File Organization and Indexing

The data of a RDB is ultimately stored in disk files
Disks − non-volatile, inexpensive storage for data
− random-access addressable device

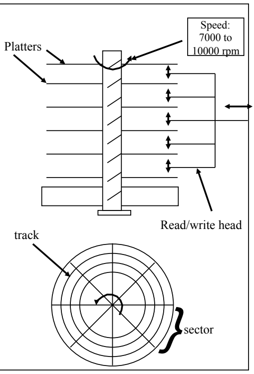
Disk space management:
Should Operating System services be used ?
Should RDBMS manage the disk space by itself ?

2^{nd} option is preferred as RDBMS requires complete control over when a block or page in main memory buffer is written to the disk.

This is important for recovering data when system crash occurs

Structure of Disks

Disk
- several platters stacked on a rotating spindle
- one read / write head per surface for fast access
- platter has several tracks
  - ~10,000 per inch
- each track - several sectors
- each sector/track - blocks
- unit of data transfer - block
- cylinder i - track i on all platters
- sectoring is optional
- block − ½ KB to 8KB
  - fixed; set at initialization time



Data Transfer from Disk

Address of a block: Surface No, Cylinder No, Block No

Data transfer:
Move the r/w head to the appropriate track
- time needed - seek time – ~ 12 to 14 ms

Wait for the appropriate block to come under r/w head
- time needed - rotational delay - ~3 to 4ms (avg)

Access time: Seek time + rotational delay
Blocks on the same cylinder - roughly close to each other
- access time-wise
- cylinder i, cylinder (i + 1), cylinder (i + 2) etc.

Data Records and Files

Fixed length record type: each field is of fixed length

- in a file of these type of records, the record number can be used to locate a specific record
- the number of records, the length of each field are available in file header

Variable length record type:

- arise due to missing fields, repeating fields, variable length fields
- special separator symbols are used to indicate the field boundaries and record boundaries
- the number of records, the separator symbols used are recorded in the file header

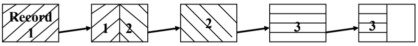
Packing Records into Blocks

Record length much less than block size

- The usual case
- Blocking factor b = $\lfloor B/r \rfloor$ B - block size (bytes)
  r - record length (bytes)
  - maximum no. of records that can be stored in a block

Record length greater than block size

- spanned organization is used



File blocks:
sequence of blocks containing all the records of the file

Mapping File Blocks onto the Disk Blocks

Contiguous allocation

- Consecutive file blocks are stored in consecutive disk blocks
- Pros: File scanning can be done fast using double buffering
  Cons: Expanding the file by including a new block in the middle of the sequence - difficult

Linked allocation

- each file block is assigned to some disk block
- each disk block has a pointer to next block of the sequence
- file expansion is easy; but scanning is slow

Mixed allocation

Operations on Files

Insertion of a new record: may involve searching for appropriate location for the new record

Deletion of a record: locating a record –may involve search; delete the record –may involve movement of other records

Update a record field/fields: equivalent to delete and insert

Search for a record: given value of a key field / non-key field

Range search: given range values for a key / non-key field

How successfully we can carry out these operations depends on the organization of the file and the availability of indexes

Primary File Organization

The logical policy / method used for placing records into file blocks

Example: *Student* file - organized to have students records sorted in increasing order of the "rollNo" values

Goal: To ensure that operations performed frequently on the file execute fast

- conflicting demands may be there
- example: on student file, access based on rollNo and also access based on name may both be frequent
- we choose to make rollNo access fast
- For making name access fast, additional access structures are needed.
  - more details later

Different File Organization Methods

We will discuss Heap files, Sorted files and Hashed files

Heap file:

Records are appended to the file as they are inserted
Simplest organization
Insertion - Read the last file block, append the record and write back the block - easy
Locating a record given values for any attribute

- requires scanning the entire file – very costly

Heap files are often used only along with other access structures.

Sorted files / Sequential files (1/2)

Ordering field: The field whose values are used for sorting the records in the data file

Ordering key field: An ordering field that is also a key

Sorted file / Sequential file:

Data file whose records are arranged such that the values of the ordering field are in ascending order

Locating a record given the value X of the ordering field:

Binary search can be performed

Address of the n^{th} file block can be obtained from the file header

O(log N) disk accesses to get the required block- efficient

Range search is also efficient

Sorted files / Sequential files (2/2)

Inserting a new record:

- Ordering gets affected
  - costly as all blocks following the block in which insertion is performed may have to be modified

- Hence not done directly in the file
  - all inserted records are kept in an auxiliary file
  - periodically file is reorganized - auxiliary file and main file are merged
  - locating record
    - carried out first on auxiliary file and then the main file.

