search-key value *k* exists.



- If there are *K* search-key values in the file, the height of the tree is no more than $\lceil \log\lceil n/2 \rceil (K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
    - and *n* is typically around 100 (40 bytes per index entry).
- With 1 million search key values and *n* = 100
    - at most *log*50(1,000,000) = 4 nodes are accessed in a lookup.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
    - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

<u>Handling Duplicates</u>

With duplicate search keys

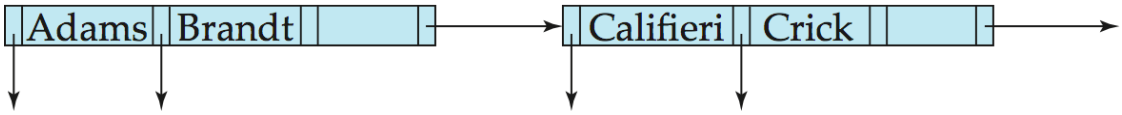
- In both leaf and internal nodes,
    - we cannot guarantee that $K_1 < K_2 < K_3 < \ldots < K_{n-1}$
    - but can guarantee $K_1 \leq K_2 \leq K_3 \leq \ldots \leq K_{n-1}$
- Search-keys in the subtree to which P_i points
    - are $\leq K_i$,, but not necessarily $< K_i$,
    - To see why, suppose same search key value *V* is present in two leaf node L_i and L_{i+1}. Then in parent node K_i must be equal to V

We modify find procedure as follows

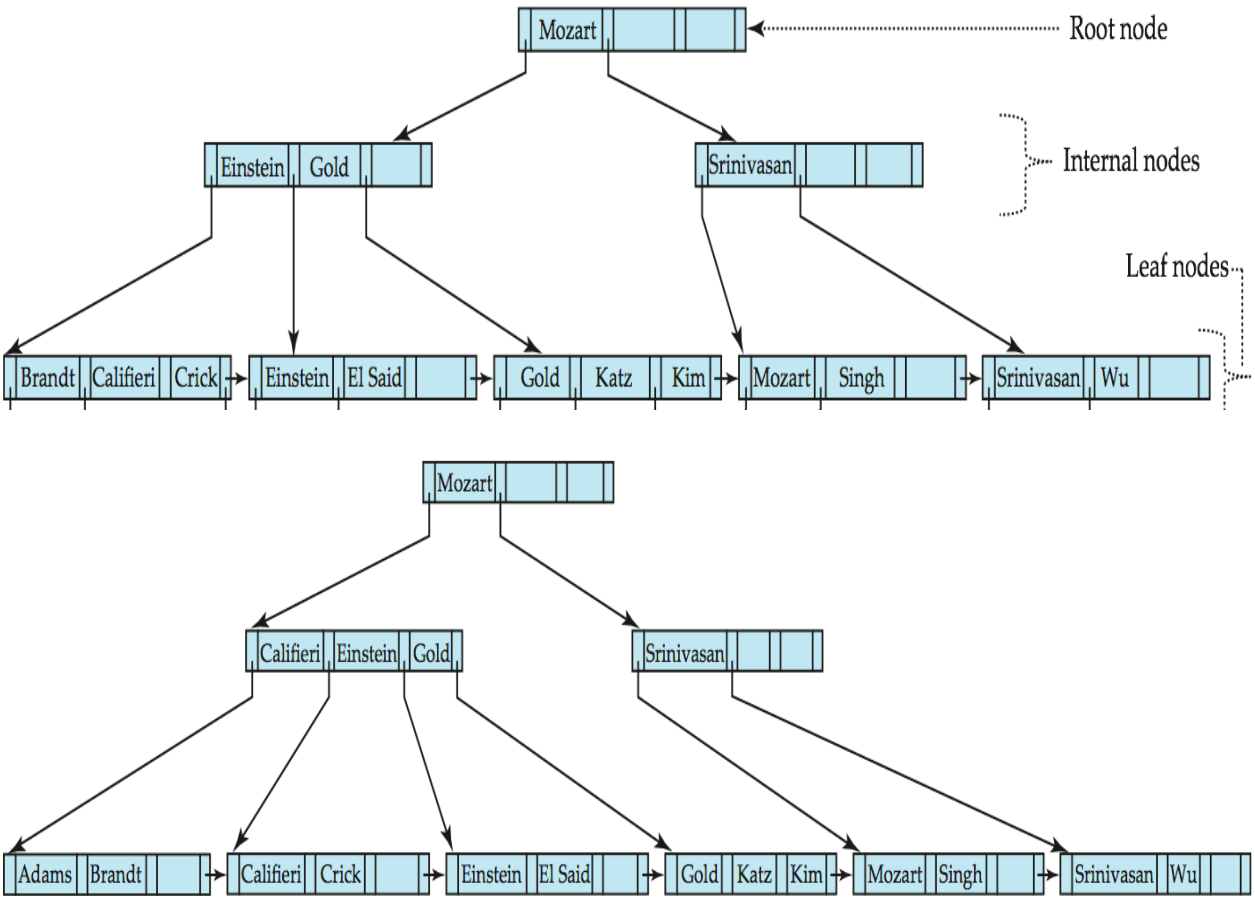
- traverse *Pi* even if *V* = *K*i
- As soon as we reach a leaf node *C* check if *C* has only search key values less than *V*
  if so set *C* = right sibling of *C* before checking whether *C* contains *V*
  Procedure printAll
- uses modified find procedure to find first occurrence of *V*
- Traverse through consecutive leaves to find all occurrences of *V*

Updates on B+ trees: Insertion

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
    1. Add record to the file
3. If the search-key value is not present, then
    1. add the record to the main file (and create a bucket if necessary)
    2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
    3. Otherwise, split the node (along with the new (key-value, pointer) entry)
4. Splitting a leaf node:
    1. take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
    2. let the new node be p, and let k be the least key value in p. Insert (k,p) in the parent of the node being split.
    3. If the parent is full, split it and **propagate** the split further up.
5. Splitting of nodes proceeds upwards till a node that is not full is found.
    1. In the worst case the root node may be split increasing the height of the tree by 1
    2.



Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri,pointer-to-new-node) into parent



B+ tree before and after insertion of "Adams"

pseudocode from book is given below!

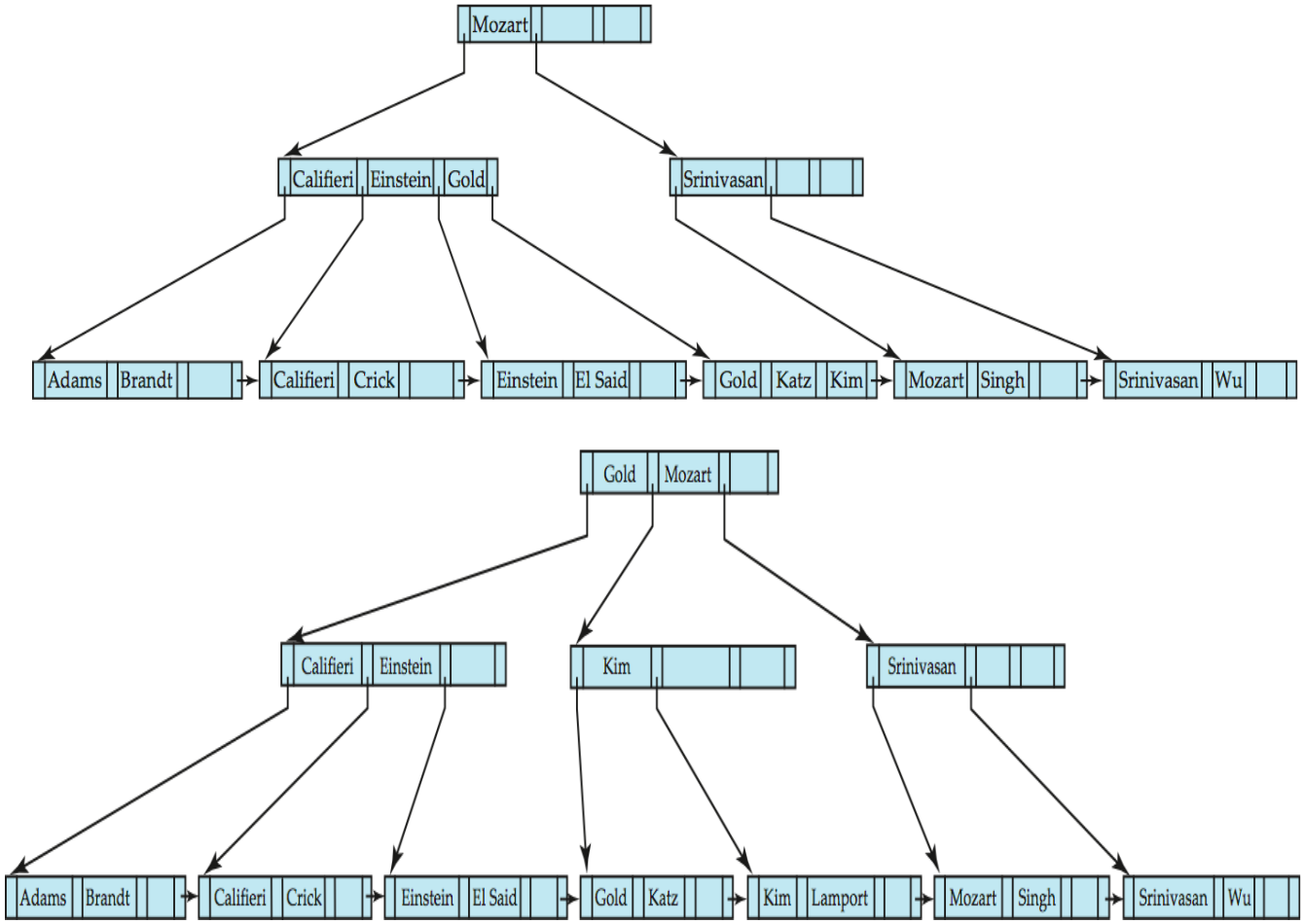
```
procedure insert(value K, pointer P)
if (tree is empty) create an empty leaf node L, which is also the root
else Find the leaf node L that should contain key value K
if (L has less than n − 1 key values)
    then insert in leaf (L, K, P)
    else begin /* L has n − 1 key values already, split it */
Create node L1
Copy L.P1 . . . L.Kn− 1 to a block of memory T that can hold n (pointer, key-value) pairs
insert in leaf (T, K, P)
Set L1.Pn = L.Pn; Set L.Pn = L1
Erase L.P1 through L.Kn− 1 from L
Copy T.P1 through T.K_n/2_ from T into L starting at L.P1
Copy T.P_n/2_+1 through T.Kn from T into L_ starting at L1.P1
Let K1 be the smallest key-value in L1
insert in parent(L, K1, L1)
end

procedure insert in leaf (node L, value K, pointer P)
if (K < L.K1)
then insert P, K into L just before L.P1
else begin
Let Ki be the highest value in L that is less than K
Insert P, K into L just after T.Ki
end

procedure insert in parent(node N, value K_, node N_)
if (N is the root of the tree)
then begin
Create a new node R containing N, K1, N1 /* N and N1 are pointers */
Make R the root of the tree
return
 end

Let P = parent (N)
if (P has less than n pointers)
then insert (K1, N1) in P just after N
else begin /* Split P */
Copy P to a block of memory T that can hold P and (K1, N1)
Insert (K1, N1) into T just after N
Erase all entries from P; Create node P_
Copy T.P1 . . . T.P_n/2_ into P
Let K" = T.K|n/2|
Copy T.P|n/2|+1 . . . T.Pn+1 into P1
  insert in parent(P, K", P1)
   end
```

