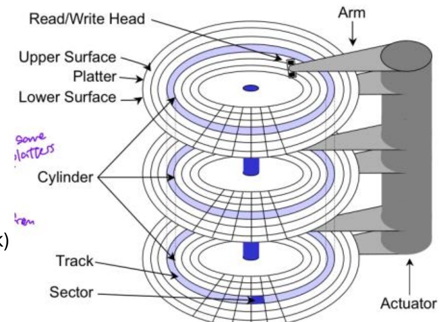
DBMS stores data on non-volatile disk for persistence
DBMS processes data in main memory (RAM)
read: transfer data from disk to RAM
write: transfer data from RAM to disk

**Disk access time:**

- seek: moving arms to position disk head on track
- rotational delay: waiting for block to rotate under head =time for one rev / 2
- transfer: moving data to/from disk surface =num sectors * (time for one rev / num sectors per track)
- access = **seek + rotational delay + transfer**




If 3 platters 2 surface, 6 tracks in 1 cylinder

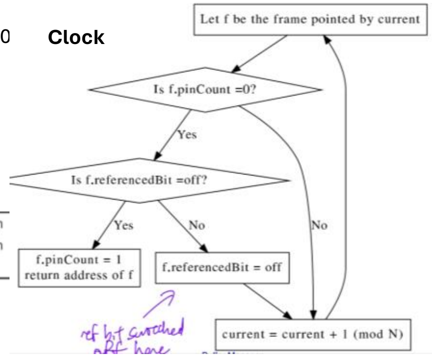
**Buffer Manager**

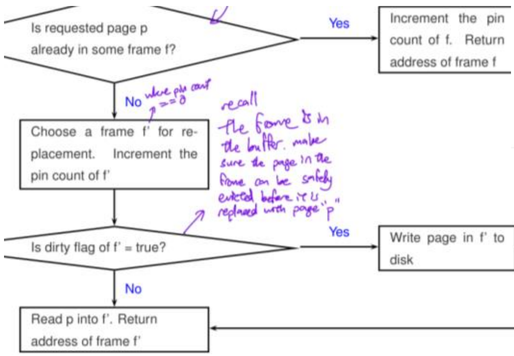
Buffer pool = main memory, partitioned into frames
Dirty = modified and not updated on disk
pin count - number of clients using page (initialized to 0
dirty flag – initialized False, update when unpinning
Replace pages only when pin = 0
Must write to disk if dirty=true before replacing

**Clock**

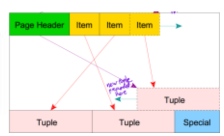
refbit: turns on when pin becomes 0
Evict pages with refbit off and pin=0

**Clock**




**Slotted Page (un)/packed**

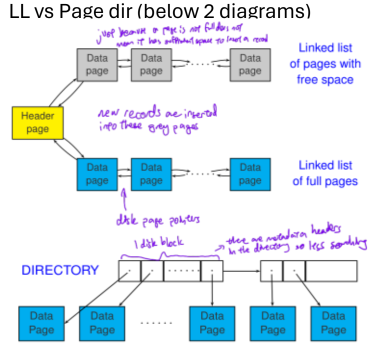



**Storing records**

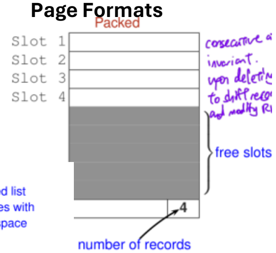
Heap file: unordered file
Sorted file ordered on search key
Hashed file: hash function

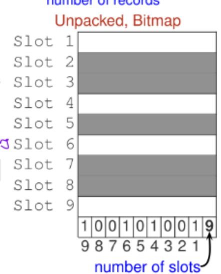
**Heap file Implementation**

LL vs Page dir (below 2 diagrams)




**Page Formats**




**B+ Tree**

Order of index d
Root node has [1,2d] entries
(up to 2d+1 pointers)
Nonroot has [d,2d] entries
(up to 2d+1 pointers)

**Format**

Format 1: k* is an actual record (clustered)
Format 2: k* is of the form (k, rid)
Format 3: k* is of the form (k, rid-lis

**Split Overflow**

- Split overflowed leaf node by distributing d+1 entries to new leaf node (i.e. right node should have 1 more records)
- Create a new index entry using the smallest key in new leaf node
- Insert new index entry into parent node of overflowed node