Deleting a record

- deletion markers are used.

Hashed Files

Very useful file organization, if quick access to the data record is needed given the value of a single attribute.

Hashing field: The attribute on which quick access is needed and on which hashing is performed

Data file: organized as a buckets with numbers 0,1, …, (M − 1) (bucket - a block or a few *consecutive* blocks)

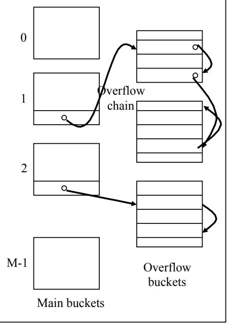
Hash function *h*: maps the values from the domain of the hashing attribute to bucket numbers

Inserting Records into a Hashed File

Insertion: for the given record R, apply *h* on the value of hashing attribute to get the bucket number *r*.

If there is space in bucket *r*, place R there else place R in the overflow chain of bucket *r*.

The overflow chains of all the buckets are maintained in the overflow buckets.

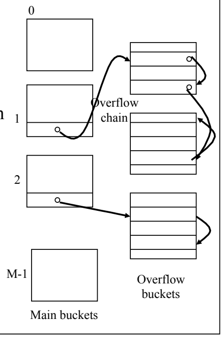


Deleting Records from a Hashed File

Deletion: Locate the record R to be deleted by applying *h*.

Remove R from its bucket/overflow chain. If possible, bring a record from the overflow chain into the bucket

Search: Given the hash filed value k, compute *r* = *h*(k). Get the bucket *r* and search for the record. If not found, search the overflow chain of bucket *r*.



Performance of Static Hashing

Static hashing:
- The hashing method discussed so far
- The number of main buckets is fixed

Locating a record given the value of the hashing attribute most often – one block access

Capacity of the hash file C = r * M records
(r - no. of records per bucket, M - no. of main buckets)

Disadvantage with static hashing:
If actual records in the file is much less than C
- wastage of disk space

If actual records in the file is much more than C
- long overflow chains – degraded performance

Hashing for Dynamic File Organization

Dynamic files
- files where record insertions and deletion take place frequently
- the file keeps growing and also shrinking

Hashing for dynamic file organization
- Bucket numbers are integers
- The binary representation of bucket numbers
  - Exploited cleverly to devise dynamic hashing schemes
  - Two schemes
    - Extendible hashing
    - Linear hashing

Extendible Hashing (1/2)

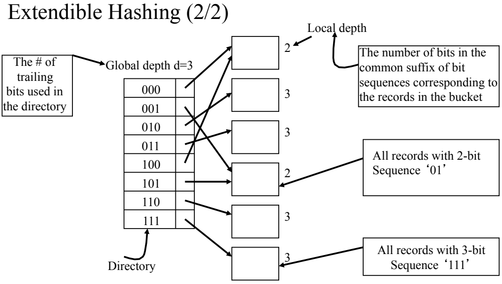
The *k*-bit sequence corresponding to a record R:

Apply hashing function to the value of the hashing field of R to get the bucket number *r*

Convert *r* into its binary representation to get the bit sequence
Take the *trailing k* bits

For instance, say record R hashes to bucket # 46
$46 = (101110)_2$
So, the 3-bit sequence corresponding to the bucket is "110"

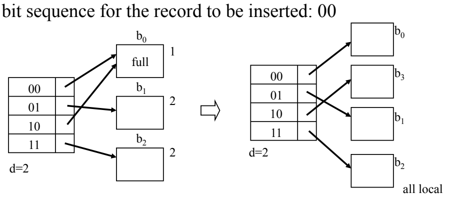
Extendible Hashing (2/2)



Locating a record
Match the d-bit sequence with an entry in the directory and go to the corresponding bucket to find the record

Insertion in Extendible Hashing Scheme (1/2)

2 - bit sequence for the record to be inserted: 00



b_0 Full: Bucket b_0 is split

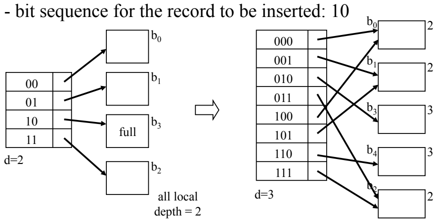
All records whose 2-bit sequence is '10' are sent to a new bucket b_3. Others are retained in b_0

Directory is modified.

b_0 Not full: New record is placed in b_0. No changes in the directory.

