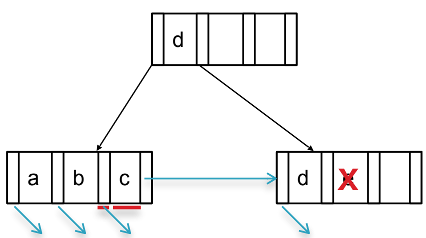
Outline

- Storage hierarchy
- Disks
- RAID
- Spark
- Buffer Manager
- File Organization
- Indexes
- B+-Tree Indexes
- Etc..

or Deletion e *(w/ redistribution)*

- try merge w/ buddy
- try to *borrow*

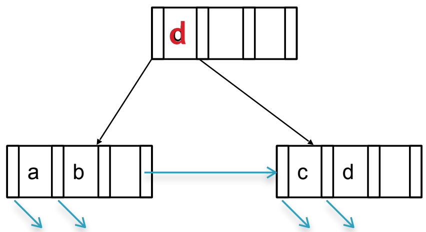


- leaf has at least $\lceil \frac{n-1}{2} \rceil$ keys
- interior node has at least $\lceil \frac{n}{2} \rceil$ pointers

else begin /* Redistribution: borrow an entry from N' */
    if (N' is a predecessor of N) **then begin**
        if (N is a nonleaf node) **then begin**
            let m be such that $N'.P_m$ is the last pointer in N'
            remove ($N'.K_{m-1}, N'.P_m$) from N'
            insert ($N'.P_m, K'$) as the first pointer and value in N,
                by shifting other pointers and values right
            replace K' in *parent*(N) by $N'.K_{m-1}$
        end
        else begin
            let m be such that ($N'.P_m, N'.K_m$) is the last pointer/value
                pair in N'
            remove ($N'.P_m, N'.K_m$) from N'
            insert ($N'.P_m, N'.K_m$) as the first pointer and value in N,
                by shifting other pointers and values right
            replace K' in *parent*(N) by $N'.K_m$
        end
    end
    else … symmetric to the **then** case …
end

or Deletion e *(w/ redistribution)*

- try merge w/ buddy
- try to *borrow*

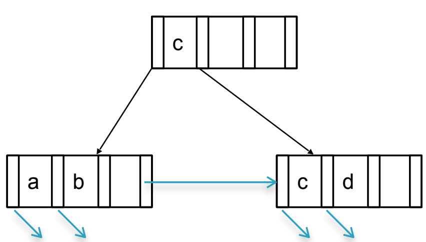


- leaf has at least $\lceil \frac{n-1}{2} \rceil$ keys
- interior node has at least $\lceil \frac{n}{2} \rceil$ pointers

else begin /* Redistribution: borrow an entry from N' */
  if (N' is a predecessor of N) **then begin**
    if (N is a nonleaf node) **then begin**
      let m be such that $N'.P_m$ is the last pointer in N'
      remove ($N'.K_{m-1}, N'.P_m$) from N'
      insert ($N'.P_m, K'$) as the first pointer and value in N,
        by shifting other pointers and values right
      replace K' in *parent*(N) by $N'.K_{m-1}$
    end
    else begin
      let m be such that ($N'.P_m, N'.K_m$) is the last pointer/value
        pair in N'
      remove ($N'.P_m, N'.K_m$) from N'
      insert ($N'.P_m, N'.K_m$) as the first pointer and value in N,
        by shifting other pointers and values right
      replace K' in *parent*(N) by $N'.K_m$
    end
  end
  else … symmetric to the **then** case …
end