Figure: Insertion of entry in a B+ tree.



B+ tree before and after insertion of "Lamport"

- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy P1,K1, …, K$\lceil n/2 \rceil$-1,P$\lceil n/2 \rceil$ from M back into node N
  - Copy P$\lceil n/2 \rceil$+1,K$\lceil n/2 \rceil$+1,…,Kn,Pn+1 from M into newly allocated node N'
  - Insert (K$\lceil n/2 \rceil$,N') into parent N

Updates on B+ trees: Deletion

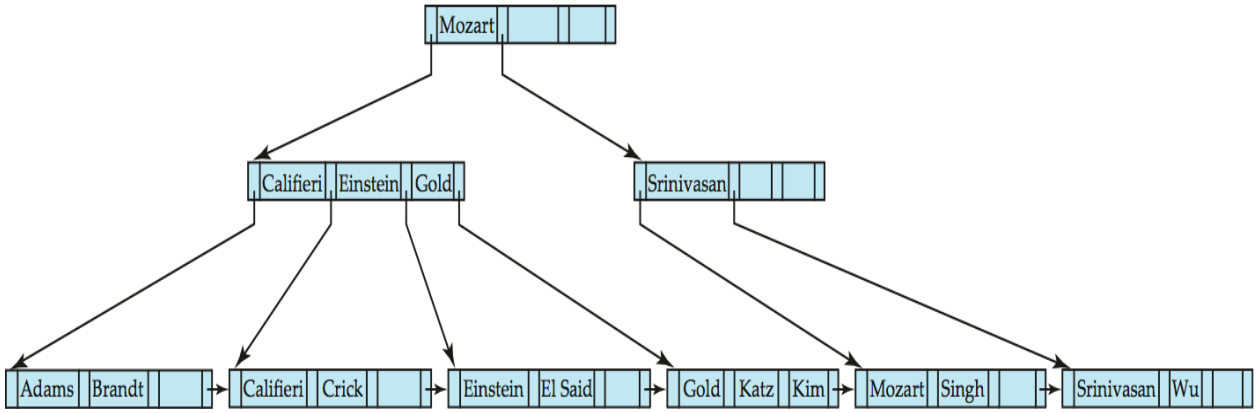
- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings:***
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (*Ki*–1, *Pi),* where *Pi* is the pointer to the deleted node, from its parent, recursively using the above procedure.

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:

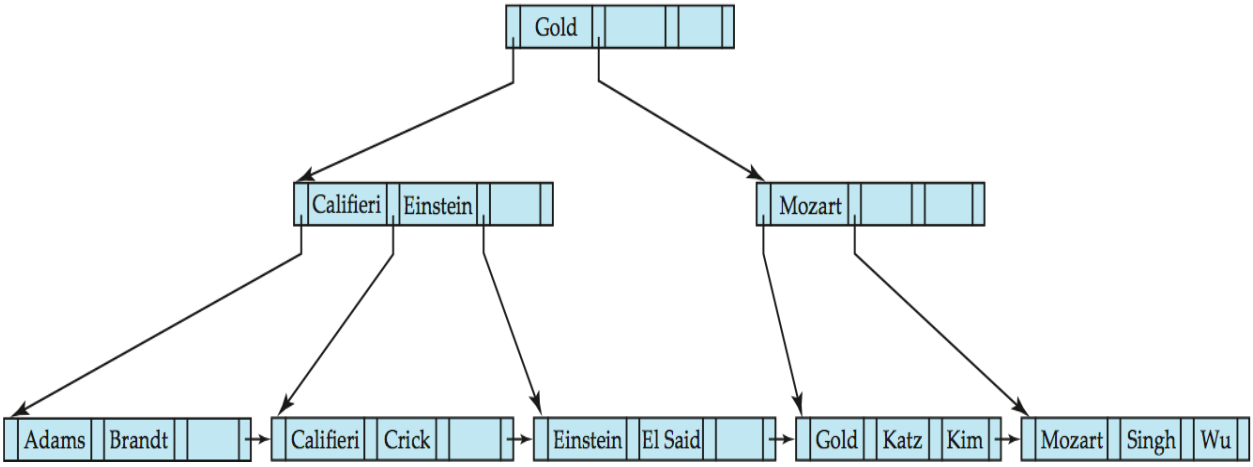
- Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
- Update the corresponding search-key value in the parent of the node.

- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

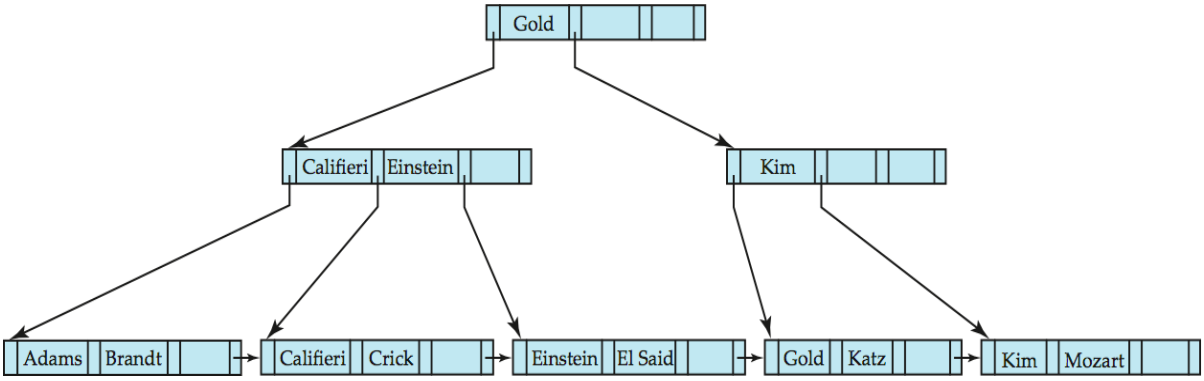
Examples of B+ tree Deletion



Before and after deleting "Srinivasan"