Insertion in Extendible Hashing Scheme (2/2)

2 - bit sequence for the record to be inserted: 10

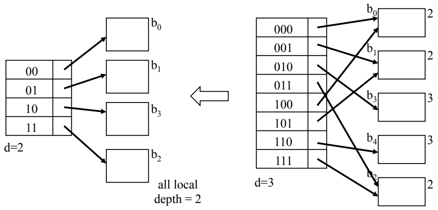


b_3 not full: new record placed in b_3. No changes.

b_3 full : b_3 is split, directory is doubled, all records with 3-bit sequence 110 sent to b_4. Others in b_3.

In general, if the local depth of the bucket to be split is equal to the global depth, directory is doubled

Deletion in Extendible Hashing Scheme



Matching pair of data buckets:

k-bit sequences have a common k-1 bit suffix, e.g, b_3 & b_4

Due to deletions, if a pair of matching data buckets

-- become less than half full – *try* to merge them into one bucket

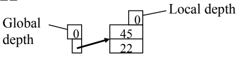
If the local depth of all buckets is one less than the global depth

-- reduce the directory to half its size

Extendible Hashing Example

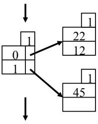
Bucket capacity – 2 Initial buckets = 1

Insert 45,22



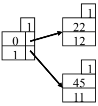
45	101101
22	10110
12	1100
11	1011

Insert 12

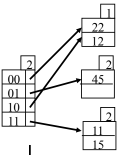


Bucket overflows
local depth = global depth
⇒ Directory doubles and split image is created

Insert 11



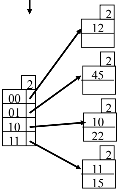
Insert 15



Overflow occurs.
Global depth = local depth
Directory doubles and split occurs

45	101101
22	10110
12	1100
11	1011
15	1111
10	1010

Insert 10



Overflows occurs.
Since local depth < global depth
Split image is created
Directory is not doubled

Linear Hashing

Does not require a separate directory structure

Uses a family of hash functions h_0, h_1, h_2,….
- the range of h_i is double the range of h_{i-1}

- $h_i(x) = x \bmod 2^iM$
  M - the initial no. of buckets
  (Assume that the hashing field is an integer)

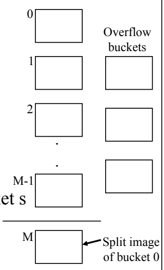
Initial hash functions
$h_0(x) = x \bmod M$
$h_1(x) = x \bmod 2M$

Insertion (1/3)

Initially the structure has M main buckets (0 ,…, M-1) and a few overflow buckets

To insert a record with hash field value x, place the record in bucket $h_o(x)$

When the *first* overflow in <u>any</u> bucket occurs:
- Say, overflow occurred in bucket s
- Insert the record in the overflow chain of bucket s
- Create a new bucket M
- Split the *bucket 0* by using h_1
  - Some records stay in bucket 0 and some go to bucket M.



Insertion (2/3)

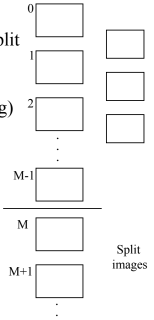
On first overflow,
- irrespective of where it occurs, bucket 0 is split

On subsequent overflows
- buckets 1, 2, 3, … are split in that order
- (This why the scheme is called linear hashing)

N: the next bucket to be split

After M overflows,
- all the original M buckets are split.
- We switch to hash functions h_1, h_2
  - and set N = 0.

$$\begin{matrix} h_o \\ h_1 \end{matrix} \longrightarrow \begin{matrix} h_1 \\ h_2 \end{matrix} \longrightarrow \cdots \quad \begin{matrix} h_i \\ h_{i+1} \end{matrix} \longrightarrow \cdots$$



Nature of Hash Functions

$h_i(x) = x \bmod 2^iM$. Let $M' = 2^iM$

- Note that if $h_i(x) = k$ then $x = M'r + k$, $k < M'$

  and $h_{i+1}(x) = (M'r + k) \bmod 2M'$

  $= k$ or $M' + k$

Since,
r – even – $(M'2s + k) \bmod 2M' = k$
r – odd – $(M'(2s + 1) + k) \bmod 2M' = M' + k$

  M'– the current number of original buckets.