**Split Insert leaf**

First d into N, last d+1 into node to right
(push middle+ entry to new node)
Then insert internal

**Insert Internal (pull middle up)**

If no parent, create parent.
Pull middle up, middle+ entry goes right

**Redistribute leaf (overflow)**

If non-full adj right, first 2d goes in, last entry goes right
Elif non-full adj left, last 2d goes in, first entry goes left
Else split

**Merge Delete w dist (pull parent down)**

Move first entry from right
Elif Move last entry from left
Else move entries to right
Else move to left

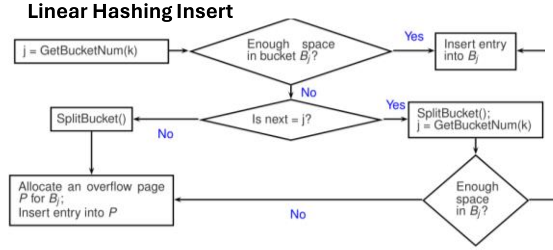
**Merge underflow**

Merge right
Elif merge left
Else

**Bulk Load**

1. Sort the data entries to be inserted by search key
2. Load the leaf pages of B+-tree with sorted entries
3. Initialize B+-tree with an empty root page
4. For each leaf page (in sequential order), insert its index entry into the rightmost parent-of-leaf level page of B+-tree

**Linear Hashing Insert**




- **GetBucketNum(k)** returns bucket # where entry with search key $k$ is located

$$GetBucketNum(k) = \begin{cases} h_{level}(k) & \text{if } h_{level}(k) \geq next \\ h_{level+1}(k) & \text{otherwise.} \end{cases}$$

if bucket not split, else use this; the diff is $h_{level+1}(k)$ uses 1 more bit

- **SplitBucket()** splits bucket $B_{next}$
  1. Redistribute the entries in $B_{next}$ into $B_{next+N_{level}}$ using $h_{level+1}()$
  2. next = next + 1
  3. if ($next = N_{level}$) then { level = level + 1; next = 0 } → you have split every bucket

**Linear Hashing Deletion**

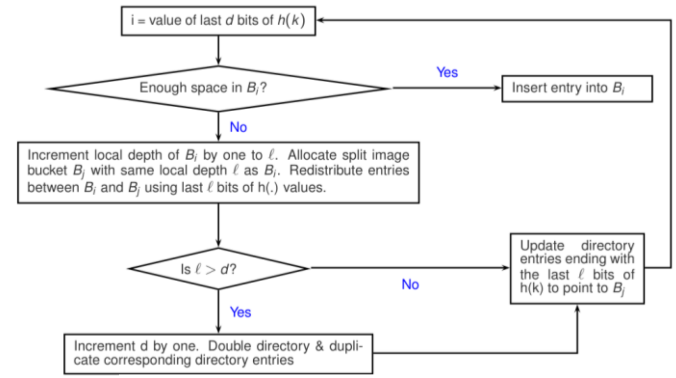
If last bucket is $B_{nlevel+next-1}$ is empty
If next>0, next-=1
If next=0 and level>0, next goes to last bucket of prev level then next-=1

**Extensible Hashing Insert**

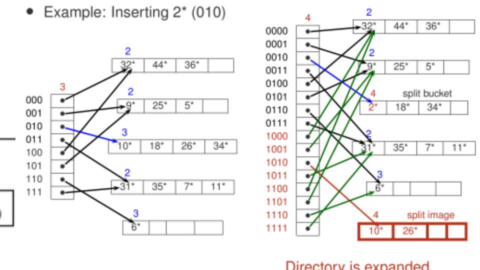
**Overflow**

Each new directory entry (except for the entry for the split image) points to the same bucket as its corresponding entry
Number of directory entries pointing to a bucket = $2^{d-\ell}$
If split depth=global depth, double directory
Else, split and increment local

Directory has $2^d$ entries
d is called global depth of the hashed file
Corresponding entries if their addresses differ only in the dth bit
Each bucket maintains a local depth (denoted by $\ell \in [0,d]$)
All entries in a bucket with local depth $\ell$ have the same last $\ell$ bits in h(.)