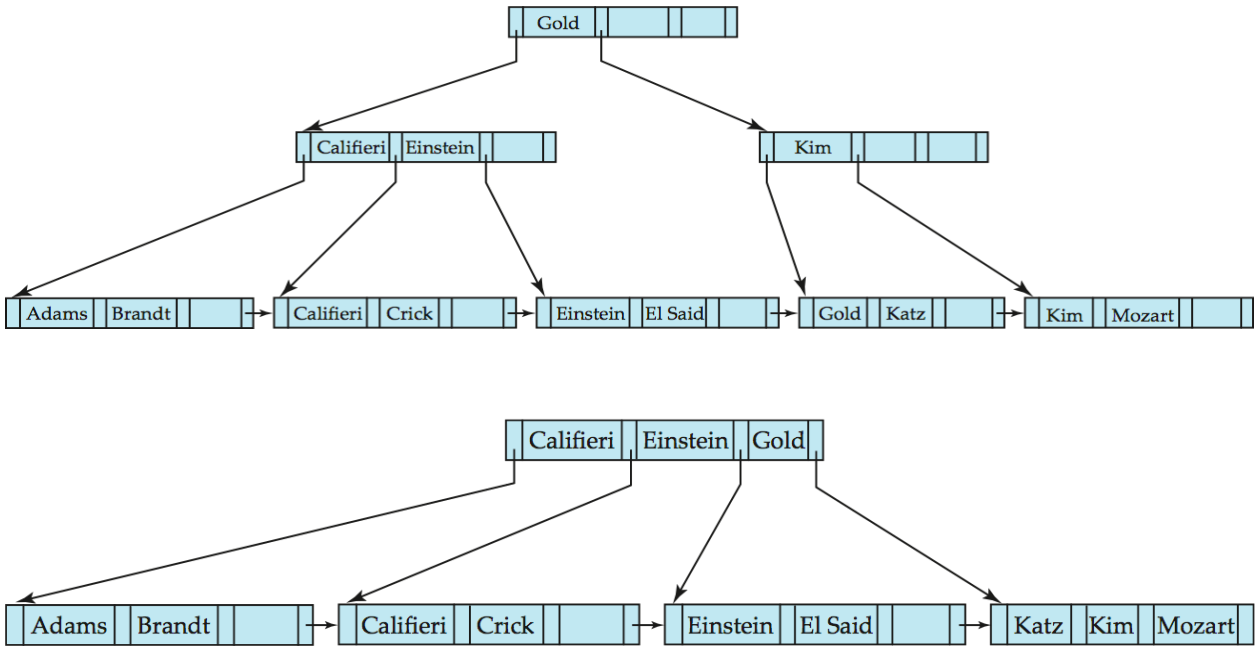


Deleting "Srinivasan" causes merging of under-full leaves.leaf node can become empty only for n=3!



Deletion of "Singh" and "Wu" from result of previous example

- Leaf containing Singh and Wu became underfull, and borrowed a value Kim from its left sibling
- Search-key value in the parent changes as a result



Before and after deletion of "Gold" from earlier example

- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
  - Value separating two nodes (at the parent) is pulled down when merging
- Root node then has only one child, and is deleted

Non-Unique Search Keys

- Alternatives to scheme described earlier
  - Buckets on separate block (bad idea)
  - List of tuple pointers with each key
    - Extra code to handle long lists
    - Deletion of a tuple can be expensive if there are many duplicates on search key
    - Low space overhead, no extra cost for queries
  - Make search key unique by adding a record-identifier
    - Extra storage overhead for keys
    - Simpler code for insertion/deletion
    - Widely used

B+ tree File Organization

- Index file degradation problem is solved by using B+ tree indices.
- Data file degradation problem is solved by using B+ tree File Organization.
- The leaf nodes in a B+ tree file organization store records, instead of pointers.
- Leaf nodes are still required to be half full
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B+ tree index.

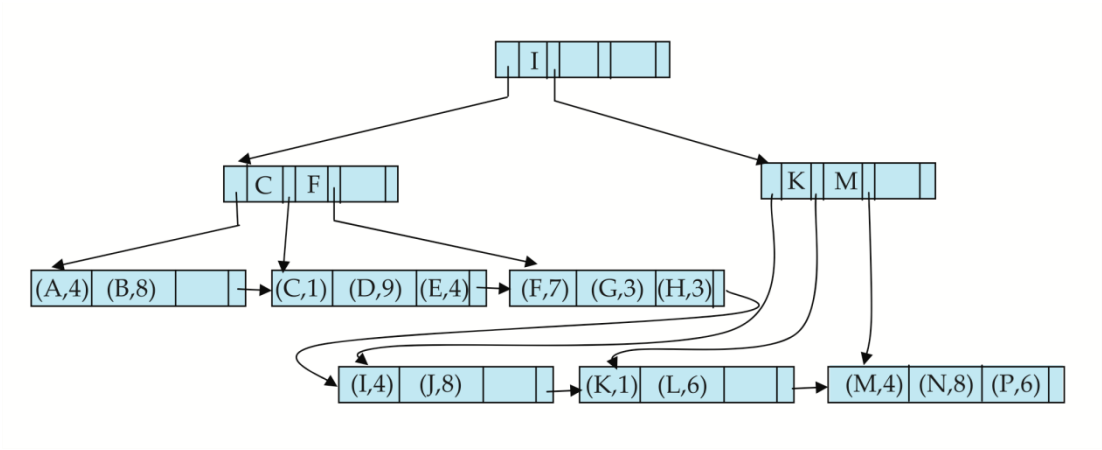


Fig: Example of B+ tree File Organization

- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries

Other Issues in Indexing

- **Record relocation and secondary indices**
  - If a record moves, all secondary indices that store record pointers have to be updated
  - Node splits in B+ tree file organizations become very expensive
  - *Solution*: use primary-index search key instead of record pointer in secondary index
    - Extra traversal of primary index to locate record
      - Higher cost for queries, but node splits are cheap
    - Add record-id if primary-index search key is non-unique

Indexing Strings

- Variable length strings as keys
  - Variable fanout
  - Use space utilization as criterion for splitting, not number of pointers
- **Prefix compression**
  - Key values at internal nodes can be prefixes of full key
    - Keep enough characters to distinguish entries in the sub trees separated by the key value
      - EX: "Silas" and "Silberschatz" can be separated by "Silb"
  - Keys in leaf node can be compressed by sharing common prefixes

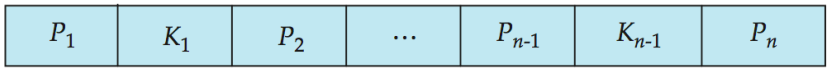
Bulk Loading and Bottom-Up Build

- Inserting entries one-at-a-time into a B+ tree requires ≥ 1 IO per entry
  - assuming leaf level does not fit in memory
  - can be very inefficient for loading a large number of entries at a time (**bulk loading**)
- Efficient alternative 1:
  - sort entries first (using efficient external-memory sort algorithms)
  - insert in sorted order
    - insertion will go to existing page (or cause a split)
    - much improved IO performance, but most leaf nodes half full

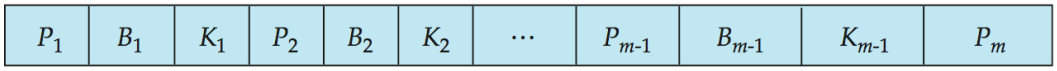
- Efficient alternative 2: **Bottom-up B+ tree construction**
  - As before sort entries
  - And then create tree layer-by-layer, starting with leaf level
    - details as an exercise
  - Implemented as part of bulk-load utility by most database systems

B-Tree Index Files

- Similar to B+ tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node



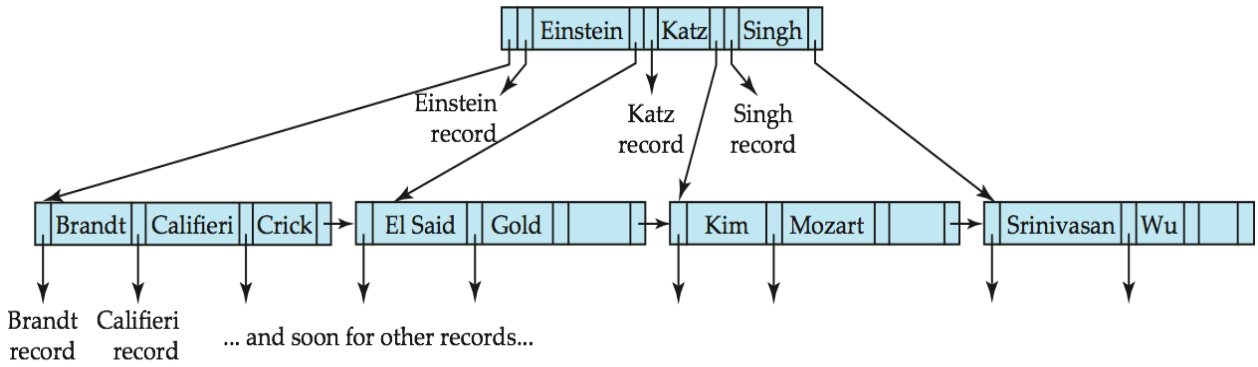
(a)



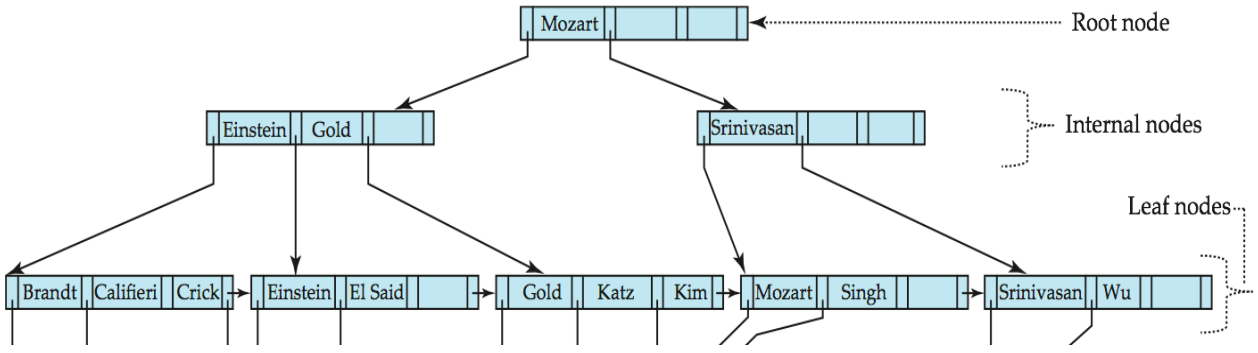
(b)

Nonleaf node – pointers Bi are the bucket or file record pointers.