or Deletion e *(w/ redistribution)*

- try merge w/ buddy
- try to *borrow*



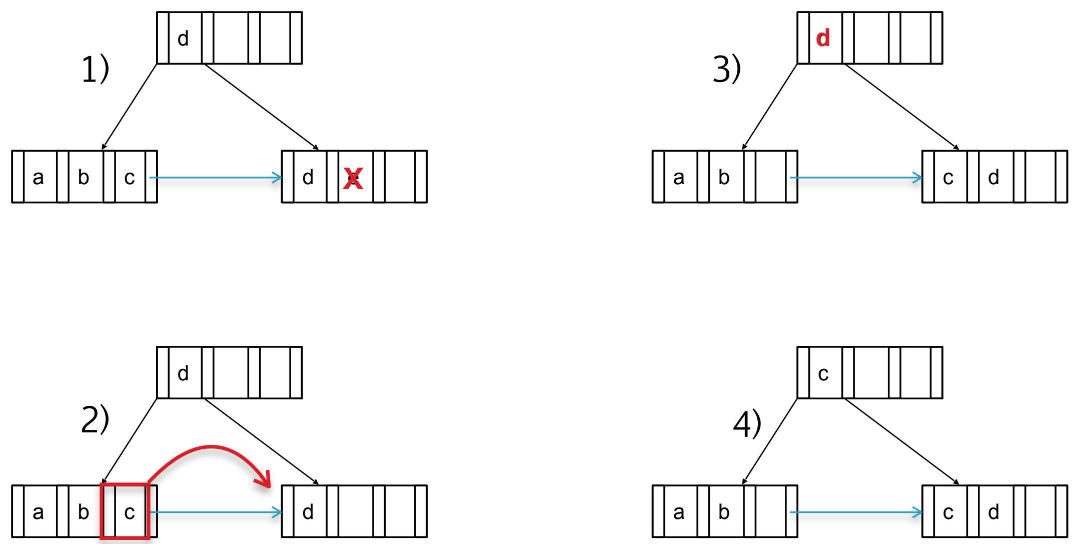
- leaf has at least $\lceil \frac{n-1}{2} \rceil$ keys
- interior node has at least $\lceil \frac{n}{2} \rceil$ pointers

else begin /* Redistribution: borrow an entry from N' */
  if (N' is a predecessor of N) **then begin**
    if (N is a nonleaf node) **then begin**
      let m be such that $N'.P_m$ is the last pointer in N'
      remove ($N'.K_{m-1}, N'.P_m$) from N'
      insert ($N'.P_m, K'$) as the first pointer and value in N,
        by shifting other pointers and values right
      replace K' in *parent*(N) by $N'.K_{m-1}$
    end
    else begin
      let m be such that ($N'.P_m, N'.K_m$) is the last pointer/value
        pair in N'
      remove ($N'.P_m, N'.K_m$) from N'
      insert ($N'.P_m, N'.K_m$) as the first pointer and value in N,
        by shifting other pointers and values right
      replace K' in *parent*(N) by $N'.K_m$
    end
  end
  else … symmetric to the **then** case …
end

interior node redistributions more full-on rotation…

Deletion e *(redis summary)*

- leaf has at least $\lceil \frac{n-1}{2} \rceil$ keys
- interior node has at least $\lceil \frac{n}{2} \rceil$ pointers
- try merge w/ buddy
- try to *borrow*



Want to delete e…

B+ Trees in Practice

- Typical order: n = 200. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities if we assume fanout 100:
  - Height 3: 100^3 =
    - 1,000,000 leafs
    - 99,000,000 entries (ptrs in leaves)
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes. (root)
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Observations about B+-trees (minimum)

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B+-tree form a hierarchy of sparse indices.
- The B+-tree contains a relatively small number of levels
  - Level below root has at least $2 * \left\lceil \frac{n}{2} \right\rceil$ ptrs height=1
  - Next level has at least $2 * \left\lceil \frac{n}{2} \right\rceil * \left\lceil \frac{n}{2} \right\rceil$ ptrs height=2
  - Height *i* tree has at least $2 * \left\lceil \frac{n}{2} \right\rceil^{i}$ ptrs
  - If there are K search-key values in the file, the tree height (dist from root to leaf) is: $h = \lceil \log_n(K) \rceil$
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time.

B+ Trees: Summary

- Searching:
  - $\log_n(e)$ – Where *n* is the order, and *e* is the number of entries
- Insertion:
  - Find the leaf to insert into
  - If full, split the node, and adjust index accordingly
  - Similar cost as searching
- Deletion
  - Find the leaf node
  - Delete
  - May not remain half-full; must adjust the index accordingly

Minimum Tree Height

- $n_R = 10,000$, $b_R = 1000$, primary (clustered), index on non-candidate key A
- $n_S = 1,000$, $b_S = 500$, secondary (non-clustered) index on non-candidate key A.
- $t_T = 0.1$ msec, $t_S = 4$ msec
- Fanout, *N*, on both is 50.

why?	cost of finding the first leaf	cost of retrieving the tuples
primary index, candidate key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S)$
primary index, not a key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S) + (b - 1) * t_T$ *Note: primary == sorted* *b = number of pages that contain the matches*
secondary index, candidate key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S)$
secondary index, not a key, equality	$h_i * (t_T + t_S)$	$n * (t_T + t_S)$ *n = number of records that match* This can be bad

- Assume R has order $n = 50$, 10,000 distinct values for search key. What is the height of *R's tree*?
  - root has n ptrs
  - h_1 has n leaves, each of which have n pointers = n^2=2500
  - h_2 means n^3 = 125000 ptrs, so:

    $h_R = 2$

- More generally, tree of fanout n and height h has *capacity* of n^{h+1}
  - …except each leaf devotes one ptr to point to next leaf!
  - so really: $n^{h+1} - n^h$

Problems w/ b-trees

- $n_R = 10,000$, $b_R = 1000$, primary (clustered), index on non-candidate key A
- $n_S = 1,000$, $b_S = 500$, secondary (non-clustered) index on non-candidate key A.
- $t_T = 0.1$ msec, $t_S = 4$ msec
- Order n on both is 50.

why?	cost of finding the first leaf	cost of retrieving the tuples
primary index, candidate key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S)$
primary index, not a key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S) + (b - 1) * t_T$ *Note: primary == sorted* *b = number of pages that contain the matches*
secondary index, candidate key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S)$
secondary index, not a key, equality	$h_i * (t_T + t_S)$	$n * (t_T + t_S)$ *n = number of records that match* This can be bad

- max capacity of tree height h is $(n^h) * (n - 1) = n^{h+1} - n^h$ leaf ptrs:
  - $h=1$: $(50^2) - (50^1) = 2450$ leaf ptrs
  - $h=2$: $(50^3) - (50^2) = 122,500$ leaf ptrs
  - $h=3$: $(50^4) - (50^3) = 6,125,000$ leaf ptrs
  - $h_R = 2$, $h_S = 1$

- Cost to return first tuple for *R.A = 42*?
  - primary, not a candidate key
  - must traverse tree, reading in each block except root, and one table block
  - $(h_R + 1) * (t_T + t_S)$ = 3*(0.1 + 4.0) = 12.3 msec

- Cost to return the rest, assuming *blocking factor* is 10, and 100 total matches?
  - $(b-1) * t_T$ = 9 * 0.1 = 0.9 msecs

blocking factor = #tuples / block

Problems w/ b-trees

why?	cost of finding the first leaf	cost of retrieving the tuples
primary index, candidate key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S)$
primary index, not a key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S) + (b - 1) * t_T$ *Note: primary == sorted* *b = number of pages that contain the matches*
secondary index, candidate key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S)$
secondary index, not a key, equality	$h_i * (t_T + t_S)$	$n * (t_T + t_S)$ *n = number of records that match* This can be bad

- $n_R = 10,000$, $b_R = 1000$, primary (clustered), index on non-candidate key A
- $n_S = 1,000$, $b_S = 500$, secondary (non-clustered) index on non-candidate key A.
- $t_T = 0.1$ msec, $t_S = 4$ msec
- Order *n* on both is 50.

- Cost to return first tuple for *S.A = 42*?
  - secondary, not a candidate key
  - $(h_S + 1) * (t_T + t_S) = 2*(0.1 + 4.0) = 8.2$ msec

- Cost to return the rest, assuming *blocking factor* is 10, and 100 total matches?
  - *(#numMatches - 1) * t_T* = 99 * (0.1 + 4.0) = 396 + 9.9 = 405.9 msecs

Blocking factor is irrelevant because matches are randomly scattered.

blocking factor = #tuples / block

Query Processing

- Overview
- Selection operation
- Join operators
- Sorting
- Other operators
- Putting it all together…

Query Processing

- Overview
- Selection operation
- Join operators
- Sorting
- Other operators
- Putting it all together…

Join

- *select * from R, S where R.a = S.a*
  - *"equi-join"*
- *select * from R, S where |R.a – S.a | < 0.5*
  - *not an equi-join*

- Option 1: Nested-loops

  for each tuple r in R

  for each tuple s in S

  check if r.a = s.a (or whether |r.a – s.a| < 0.5)
- Can be used for any join condition
  - As opposed to some algorithms we will see later
- R called *outer relation*
- S called *inner relation*

Nested-loops Join

not using indexes

- Cost ? Depends on the actual values of parameters, especially memory
- b_r , b_s → *Number blocks of R and S*
- n_r , n_s → *Number tuples of R and S*
- Case 1: Minimum memory required = 3 blocks
  - One to hold the current *R* block, one for current S block, one for the result being produced
  - Blocks transferred:
    - Must scan *R* tuples once: b_r blocks
    - For each *R* tuple, must scan *S*: $n_r * b_s$
    - $b_r + n_r * b_s$
  - Seeks ?
    - $n_r + b_r$

Nested-loops Join

- Case 1: Minimum memory required = 3 blocks
  - Blocks transferred: $n_r * b_s + b_r$
  - Seeks: $n_r + b_r$
- Example:
  - Number of records -- *R*: $n_r = 10{,}000$, *S*: $n_s = 5000$
  - Number of blocks -- *R*: $b_r = 400$, *S*: $b_s = 100$
- *R* as outer relation:
  - blocks transferred: $n_r * b_s + b_r$ = 10000 * 100 + 400 = 1,000,400
  - seeks: 10400
  - time: 1000400 t_T + 10400 t_S = 1000400(.1ms) + 10400(4ms) = 141.64 sec
- *S* outer relation:
  - 5000 * 400 + 100 = 2,000,100 block transfers,
  - 5100 seeks
  - = 2000100 t_T + 5100 t_S = 220.41 sec

Order matters!