Insertion (3/3)

Say the hash functions in use are h_i, h_{i+1}

To insert record with hash field value x,

Compute $h_i(x)$

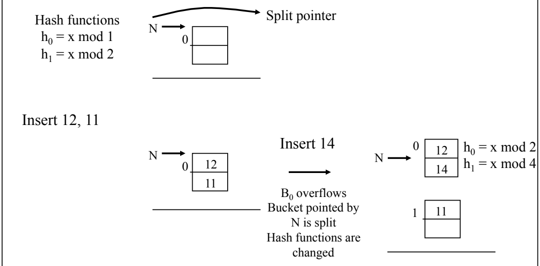
if $h_i(x) < N$, the original bucket is already split

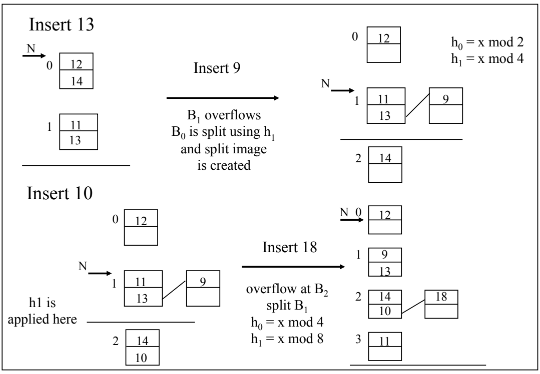
place the record in bucket $h_{i+1}(x)$

else place the record in bucket $h_i(x)$

Linear Hashing Example

Initial Buckets = 1   Bucket capacity = 2 records





Index Structures

Index: A disk data structure
– enables efficient retrieval of a record
given the value (s) of certain attributes
– indexing attributes

Primary Index:
Index built on *ordering key* field of a file

Clustering Index:
Index built on *ordering non-key* field of a file

Secondary Index:
Index built on any *non-ordering* field of a file

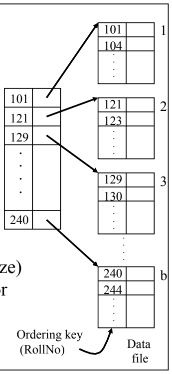
Primary Index

Can be built on ordered / sorted files
Index attribute – ordering key field (OKF)

Index Entry:

value of OKF for the <u>first record</u> of a block B_j	disk address of B_j

Index file: ordered file (sorted on OKF)
size: no. of blocks in the data file
Index file blocking factor $BF_i = \lfloor B/(V+P) \rfloor$
(B-block size, V-OKF size, P-block pointer size)
- generally more than data file blocking factor
No of Index file blocks $b_i = \lceil b/BF_i \rceil$
(b - no. of data file blocks)



Record Access Using Primary Index

Given Ordering key field (OKF) value: x
Carry out binary search on the index file
m – value of OKF for the first record in the *middle block k* of the index file
$x < m$: do binary search on blocks $1, \ldots, (k-1)$ of index file
$x \geq m$: if there are an index entries (v_j, P_j), (v_{j+1}, P_{j+1}) in block k such that $v_j \leq x < v_{(j+1)}$,
use the block pointer P_j, get the data file block and search for the data record with OKF value x
else
do binary search on blocks $k+1, \ldots, b_i$ of index file

Maximum block accesses required: $\lceil \log_2 b_i \rceil$

An Example

Data file:
No. of blocks b = 9500
Block size B = 4KB
OKF length V = 15 bytes
Block pointer length p = 6 bytes
Index file
No. of records $r_i = 9500$
Size of entry V + P = 21 bytes
Blocking factor $BF_i = \lfloor 4096/21 \rfloor = 195$
No. of blocks $b_i = \lceil r_i/BF_i \rceil = 49$

Max No. of block accesses for getting record using the primary index | $1 + \lceil \log_2 b_i \rceil = 7$

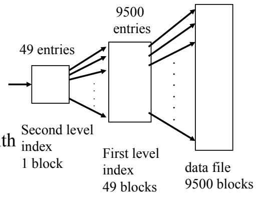
Max No. of block accesses for getting record without using primary index | $\lceil \log_2 b \rceil = 14$

Making the Index Multi-level

Index file – itself an ordered file
– another level of index can be built
Multilevel Index –
Successive levels of indices are built till the last level has one block

height – no. of levels
block accesses: height + 1
(no binary search required)