B-Tree Index File Example



B-tree (above) and B+ tree (below) on same data



Advantages of B-Tree indices:

- May use less tree nodes than a corresponding B+ tree.
- Sometimes possible to find search-key value before reaching leaf node.

Disadvantages of B-Tree indices:

- Only small fraction of all search-key values are found early
- Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B+ tree
- Insertion and deletion more complicated than in B+ trees
- Implementation is harder than B+ trees.

Typically, advantages of B-Trees do not outweigh disadvantages.

Multiple-Key Access

Use multiple indices for certain types of queries.

Example:
select *ID*
from *instructor*
where *dept_name* = "Finance" **and** *salary* = 80000

Possible strategies for processing query using indices on single attributes:

1. Use index on *dept_name* to find instructors with department name Finance; test *salary = 80000*
2. Use index on *salary* to find instructors with a salary of $80000; test *dept_name* = "Finance".
3. Use *dept_name* index to find pointers to all records pertaining to the "Finance" department.

Similarly use index on *salary*. Take intersection of both sets of pointers obtained.

Indices on Multiple Keys

Composite search keys are search keys containing more than one attribute
EX: (*branch_name, balance*)
Lexicographic ordering: (a1, a2) < (b1, b2) if either

1. a1 < b1, or
2. a1=b1 and a2 < b2

Indices on Multiple Attributes

Suppose we have an index on combined search-key
(dept_*name, salary*).

With the **where** clause
where *dept_name* = "Finance" **and** *salary* = 80000

the index on (*dept_name, salary*) can be used to fetch only records that satisfy both conditions.

- Using separate indices in less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.

- Can also efficiently handle
  where *dept_name* = "Finance" **and** *salary* < 80000
- But cannot efficiently handle
  where *dept_name* < "Finance" **and** *balance* = 80000
  - May fetch many records that satisfy the first but not the second condition

Non-Unique Search Keys

Alternatives:

- Buckets on separate block (bad idea)
- List of tuple pointers with each key
  - Low space overhead, no extra cost for queries
  - Extra code to handle read/update of long lists
  - Deletion of a tuple can be expensive if there are many duplicates on search key
- Make search key unique by adding a record-identifier
  - Extra storage overhead for keys
  - Simpler code for insertion/deletion
  - Widely used

Other Issues in Indexing

Covering indices

Add extra attributes to index so (some) queries can avoid fetching the actual records

- Particularly useful for secondary indices
  - Why?

Can store extra attributes only at leaf

Record relocation and secondary indices

- If a record moves, all secondary indices that store record pointers have to be updated
- Node splits in B+ tree file organizations become very expensive
- *Solution*: use primary-index search key instead of record pointer in secondary index
  - Extra traversal of primary index to locate record
    - Higher cost for queries, but node splits are cheap
  - Add record-id if primary-index search key is non-unique

Hashing

Static Hashing

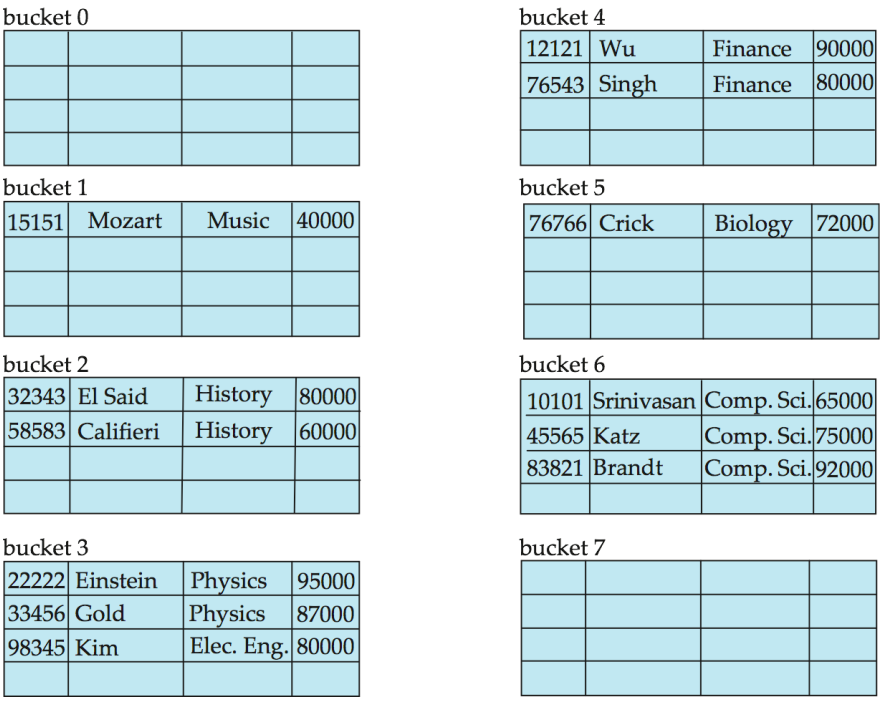
- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function.**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key

There are 10 buckets,

- The binary representation of the *i*th character is assumed to be the integer *i.*
- The hash function returns the sum of the binary representations of the characters modulo 10.
  EX: h(Music) = 1 h(History) = 2
  h(Physics) = 3 h(Elec. Eng.) = 3



Hash file organization of *instructor* file, using *dept_name* as key

Hash Functions

- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform***,* i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on internal binary representation of search-key.
    - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned. .

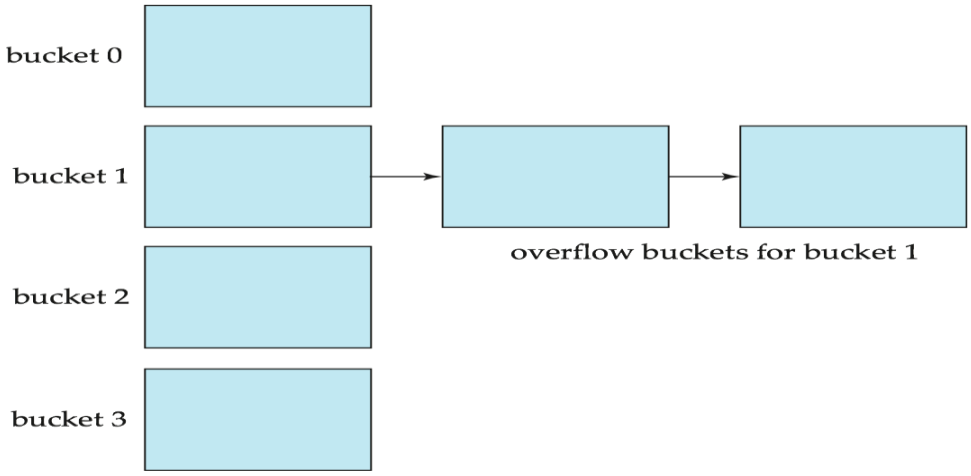
Handling of Bucket Overflows: Bucket overflow can occur because of

1. Insufficient buckets
2. Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values

Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using *overflow buckets.*

Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.
Above scheme is called ***closed hashing*****.**

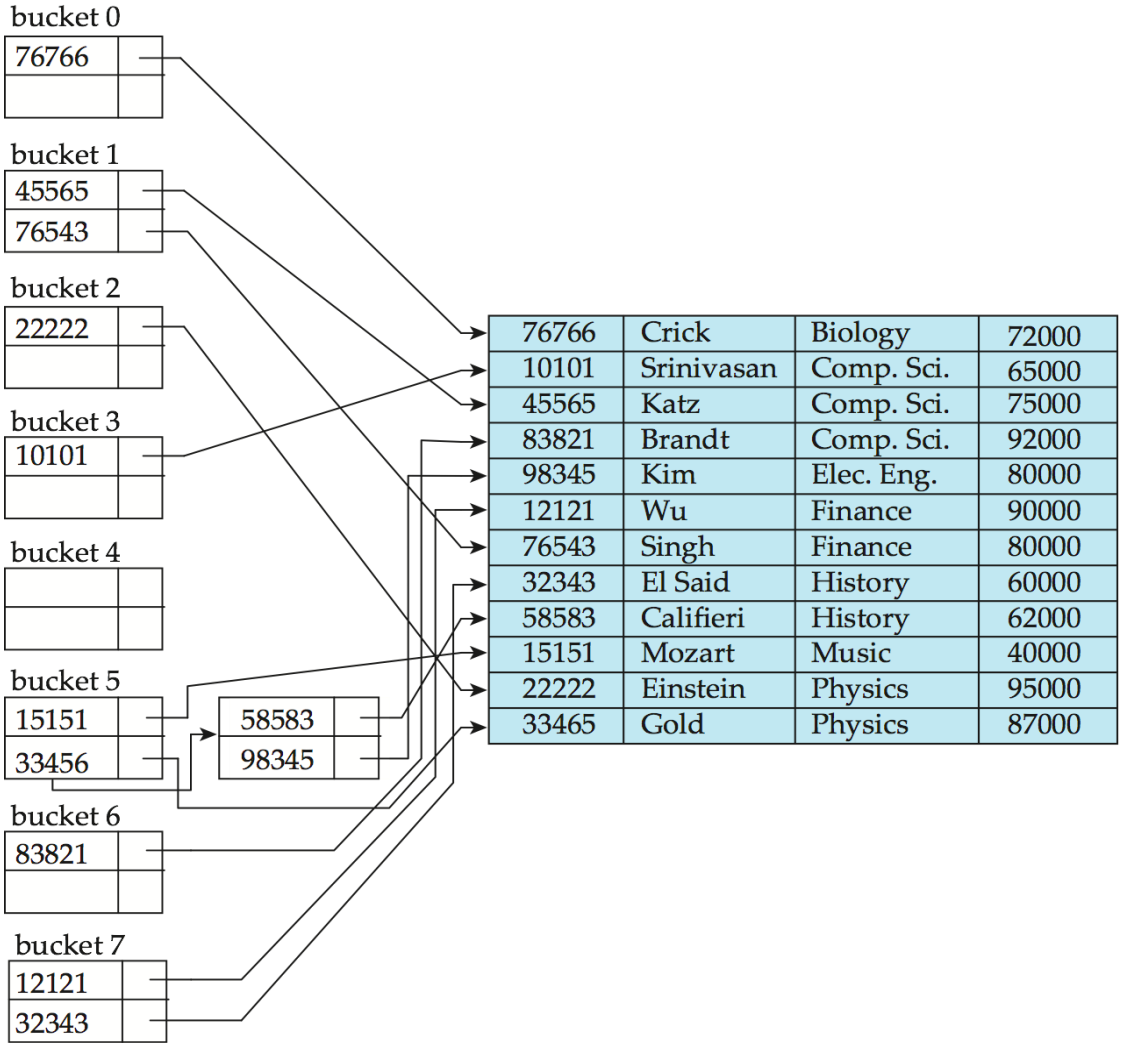
- An alternative, called ***open hashing***, which does not use overflow buckets, is not suitable for database applications.