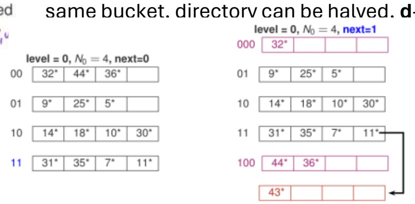



- Example: Inserting 2* (010)




**Extensible Hashing Deletion**

If same local depth and differ only in ℓth bit, merge
If each pair of corresponding entries point to the same bucket, directory can be halved. **d**-=1




- Bucket $B_{11}$ overflows
- Split bucket $B_{00}$ (32 = 100000, 36 = 100100, 44 = 101100)
- Increment next to 1
- Insert 43* into overflow page

## External Merge Sort

E.g. Sort 11-page using B=3
1. Read first 3 into B
2. Sort in main memory
3. Write sorted run to disk
4. Repeat 2 and 3
5. Merge each run, using B-1 for input, 1 for output

**File size of N pages**

Pass 0: Creation of sorted runs
- Read in and sort B pages at a time
- Number of sorted runs created = ⌈N/B⌉
- Size of each sorted run = B pages

Pass i, i ≥ 1: Merging of sorted runs
- Use B − 1 buffer pages for input & one buffer page for output
- Performs (B-1)-way merge

Analysis:
- $N_0$ = initial sorted runs = ⌈N/B⌉

+1 comes from initial pass
- Total number of passes = $\lceil \log_{B-1}(N_0)\rceil + 1$
- Total number of I/O = $2N(\lceil \log_{B-1}(N_0)\rceil + 1)$

⋆ Each pass reads N pages & writes N pages

**Blocked I/O (optimize merge passes)**

Read and write in units of buffer blocks of b

Given an allocation of B buffer pages for sorting,

Allocate one block (b pages) for output
- Remainder space can accommodate ⌊(B−b)/b⌋ blocks for input
- Thus, can merge at most ⌊(B−b)/b⌋ sorted runs in each merge pass

Analysis:

N = number of pages in file to be sorted

B = number of available buffer pages

b = number of pages of each buffer block

$N_0$ = number of initial sorted runs = ⌈N/B⌉

F = number of runs that can be merged at each merge pass = ⌊B/b⌋−1

Num passes = $\lceil \log_F(N_0)\rceil + 1$

If b=1, then same as regular external merge

If b is good, then $\lceil \log_F(N_0)\rceil > 2N(\lceil \log_{B-1}(N_0)\rceil$

With projection, size of record divide by size of attributes projected out

Page-nested needs >= 3 buffers

For block nested, check the formula

When joining, size of record *2 cos R+S size

Always need 1 buffer for holding input, output, and smaller table

**Covering Index**

An index I is a covering index for a query Q if all the attributes referenced in Q are part of the key or include column(s) of I

i.e. index includes all the attributes in Q.

esp if index is format1
- No RID lookup needed
- index-only plan

**Conjunctive Normal Form (CNF) predicate**

$\underbrace{(\text{rating} \geq 8}_{\text{term/conjunct}} \vee \underbrace{\text{director} = \text{“Coen”})}_{\text{term/conjunct}}$ (disjunctive conjunct) $\wedge \underbrace{(\text{year} > 2003)}_{\text{term/conjunct}} \wedge \underbrace{(\text{language} = \text{“English”})}_{\text{term/conjunct}}$

B+ Index matches non-disjunctive CNF predicate if at most one non-equality op (last attribute). Q: <,=,=, then matches is at most <

Hash index matches p if all are equality

**Primary Conjuncts**

The conjuncts in p that I matches

**Covered Conjunct**

The conjuncts in p that I matches and appear in key or include columns of index

**Notation**

r   relational algebra expression

||r||   number of tuples in output of r

|r|   number of pages in output of r

$b_d$   num records on a page

$b_i$   num entries on a page (leaf node)

F   ave fanout of B+-tree index (i.e., number of pointers to child nodes)

h   height of B+-tree inde: $h = \lceil \log_F(\lceil \frac{||R||}{b_i} \rceil)\rceil$ levels of internal nodes).

if format-2 index on table R

**Cost of B+ tree index σ Selection**

C

$$Cost_{internal} = \begin{cases} \lceil \log_F(\lceil \frac{||R||}{b_d} \rceil)\rceil & \text{if } I \text{ is a format-1 index,} \\ \lceil \log_F(\lceil \frac{||R||}{b_i} \rceil)\rceil & \text{otherwise.} \end{cases}$$