For the example data file:
No of block accesses required with multi-level primary index: 3
without any index: 14



Range Search, Insertion and Deletion

Range search on the ordering key field:
Get records with OKF value between x_1 and x_2 (inclusive)
Use the index to locate the record with OKF value x_1 and read succeeding records till OKF value exceeds x_2.
Very efficient

Insertion: Data file – keep 25% of space in each block free
-- to take care of future insertions
index doesn't get changed
-- or use overflow chains for blocks that overflow

Deletion: Handle using deletion markers so that index doesn't get affected
Basically, avoid changes to index

Clustering Index

Built on ordered files where ordering field is *not a key*
Index attribute: ordering field (OF)

Index entry:

Distinct value V_i of the OF	address of the first block that has a record with OF value V_i

Index file: Ordered file (sorted on OF)
size – no. of distinct values of OF

Secondary Index

Built on any non-ordering field (NOF) of a data file.
Case I: NOF is also a key (Secondary key)

value of the NOF V_i	pointer to the record with V_i as the NOF value

Case II: NOF is not a key: two options

(1)

value of the NOF V_i	pointer(s) to the record(s) with V_i as the NOF value

(2)

value of the NOF V_i	pointer to a block that has pointer(s) to the record(s) with V_i as the NOF value

Remarks:
(1) index entry – variable length record
(2) index entry – fixed length – One more level of indirection

Secondary Index (key)

Can be built on ordered and also other type of files
Index attribute: non-ordering key field
Index entry:

value of the NOF V_i	pointer to the *record* with V_i as the NOF value

Index file: ordered file (sorted on NOF values)
No. of entries – same as the no. of *records* in the data file

Index file blocking factor $Bf_i = \lfloor B/(V+P_r) \rfloor$
(B: block size, V: length of the NOF,
P_r: length of a record pointer)

Index file blocks = $\lceil r/Bf_i \rceil$
(r – no. of records in the data file)

An Example

Data file:

No. of records r = 90,000 | Block size B = 4KB

Record length R = 100 bytes | BF = $\lfloor 4096/100 \rfloor = 40$,

$b = \lceil 90000/40 \rceil = 2250$

NOF length V = 15 bytes | length of a record pointer P_r = 7 bytes

Index file :

No. of records r_i = 90,000 | record length = V + P_r = 22 bytes

$BF_i = \lfloor 4096/22 \rfloor = 186$ | No. of blocks $b_i = \lceil 90000/186 \rceil = 484$

Max no. of block accesses to get a record using the secondary index: $1 + \lceil \log_2^{b_i} \rceil = 10$

Avg no. of block accesses to get a record without using the secondary index: b/2 = 1125

A very significant improvement

Multi-level Secondary Indexes

Secondary indexes can also be converted to multi-level indexes

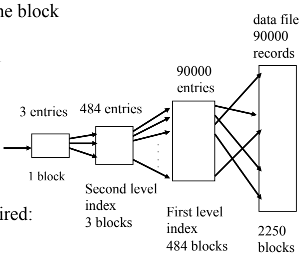
First level index
– as many entries as there are records in the data file

First level index is an ordered file
so, in the second level index, the number of entries will be equal to the number of *blocks* in the first level index rather than the number of *records*

Similarly in other higher levels

Making the Secondary Index Multi-level

Multilevel Index –
Successive levels of indices are built
till the last level has one block
height – no. of levels
block accesses: height + 1



For the example data file:
No of block accesses required:
multi-level index: 4
single level index: 10

Index Sequential Access Method (ISAM) Files

ISAM files –
Ordered files with a multilevel primary/clustering index

Insertions:
Handled using overflow chains at data file blocks

Deletions:
Handled using deletion markers

Most suitable for files that are relatively static

If the files are dynamic, we need to go for dynamic multi-level index structures based on B^+- trees

B^+- trees

Bayer & McCreight
Acta Informatica 1972

- Balanced search trees (self-balancing)
  - Internal nodes have variable number of children
  - All leaves are at the same level
  - Nodes − internal or leaf − are disk blocks
- Leaf node entries point to the actual data records
  - All leaf nodes are linked up as a list
- Internal node entries carry only index information
  - In B-trees, internal nodes carry data record pointers also
  - The fan-out in B-trees is less
- Make sure that blocks are always at least half filled
- Support both random and sequential access of records

Order

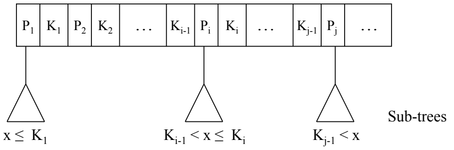
Order (m) of an Internal Node
- Order of an internal node is the maximum number of tree pointers held in it.
- Maximum of (m-1) keys can be present in an internal node

Order (m_{leaf}) of a Leaf Node
- Order of a leaf node is the maximum number of record pointers held in it. It is equal to the number of keys in a leaf node.