Hash Indices: Hashing can be used not only for file organization, but also for index-structure creation.

- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary indices
  - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - However, we use the term hash index to refer to both secondary index structures and hash organized files.

Example of Hash Index: Hash index on *instructor,* on attribute *ID*



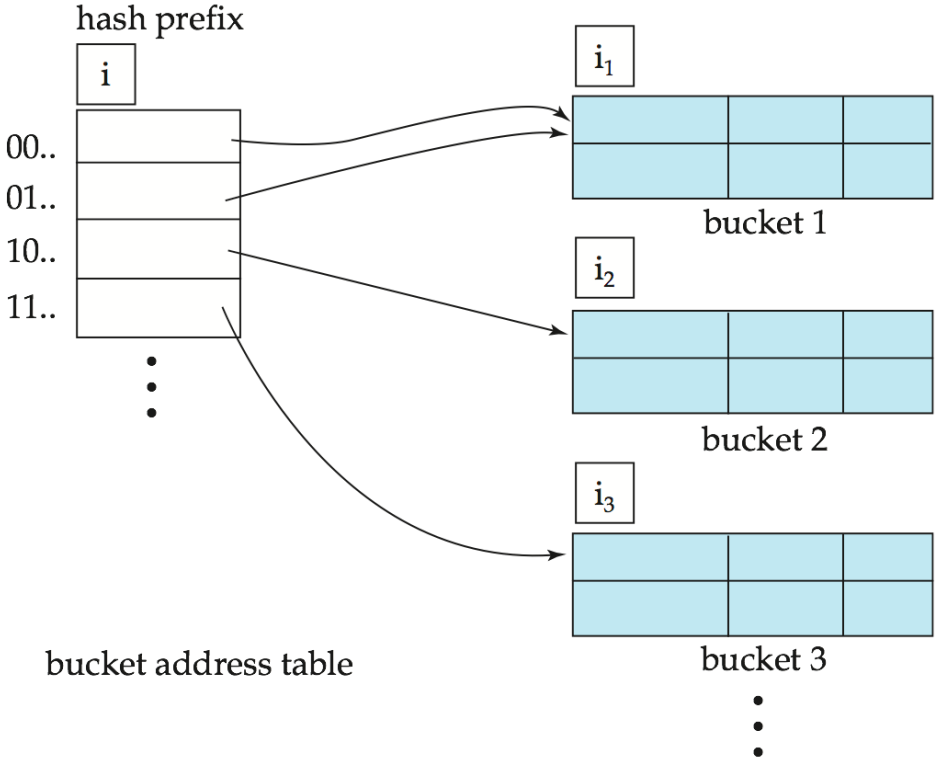
Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
  - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
  - Hash function generates values over a large range, typically *b*-bit integers, with *b*= 32.
  - At any time use only a prefix of the hash function to index into a table of bucket addresses.
  - Let the length of the prefix be *i* bits, $0 \le i \le 32$.
    - Bucket address table size = 2i. Initially *i* = 0
    - Value of *i* grows and shrinks as the size of the database grows and shrinks.
  - Multiple entries in the bucket address table may point to a bucket
  - Thus, actual number of buckets is < 2*i*
    - The number of buckets also changes dynamically due to coalescing and splitting of buckets.

General Extendable Hash Structure



In this structure, *i*2 = *i*3 = *i*, whereas *i*1 = *i* – 1

Use of Extendable Hash Structure

- Each bucket *j* stores a value i_j
  - All the entries that point to the same bucket have the same values on the first *ij* bits.
- To locate the bucket containing search-key K_j:
  - Compute *h(Kj) = X*
  - Use the first *i* high order bits of *X* as a displacement into bucket address table, and follow the pointer to appropriate bucket
- To insert a record with search-key value *Kj*
  - follow same procedure as look-up and locate the bucket, say *j*.
  - If there is room in the bucket *j* insert record in the bucket.
  - Else the bucket must be split and insertion re-attempted.
    - Overflow buckets used instead in some cases

Insertion in Extendable Hash Structure:

To split a bucket *j* when inserting record with search-key value *Kj*:

1. If *i* > *ij* (more than one pointer to bucket *j*)
   a. allocate a new bucket *z*, and set *ij* = *iz* = (*ij* + 1)
   b. Update second half of bucket address table entries originally pointing to *j,* to point to *z*
   c. remove each record in bucket *j* and reinsert (in *j* or *z*)
   d. recompute new bucket for *Kj* and insert record in the bucket (further splitting is required if the bucket is still full)
2. If *i* = *ij* (only one pointer to bucket *j*)
   a. If *i* reaches some limit *b*, or too many splits have happened in this insertion, create an overflow bucket
   b. Else
      i. increment *i* and double the size of the bucket address table.
      ii. replace each entry in the table by two entries that point to the same bucket.
      iii. recompute new bucket address table entry for K_j
      Now $i > i_j$ so use the first case above.

Deletion in Extendable Hash Structure:

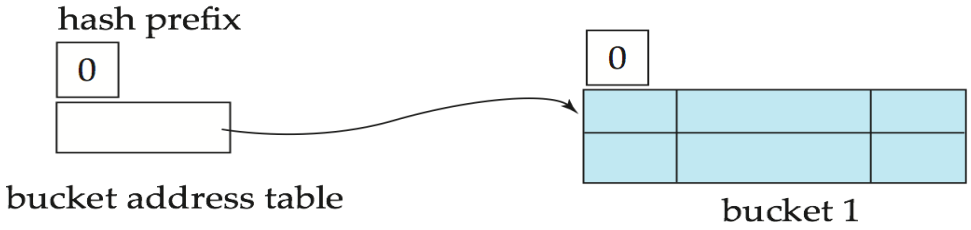
To delete a key value,

- locate it in its bucket and remove it.
- The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
- Coalescing of buckets can be done (can coalesce only with a "*buddy*" bucket having same value of ij and same ij –1 prefix, if it is present)
- Decreasing bucket address table size is also possible
  - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

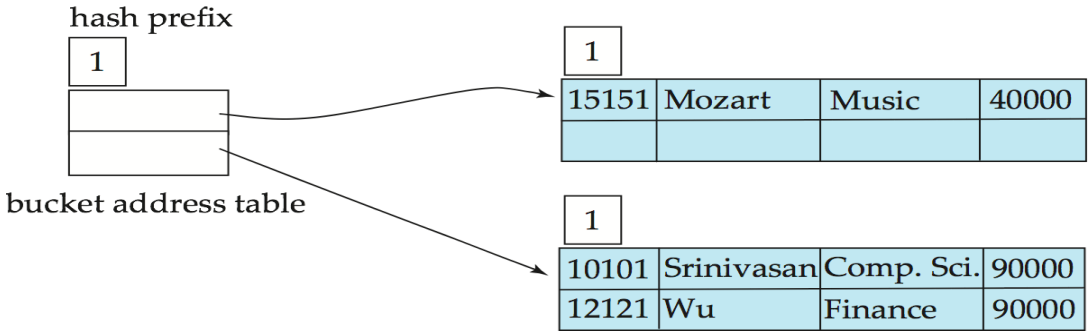
Use of Extendable Hash Structure: Example

dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

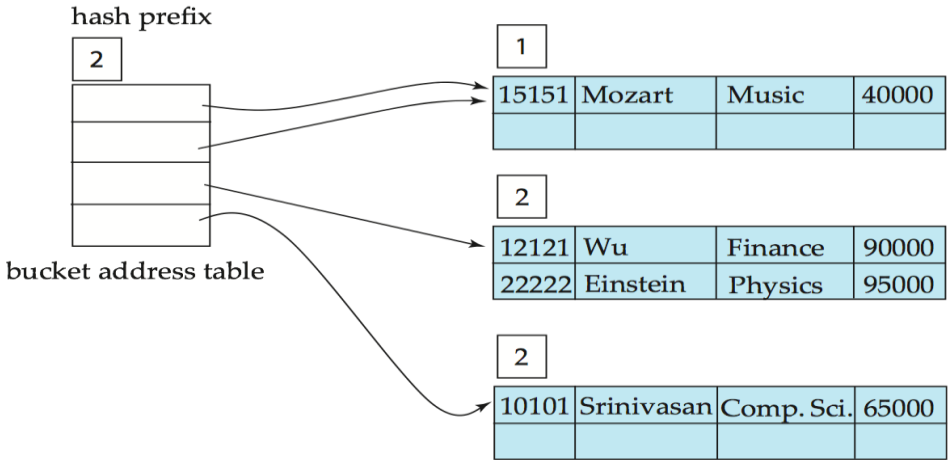
Initial Hash structure, bucket size = 2



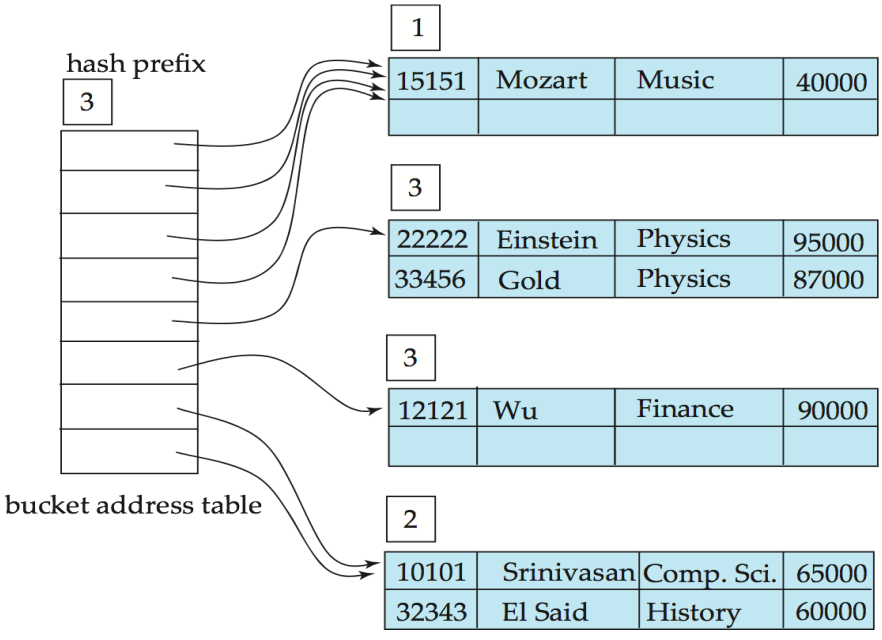
Hash structure after insertion of "Mozart", "Srinivasan", and "Wu" records



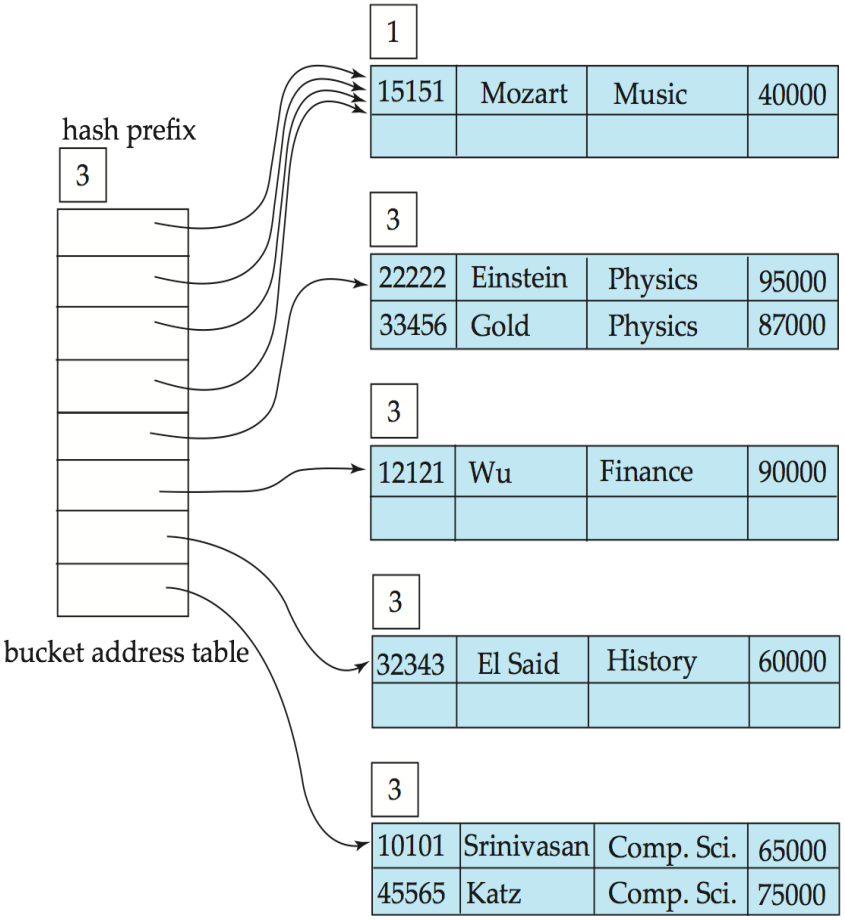
Hash structure after insertion of Einstein record



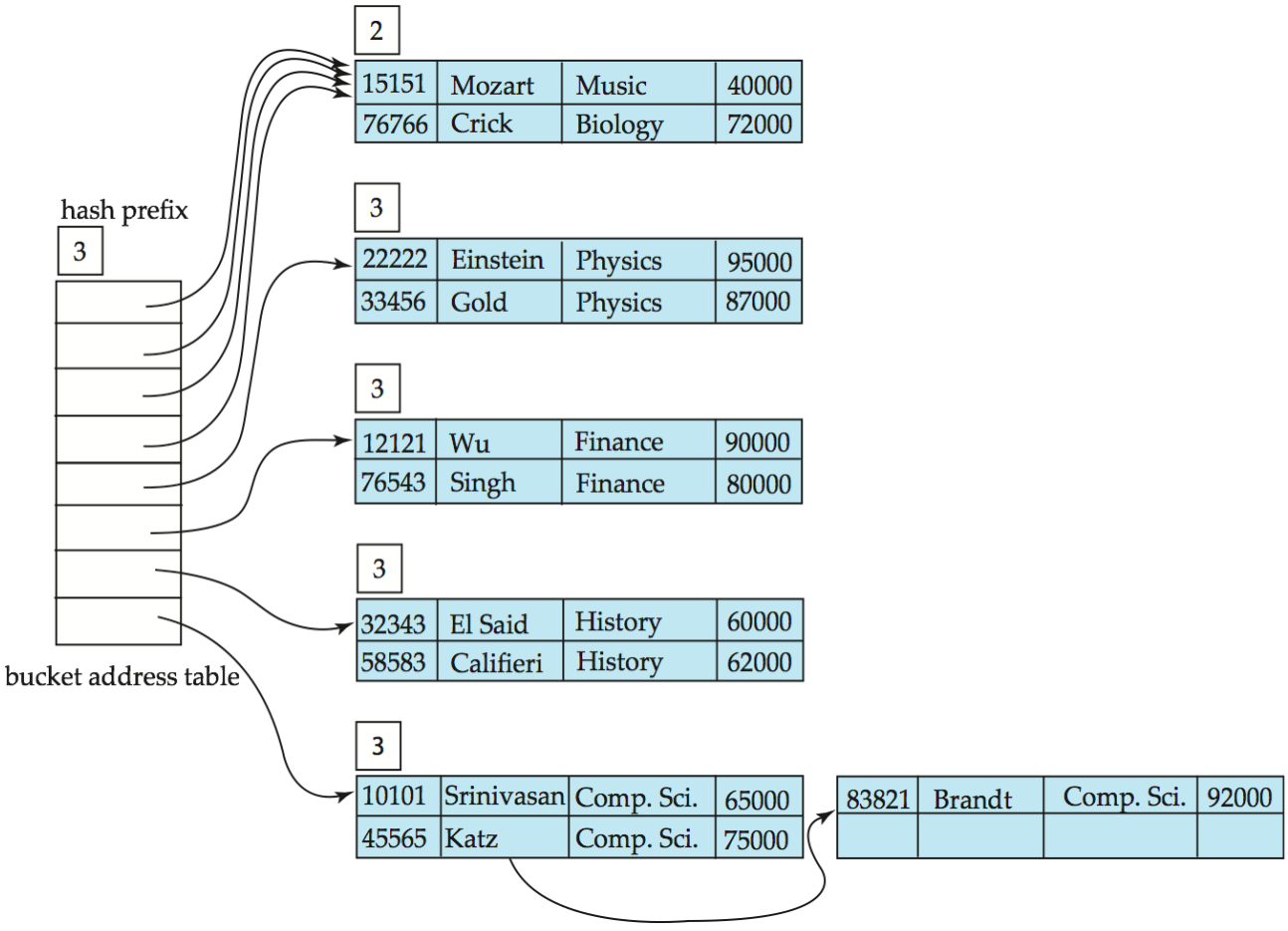
Hash structure after insertion of Gold and El Said records