Nested-loops Join

- Case 2: S fits in memory
  - Blocks transferred: $b_s + b_r$
  - Seeks: 2
- Example:
  - Number of records -- R: $n_r = 10{,}000$, S: $n_s = 5000$
  - Number of blocks -- R: $b_r = 400$, S: $b_s = 100$
- Then:
  - blocks transferred: 400 + 100 = 500
  - seeks: 2
  - $= 500t_T + 2t_S = 0.058$ sec

Orders of magnitude difference

Block Nested-loops Join

$n_r = 10{,}000$, S: $n_s = 5000$
$b_r = 400$, S: $b_s = 100$

- Simple modification to "nested-loops join"

  for each block B_r in R
  for each block B_s in S
  for each tuple r in B_r
  for each tuple s in B_s
  check if $r.a = s.a$ (or whether $|r.a – s.a| < 0.5$)

- Case 1: Minimum memory required = 3 blocks
  - Blocks transferred: $b_r * b_s + b_r$
  - Seeks: $2 * b_r$
- For the example:
  - blocks: 400*100 + 400 = 40,400 msec = 40.4 sec
  - seeks: 800*4 = 3200 msec = 3.2 sec
  - 43.6 seconds

Block Nested-loops Join

n_r = 10,000, S: n_s = 5000
b_r = 400 , S: b_s = 100

- Case 1: Minimum memory required = 3 blocks
    - Blocks transferred: $b_r * b_s + b_r$
    - Seeks: $2 * b_r$
- Case 2: S fits in memory
    - Blocks transferred: $b_s + b_r$
    - Seeks: 2
- What about in between ?
    - Say there are 50 blocks, but S is 100 blocks
    - Why not use all the memory that we can…

Block Nested-loops Join

n_r = 10,000, S: n_s = 5000
b_r = 400 , S: b_s = 100

- Case 3: 50 blocks (S = 100 blocks)

for each group of 48 blocks in R
for each block B_s in S
for each tuple r in the group of 48 blocks
for each tuple s in B_s
check if r.a = s.a (or whether |r.a – s.a| < 0.5)

- 48 blocks for R
- 1 block for S
- 1 block for output

- Why is this good ?
    - We only have to read S a total of ceiling(b_r / 48) times (instead of b_r times)
    - Blocks transferred:
        - $\lceil \frac{b_r}{48} \rceil * b_s + b_r = \lceil \frac{400}{48} \rceil * 100 + 400 = 1300$
    - Seeks:
        - $2 * \lceil \frac{b_r}{48} \rceil = 18$
    - 1300 * 0.1 + 18 * 4 = 130 msec + 72 msec = 0.202 seconds
- Use S as the outer relation:
    - Blocks transferred:
        - $\lceil \frac{b_s}{48} \rceil * b_r + b_s = \lceil \frac{100}{48} \rceil * 400 + 100 = 1300$
    - Seeks:
        - $2 * \lceil \frac{b_s}{48} \rceil = 6$
    - 1300 * 0.1 + 6 * 4 = 130 msec + 24 msec = 0.154 seconds

So far…

- Block Nested-loops join
  - Can always be applied irrespective of the join condition
  - If the smaller relation fits in memory, then cost:
    - $b_r + b_s$
    - This is the best we can hope if we have to read the relations once each
  - CPU cost of the inner loop is high…

Index Nested-loops Join

- "select * from R, S where R.a = S.a"
  - equi-join
- Nested-loops

  for each tuple r in R

    for each tuple s in S

      check if r.a = s.a (or whether |r.a – s.a| < 0.5)

- If index on S.a, why not use the index instead of the inner loop ?

  for each tuple r in R

    use the index to find S tuples with S.a = r.a

Index Nested-loops Join

- *select * from R, S where R.a = S.a*
  - Called an "*equi-join*"
- *Why not use the index instead of the inner loop ?*

  for each tuple r in R

  use the index to find S tuples with S.a = r.a
- Cost of the join:
  - $b_r (t_T + t_S) + n_r * c$
  - *c == the cost of index access*
    - *Computed using the formulas discussed earlier*