← if leaf stores records

← if leaf stores entries

← how many records fit / how many records on 1 page

$$Cost_{leaf} = \begin{cases} \lceil \frac{||\sigma_{p'}(R)||}{b_d} \rceil & \text{if } I \text{ is a format-1 index,} \\ \lceil \frac{||\sigma_{p'}(R)||}{b_i} \rceil & \text{otherwise.} \end{cases}$$

← how many records / how many entries on 1 page

**Hash-based** (assume $B > \sqrt{f \times |\pi_L^*(R)|}$; i.e., no partition overflow)

- Cost = $\underbrace{|R| + |\pi_L^*(R)|}_{\text{partitioning phase}} + \underbrace{|\pi_L^*(R)|}_{\text{duplicate elimination phase}}$

$f > 1$

$B > \sqrt{f \times |\pi_L^*(R)|} > \sqrt{|\pi_L^*(R)|}$

**Sort-based**

- Output is sorted
- Good if there are many duplicates or if distribution of hashed values is non-uniform
- If $B > \sqrt{|\pi_L^*(R)|}$, ← This is derived
  - ⋆ Number of initial sorted runs $N_0 = \lceil \frac{|R|}{B} \rceil \approx \sqrt{|\pi_L^*(R)|}$
  - ⋆ Number of merging passes = $log_{B-1}(N_0) \approx 1$
  - ⋆ Sort-based approach requires 2 passes for sorting
  - ⋆ Cost = $\underbrace{|R| + |\pi_L^*(R)|}_{\text{pass 0}} + \underbrace{|\pi_L^*(R)|}_{\text{pass 1}}$
  - ⋆ Both hash-based & sort-based methods have same I/O cost

**Hash Based Projection $\pi_L$**

Consider $\pi_L(R)$

1 initialize an empty hash table T

2 for each tuple t in R do

3   apply hash function h on πL(t)

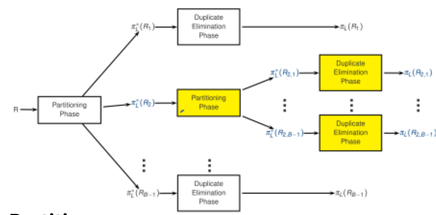
4   let t be hashed to bucket Bi in T

5   if ($\pi_L$(t) is not in Bi) then

6     insert $\pi_L$(t) into Bi

7   output all entries in T

• Cost = |R| if T fits in main memory




**Partition**

1 buffer input, (B − 1) buffers for output
- Read R one page at a time into input buffer
- For each tuple t in input buffer,
- project out unwanted attributes
- apply a hash function h on t' to distribute t' into one output buffer
- Flush to disk

**Duplicate Elimination**

For each partition $R_i$,
- Initialize an in-memory hash table
- Read $\pi^*_L(R_i)$ one page at a time; for each tuple t read,
  - Hash t into bucket Bj with hash function h' (h'≠ h)
  - Insert t into Bj if t∉ Bj
- Output tuples in hash table

Approach is effective if $B$ is large relative to $|R|$

How large should $B$ be?
- Assume that $h$ distributes tuples in $R$ uniformly
- Each $R_i$ has $\frac{|\pi_L^*(R)|}{B-1}$ pages
- Size of hash table for each $R_i$ = $\frac{|\pi_L^*(R)|}{B-1} \times f$
  - ⋆ $f$ = fudge factor >1
- Therefore, to avoid partition overflow, $B > \frac{|\pi_L^*(R)|}{B-1} \times f$
  - ⋆ Approximately, $B > \sqrt{f \times |\pi_L^*(R)|}$

Analysis: Assume there's no partition overflow
- Cost of partitioning phase: $|R| + |\pi_L^*(R)|$
- Cost of duplicate elimination phase: $|\pi_L^*(R)|$
- Total cost = $|R| + 2|\pi_L^*(R)|$