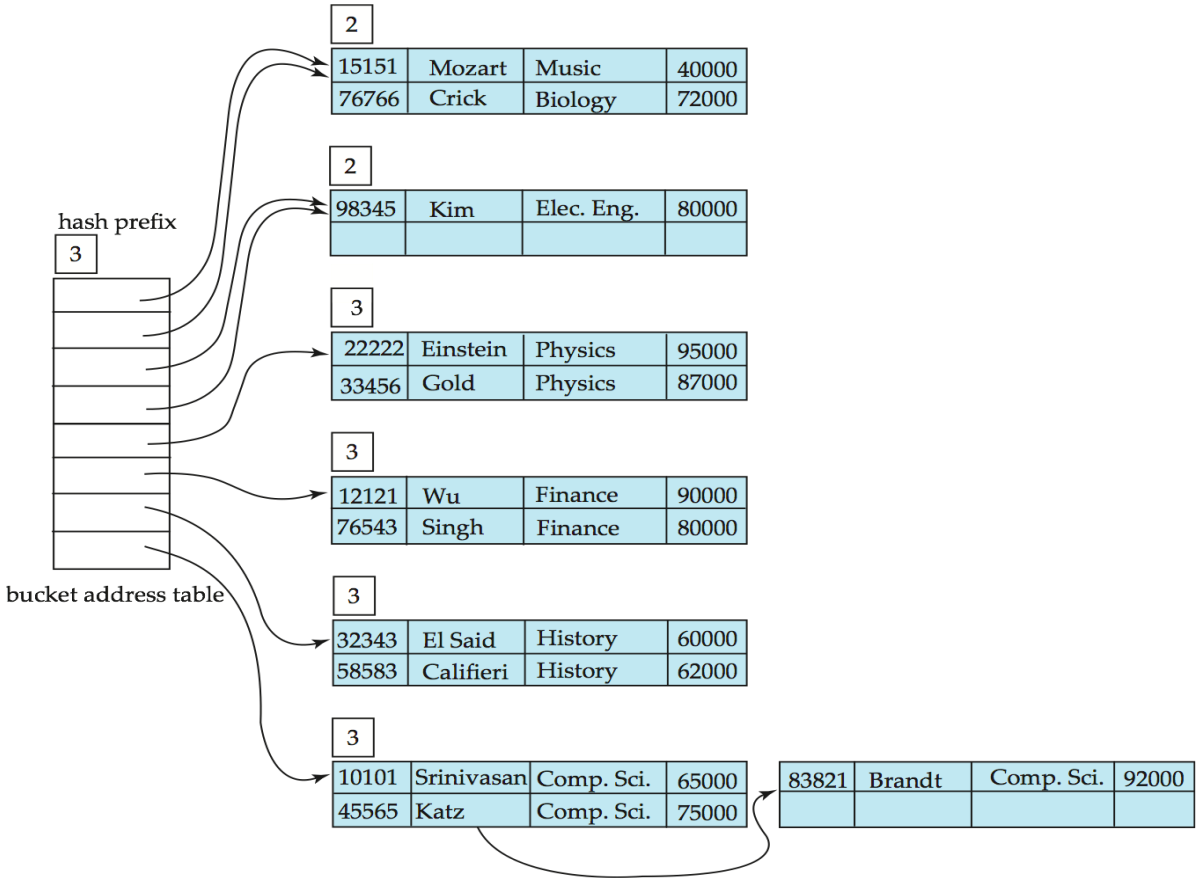
Hash structure after insertion of Katz record



And after insertion of eleven records



And after insertion of Kim record in previous hash structure



Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Cannot allocate very large contiguous areas on disk either
    - Solution: B+ tree file organization to store bucket address table
  - Changing size of bucket address table is an expensive operation
- **Linear hashing** is an alternative mechanism
  - Allows incremental growth of its directory (equivalent to bucket address table)
  - At the cost of more bucket overflows

Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred
- In practice:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B+ trees

Bitmap Indices: Bitmap indices are a special type of index designed for efficient querying on multiple keys

- Records in a relation are assumed to be numbered sequentially from, say, 0
  - Given a number *n* it must be easy to retrieve record *n*
    - Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
  - EX: gender, country, state, …
  - EX: income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits
- In its simplest form a bitmap index on an attribute has a bitmap for each value of attribute
  - Bitmap has as many bits as records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (and)
  - Union (or)
  - Complementation (not)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - EX: 100110 AND 110011 = 100010
    100110 OR 110011 = 110111
    NOT 100110 = 011001
  - Males with income level L1: 10010 AND 10100 = 10000
    - Can then retrieve required tuples.
    - Counting number of matching tuples is even faster
- Bitmap indices generally very small compared with relation size
  - EX: if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.
  - If number of distinct attribute values is 8, bitmap is only 1% of relation size
- Deletion needs to be handled properly
  - **Existence bitmap** to note if there is a valid record at a record location
  - Needed for complementation
    - not(*A*=*v*): *(NOT bitmap-A-v) AND ExistenceBitmap*
- Should keep bitmaps for all values, even null value
  - To correctly handle SQL null semantics for NOT(*A*=*v*):
    - intersect above result with (NOT *bitmap-A-Null*)

Efficient Implementation of Bitmap Operations

- Bitmaps are packed into words; a single word and (a basic CPU instruction) computes and of 32 or 64 bits at once
  - EX: 1-million-bit maps can be and-ed with just 31,250 instruction
- Counting number of 1s can be done fast by a trick:
  - Use each byte to index into a precomputed array of 256 elements each storing the count of 1s in the binary representation
    - Can use pairs of bytes to speed up further at a higher memory cost
  - Add up the retrieved counts
- Bitmaps can be used instead of Tuple-ID lists at leaf levels of B+ trees, for values that have a large number of matching records
  - Worthwhile if > 1/64 of the records have that value, assuming a tuple-id is 64 bits
  - Above technique merges benefits of bitmap and B+ tree indices

Index Definition in SQL

Create an index

create index <index-name> **on** <relation-name>
(<attribute-list>)

EX:: **create index** *b-index* **on** *branch(branch_name)*

Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key is a candidate key.

- Not really required if SQL **unique** integrity constraint is supported

To drop an index

drop index <index-name>

Most database systems allow specification of type of index, and clustering.

TRANSACTIONS

Transaction Concept: A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.

EX: transaction to transfer $50 from account A to account B:

```
read(A)
A := A – 50
write(A)
read(B)
B := B + 50
write(B)
```

Two main issues to deal with:

- Failures of various kinds, such as hardware failures and system crashes
- Concurrent execution of multiple transactions

Atomicity requirement — If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state

- Failure could be due to software or hardware

✓ System should ensure that updates of a partially executed transaction are not reflected in database

Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Consistency requirement in above example: the sum of A and B is unchanged by the execution of the transaction. In general, consistency requirements include

- Explicitly specified integrity constraints such as primary keys and foreign keys
- Implicit integrity constraints

– EX: sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand

- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency

Isolation requirement — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

	T1	**T2**
1.	**read**(*A*)	
2.	*A* := *A* – 50	
3.	**write**(*A*)	
		read(A), read(B), print(A+B)
4.	**read**(*B*)	
5.	*B* := *B* + 50	
6.	**write**(*B*	

Isolation can be ensured trivially by running transactions **serially**

-that is, one after the other.

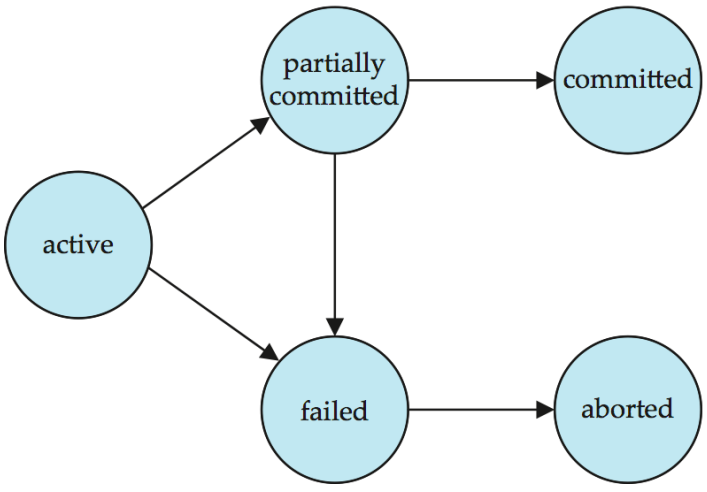
However, executing multiple transactions concurrently has significant benefits.

ACID Properties: A transaction is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

1. **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
2. **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
3. **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
    a. That is, for every pair of transactions *Ti* and *Tj,* it appears to *Ti* that either *Tj,* finished execution before *Ti* started, or *Tj* started execution after *Ti* finished.
4. **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State:

1. **Active** – the initial state; the transaction stays in this state while it is executing
2. **Partially committed** – after the final statement has been executed.
3. **Failed** -- after the discovery that normal execution can no longer proceed.
4. **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
    a. restart the transaction
        i. can be done only if no internal logical error
    b. kill the transaction
5. **Committed** – after successful completion.