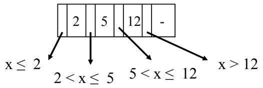
Internal Node Structure

$\left\lceil \frac{m}{2} \right\rceil \leq j \leq m$

P_i: Tree pointer (Block pointer)
K_i: Key value
m : Order(internal)



Example



Internal Nodes

An internal node of a B^+- tree of order m:

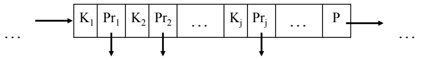
- It contains at least $\left\lceil \frac{m}{2} \right\rceil$ pointers, except when it is the root node (Root nodes − a min of 2 pointers is ok)
- It contains at most m pointers.
- If it has $P_1, P_2, \ldots, P_j$ pointers with $K_1 < K_2 < K_3 \ldots < K_{j-1}$ as keys, where $\left\lceil \frac{m}{2} \right\rceil \leq j \leq m$, then
  - P_1 points to the sub-tree with records having key value $x \leq K_1$
  - P_i $(1 < i < j)$ points to the sub-tree with records having key value x such that $K_{i-1} < x \leq K_i$
  - P_j points to records with key value $x > K_{j-1}$

Leaf Node Structure

Structure of leaf node of B^+- of order m_{leaf} :

- It contains one block pointer P to point to next leaf node
- At least $\left\lceil \frac{m_{leaf}}{2} \right\rceil$ record pointers and $\left\lceil \frac{m_{leaf}}{2} \right\rceil$ key values
- At most m_{leaf} record pointers and key values
- If a node has keys $K_1 < K_2 < \ldots < K_j$ with $Pr_1, Pr_2 \ldots Pr_j$ as record pointers and P as block pointer, then

Pr_i points to record with K_i as the search field value, $1 \leq i \leq j$
P points to next leaf block



Order Calculation

Block size: B, Size of Index field: V
Size of block pointer: P, Size of record pointer: P_r

Order of Internal node (m):
As there can be at most m block pointers and (m-1) keys
$(m*P) + ((m-1) * V) \leq B$
m can be calculated by using the above inequality (choose max)

Order of leaf node:
As there can be at most m_{leaf} record pointers and keys with one block pointer in a leaf node,
m_{leaf} can be calculated by using the inequality: (choose max)
$(m_{leaf} * (P_r + V)) + P \leq B$

Example Order Calculation

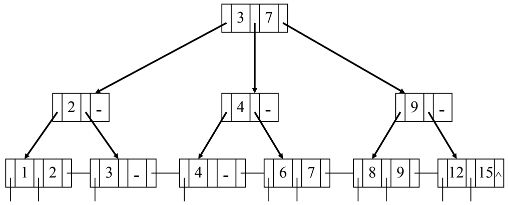
Given B = 512 bytes V = 8 bytes
P = 6 bytes P_r = 7 bytes. Then

Internal node order m = ?
$m * P + ((m-1) *V) \leq B$
$m * 6 + ((m-1) *8) \leq 512$
$14m \leq 520$
$m \leq 37$

Leaf order m_{leaf} = ?
$m_{leaf} (P_r + V) + P \leq 512$
$m_{leaf} (7 + 8) + 6 \leq 512$
$15m_{leaf} \leq 506$
$m_{leaf} \leq 33$

Example B^+- tree

m = 3 m_{leaf} = 2



Insertion into B^+- trees

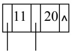
Every (key, record pointer) pair is inserted in an appropriate leaf (Search for it)

- If a leaf node overflows:
  - Node is split at $j = \left\lceil \frac{(m_{leaf} + 1)}{2} \right\rceil$
  - First j entries are kept in original node
  - Entities from j+1 are moved to new node
  - j^{th} key value K_j is *replicated* in the parent of the leaf.
- If an internal node overflows:
  - Node is split at $j = \left\lfloor \frac{(m + 1)}{2} \right\rfloor$
  - Values and pointers up to P_j are kept in the original node
  - j^{th} key value K_j is *moved* to the parent of the internal node
  - P_{j+1} and the rest of entries are moved to a new node.