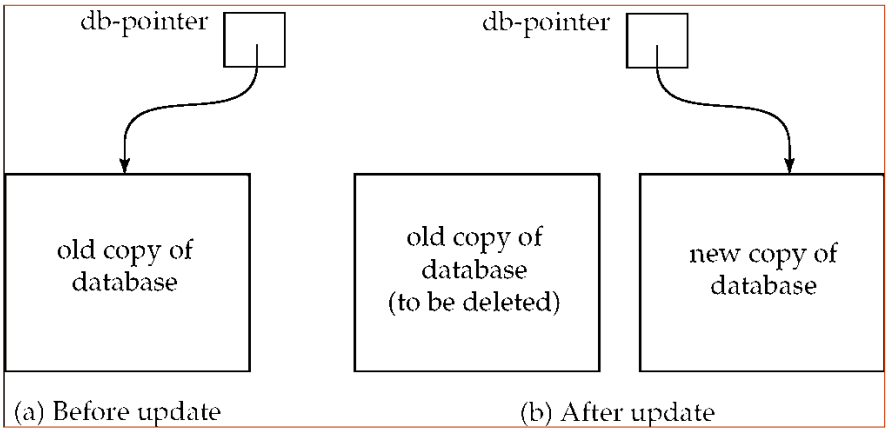
Implementation of Atomicity and Durability: The **recovery-management** component of a database system implements the support for atomicity and durability.

EX: the ***shadow-database*** scheme:

-all updates are made on a *shadow copy* of the database

db_pointer is made to point to the updated shadow copy after

- the transaction reaches partial commit and
- all updated pages have been flushed to disk.



db_pointer always points to the current consistent copy of the database.

-- In case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.

The shadow-database scheme:

- Assumes that only one transaction is active at a time.
- Assumes disks do not fail
- Useful for text editors, but
  - extremely inefficient for large databases
    - Variant called shadow paging reduces copying of data, but is still not practical for large databases
- Does not handle concurrent transactions

Concurrent Executions: Multiple transactions are allowed to run concurrently in the system. Advantages are:

- increased processor and disk utilization, leading to better transaction *throughput*

- EX: one transaction can be using the CPU while another is reading from or writing to the disk

- reduced average response time for transactions: short transactions need not wait behind long ones.

Concurrency control schemes – These are the mechanisms to achieve isolation. That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

Schedules: A sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed. a schedule for a set of transactions must consist of all instructions of those transactions and must preserve the order in which the instructions appear in each individual transaction.

A transaction that successfully completes its execution will have commit instructions as the last statement. by default transaction assumed to execute commit instruction as its last step. A transaction that fails to successfully complete its execution will have an abort instruction as the last statement.

Schedule 1: Let *T*1 transfer $50 from *A* to *B*, and *T*2 transfer 10% of the balance from *A* to *B.*

A serial schedule in which *T*1 is followed by *T*2:

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 2:

T_1	T_2
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	

Schedule 3:

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

Let *T*1 and *T*2 be the transactions defined previously. The above schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

Note: In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4: The following concurrent schedule does not preserve the value of (*A* + *B)*.

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability

Each transaction preserves database consistency. Thus serial execution of a set of transactions preserves database consistency. A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. **conflict serializability**
2. **view serializability**

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

Instructions *li* and *lj* of transactions *Ti* and *Tj* respectively, **conflict** if and only if there exists some item *Q* accessed by both *li* and *lj*, and at least one of these instructions wrote *Q.*

1. *li* = **read**(*Q*), *lj* = **read**(*Q*). *li* and *lj* don't conflict.
2. *li* = **read**(*Q*), *lj* = **write**(*Q*). They conflict.
3. *li* = **write**(*Q*), *lj* = **read**(*Q*). They conflict
4. *li* = **write**(*Q*), *lj* = **write**(*Q*). They conflict

Intuitively, a conflict between *li* and *lj* forces a (logical) temporal order between them. If *li* and *lj* are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability: If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are **conflict equivalent**.

We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule. Schedule 3 can be transformed into Schedule 6, a serial schedule where *T*2 follows *T*1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable

Schedule 3

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 6

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

We are unable to swap instructions in the above schedule to obtain either the serial schedule < *T*3, *T*4 >, or the serial schedule < *T*4, *T*3 >.

View Serializability

Let *S* and *S´* be two schedules with the same set of transactions. *S* and *S´* are **view equivalent** if the following three conditions are met, for each data item *Q,*

1. If in schedule S, transaction *Ti* reads the initial value of *Q*, then in schedule *S´* also transaction *Ti* must read the initial value of *Q.*
2. If in schedule S transaction *Ti* executes **read**(*Q*), and that value was produced by transaction *Tj* (if any), then in schedule *S´* also transaction *Ti* must read the value of *Q* that was produced by the same **write**(Q) operation of transaction *Tj* .
3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S´.*

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.
A schedule *S* is **view serializable** if it is view equivalent to a serial schedule. Every conflict serializable schedule is also view serializable. Below is a schedule which is view-serializable but *not* conflict serializable.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes.**

Other Notions of Serializability

The schedule below produces same outcome as the serial schedule < $T1$, $T5$ >, yet is not conflict equivalent or view equivalent to it.

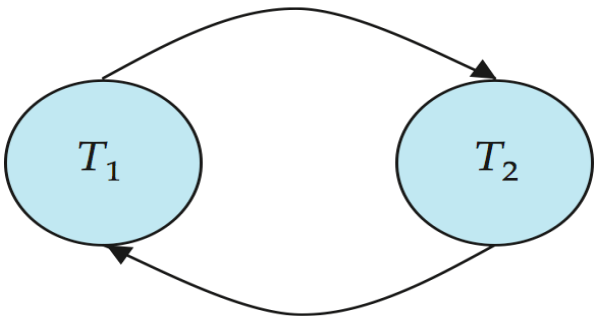
T_1	T_5
read (A)	
$A := A - 50$	
write (A)	
	read (B)
	$B := B - 10$
	write (B)
read (B)	
$B := B + 50$	
write (B)	
	read (A)
	$A := A + 10$
	write (A)

Determining such equivalence requires analysis of operations other than read and write.

Testing for Serializability

- Consider some schedule of a set of transactions $T1$, $T2$, ..., Tn
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from Ti to Tj if the two transaction conflict and Ti accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.

Example



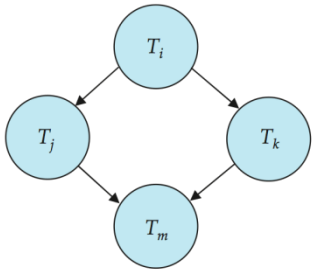
Example Schedule (Schedule A) + Precedence Graph

T1	*T2*	*T3*	*T4*	*T5*
read(Y) read(Z) read(U) read(U) write(U)	read(X) read(Y) write(Y)	 write(Z)	 read(Y) write(Y) read(Z) write(Z)	 read(V) read(W) read(W)

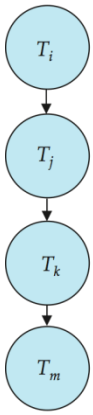
Test for Conflict Serializability

A schedule is conflict serializable if and only if its precedence graph is acyclic.

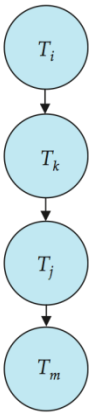
- Cycle-detection algorithms exist which take order $n2$ time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a Serializability order for Schedule A would be $T5 \rightarrow T1 \rightarrow T3 \rightarrow T2 \rightarrow T4$



(a)



(b)



(c)

Test for View Serializability

- The precedence graph test for conflict Serializability cannot be used directly to test for view Serializability.
  - Extension to test for view Serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  - Thus existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view Serializability can still be used.

Recoverable Schedules: Need to address the effect of transaction failures on concurrently running transactions.

Recoverable schedule — if a transaction *Tj* reads a data item previously written by a transaction *Ti* , then the commit operation of *Ti* appears before the commit operation of *Tj.*

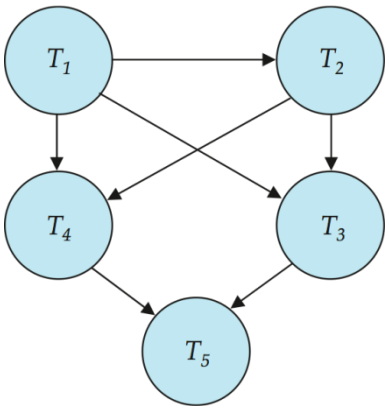
The following schedule (Schedule 11) is not recoverable if *T9* commits immediately after the read

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

If *T*8 should abort, *T*9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks: A single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		



If *T*10 fails, *T*11 and *T*12 must also be rolled back.

✓ Can lead to the undoing of a significant amount of work.

Cascadeless schedules — cascading rollbacks cannot occur; for each pair of transactions *Ti* and *Tj* such that *Tj* reads a data item previously written by *Ti*, the commit operation of *Ti* appears before the read operation of *Tj*.

Every cascadeless schedule is also recoverable. It is desirable to restrict the schedules to those that are cascadeless.

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for Serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless.
- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids nonseralizable schedules.
  - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for Serializability help us understand why a concurrency control protocol is correct.

Weak Levels of Consistency

Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable

- EX: a read-only transaction that wants to get an approximate total balance of all accounts
- EX: database statistics computed for query optimization can be approximate

Such transactions need not be serializable with respect to other transactions

- Tradeoff accuracy for performance

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Lower degrees of consistency useful for gathering approximate information about the database

- Warning: some database systems do not ensure serializable schedules by default

EX: Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)

Transaction Definition in SQL: In SQL, a transaction begins implicitly

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - EX: in JDBC, connection. setAutoCommit(false);