Example of Insertions $m = 3$ $m_{leaf} = 2$

Insert 20, 11

1

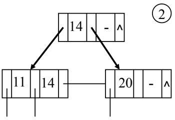


Insert 14

Overflow. leaf is split at $j = \left\lceil \frac{(m_{leaf} + 1)}{2} \right\rceil = 2$

14 is copied to parent level

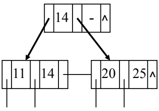
2



Insert 25

Inserted at leaf level

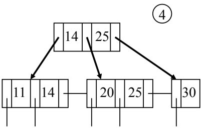
3



Insert 30

Overflow. split at 25. 25 is copied to upper level

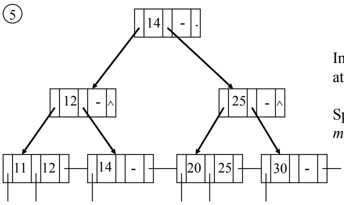
4



Insert 12 Overflow at leaf level.

- Split at leaf level,
- Triggers overflow at internal node
- Split occurs at internal node;

5

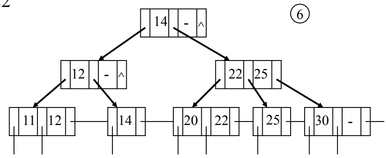


Internal node split at $j = \left\lceil \frac{m}{2} \right\rceil$

Split at 14 and 14 is *moved* up

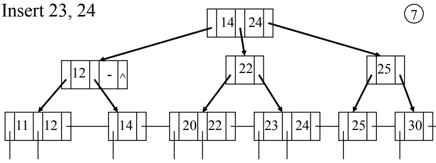
Insert 22

6



Insert 23, 24

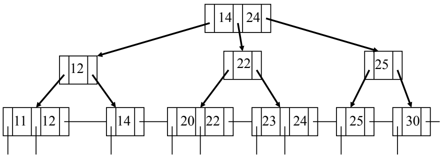
7



Deletion in B$^+$- trees

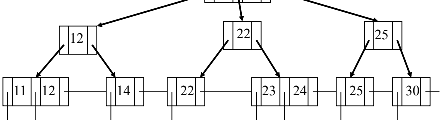
- Delete the entry from the leaf node
- Delete the entry if it is present in Internal node and replace with the entry to its right / right sibling.
- If underflow occurs after deletion
  - Distribute the entries from left sibling
    if not possible – Distribute the entries from right sibling
    if not possible – Merge the node with left and right sibling

Example

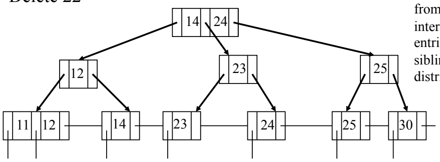


Delete 20

Removed entry
from leaf here

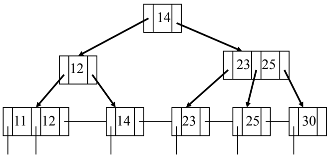


Delete 22

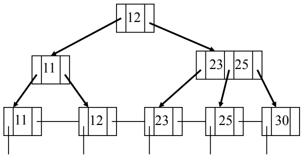


22 is removed from leaf and internal node entries from right sibling are distributed to left

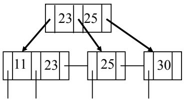
Delete 24



Delete 14



Delete 12



Level drop has occurred

Advantages of B^+- trees:

1) Any record can be fetched in equal number of disk accesses.

2) Range queries can be performed easily as leaves are linked up

3) Height of the tree is less as only keys are used for indexing

4) Supports both random and sequential access.

Disadvantages of B^+- trees:

Insert and delete operations are complicated

Root node becomes a *hotspot*

Parallel Access of Multiple Disks

Single Disk: high block access time: 6msec − 50msec

Why not use parallel access to improve performance?

RAID − Redundant Array of Independent Disks (current usage)

Redundant Array of Inexpensive Disks (early usage)

RAID techniques aim to improve performance and reliability

Two ideas are employed

1) Data Striping − distribute data on to multiple disks
   Parallel reading of disks − faster data access
2) Add redundant data to help recover from disk crashes
   Take help of error-recovery codes

Details follow …

Data Striping

Data Striping − distribute data on multiple disks

Bit-level striping: i^{th} bit of each byte − stored on the i^{th} disk

Use 8 disks for 8 bits of a byte. // higher granularity is also possible

One (parallel) block read − 8 blocks of the data file

Transfer rate − eight times that of single disk

Read/write of a block − involves use of all the disks

Block-level striping: i^{th} block of data − i^{th} disk

Using *n* disks −

Single block access: *n* simulataneous block reads can happen

Multi-block access: *n* fold increase in transfer rate (parallel reads)

Downside: reliability of the set of disks comes down

Reliability of Multiple Disks

Reliability is modeled using Mean Time To Failure (MTTF)

An example scenario:

Mean Time To Failure (MTTF) of a disk: 2,40,000hrs

That is, probability of failure of a single disk in an hour: 1/2,40,000

Probability of Failure of a single disk in a 100-disk set: 1/2,400

MTTF of the 100-disk system is 2,400hrs = 100days ~ 3.3months!

This is unacceptable..

Mirroring disks to increase reliability

Mirroring − Each disk has a mirror disk − same data on both

If a disk fails − use the mirror of that disk till the original is replaced

One can improve reliability greatly:

- A disk with MTTF = 2,40,000hrs − mirrored with same kind of disk
- Probability of a disk failure in a particular hour: 2/2,40,000
- Time to repair/copy a disk is, say, 24hrs
- Probability of disk failure while copying/repair: 24/2,40,000
- Probability of a *data loss*: (2/2,40,00) * (24/2,40,000) = 1/(12*10^8)
- Or MTTF of the combination = 12*10^8 hrs

Performance: reading: same as a single disk or better

Writing: same as single disk, both disks are updated in parallel

Reliability and performance with parity disks

Mirroring − High reliability; uses 50% more disks!

Get good reliability & also performance with fewer additional disks?

Idea: Store additional information to recover data of the failed disk

Error-correcting codes − parity bit (1 if #of 1's is odd, 0 otherwise)

Data: 1 0 1 1 0 0 1 0 - Parity Bit: 0 (#of 1's in Data & Parity is *even*)

Data: 1 0 0 1 1 0 1 1 - Parity Bit: 1 (#of 1's in Data & Parity is *even*)

Parity block: (Assuming block-level data striping with N disks)

The i^{th} bit of the parity block j: parity of the i^{th} bits of block j on all disks

Parity Disk − has parity blocks for all data blocks

If a disk k fails: Set the i^{th} bit of block j using i^{th} parity bit of block j

Do this for all blocks to recover data of disk k!

N data disks, one extra disk − good performance and reliability!

Distributed Parity

N data disks and 1 redundant (parity) disk

- Very good performance and protection against single-disk crash
- Updating *any* data block − requires updating the parity disk
- Usage of parity disk − high and it ages faster!

Can we distribute the parity information?

Use each disk as a redundant (parity) disk for some *part* of the data!

Say, we have D_0, D_1, D_2, ⋯, D_5 − 6 disks with, say, 60 cylinders each

Use each as the redundant disk for 1/6 of data:

Cyl# 0, 6, 12, ⋯ of D_0 – parity blocks for other disk cyl# 0, 6, 12, ...

Cyl# 1, 7, 13, ⋯ of D_1 – parity blocks for other disk cyl# 1, 7, 13, ...

Etc…

This is called *distributed parity* − disk usage is uniform!

Standard RAID Levels

RAID-0 − Bit-level striping; No parity data; No mirroring

RAID-1 − Mirrored disks; No parity; No data striping

RAID-2 − Bit-level striping; Redundancy using Hamming codes

Not in much use currently.

RAID-3 − Byte-level striping; dedicated parity disk

Not in common use currently.

RAID-4 − Block-level striping; dedicated parity disk

RAID-5 − Block-level striping; distributed parity

RAID-6 − Block-level striping; double distributed parity;

Up to 2 disk crashes can be tolerated

Storage Area Networks (SAN)

Specialized computing systems for providing large-scale storage

-- Dedicated hardware and software

-- Shared across several servers

-- Connected to servers through a dedicated high-speed network using special optical cables − Fiber channels

-- Block-level data storage

-- Internally use a large number of disks under a suitable RAID

-- Offer SCSI (Small Computer System Interface) interface to servers

-- Details are beyond the scope of this course