**No optimization sort**

Cost Analysis   will omit cost of writing output which is consistent
- **Step 1**:
  - Cost to scan records = $|R|$
  - Cost to output temporary result = $|\pi_L^*(R)|$
- **Step 2**:
  - Cost to sort records = $2|\pi_L^*(R)|\ (\log_m(N_0) + 1)$ → external merge sort
  - $N_0$ = number of initial sorted runs,   $m$ = merge factor
- **Step 3**:
  - Cost to scan records = $|\pi_L^*(R)|$ → sequential scan

**Tuple Nested Loop Join ( |R| + ||R|| * |S| )**

for each tuple r ∈ R do
  for each tuple s ∈ S do
    if (r matches s) then
      output (r,s) to result

**Page Nested Loop Join ( |R| + |R| * |S| )**

for each page PR of R do
  for each page PS of S do
    for each tuple r ∈ PR do
      for each tuple s ∈ PS do
        if (r matches s) then
          output (r,s) to result

**Block Nested Loop Join**

|R| < |S|. Allocate one page for S, one page for output, remaining pages for R

while (scan of R is not done) do
  read next (B − 2) pages of R into buffer
  for each page $P_S$ of S do
    read $P_S$ into buffer
      for each r∈R in buffer, each s ∈ $P_S$ do
          if (r matches s) then output (r,s)

$$\text{Cost} = |R| + \left(\left\lceil \frac{|R|}{B-2} \right\rceil \times |S|\right)$$

**Index Nested Loop Join**

Idea:

for each tuple $r \in R$ do
  use $r$ to probe S's index to find matching tuples

Analysis:   $||S|| \div ||\text{projectionB}(S)||$
- Let $R.A_i = S.B_j$ be the join condition
- Uniform distribution assumption:
  each $R$-tuple joins with $\lceil \frac{||S||}{||\pi_{B_j}(S)||} \rceil$ number of $S$-tuples
- For a format-1 B$^+$-tree index on $S$,
  - ⋆ I/O Cost = $\underbrace{|R|}_{\text{scan R}} + \underbrace{||R|| \times J}_{\text{join each R-tuple with S}}$
  - ⋆ $J = \underbrace{\log_F(\lceil \frac{||S||}{b_d} \rceil)}_{\text{search index's internal nodes}} + \underbrace{\lceil \frac{||S||}{b_d\ ||\pi_{B_j}(S)||} \rceil}_{\text{search index's leaf nodes}}$

Consider the join with $S$ as the outer relation. Since attribute $b$ is the primary key of $S$, we assume that each $S$-tuple joins with $\frac{10000}{2000} = 5$ $R$-tuples. Since each data page of relation $R$ can contain 10 tuples, we assume that all the 5 data entries that match each $S$-tuple fit in one leaf page of $R$'s index. Since the index on $R$ is unclustered, we assume that the I/O cost of retrieving the 5 matching data records is 5 disk I/Os. Therefore, the I/O cost of Index Nested Loop Join with $S$ as outer relation is $|S| + ||S||(2+1+5) = 200 + 2000 \times (2+1+5) = 16,200$.

lecture uses R

Consider the join with $R$ as the outer relation. Since attribute $b$ is the primary key of $S$, each $R$-tuple joins with at most one $S$-tuple. Therefore, the I/O cost of Index Nested Loop Join with $R$ as outer relation is $|R| + ||R||(1+1+1) = 1000 + 10000 \times (1+1+1) = 31,000$.

Thus, Block Nested Loop Join, which requires 4,200 I/Os (from Question 2), is still the cheaper alternative.

$200 + 1000 \times \frac{200}{3}$

1. With 5 buffer pages, the cost of the Index Nested Loop Join remains the same, but the cost of the Block Nested Loop join will increase. The new cost of the Block Nested Loop join is $|S| + |R| \times \lceil \frac{|S|}{B-2} \rceil = 67,200$. And now the cheapest solution is the Index Nested Loop Join with S as the outer relation.
2. If S contains 10 tuples then we'll need to change some of our previous assumptions. Now all of the S tuples fit on a single page, and each tuple in S will match $\frac{10,000}{10} = 1,000$ tuples in R. We have

   $1 + 1000 \times \lceil \frac{1}{30} \rceil = 1001$
   - Block Nested Loop: Total I/O cost $= |S| + |R| \times \lceil \frac{|S|}{B-2} \rceil = 1,001$.
   - Index Nested Loop with S as the outer relation: Total I/O cost $= 1 + 10 \times (2 + \lceil \frac{1000}{b_i} \rceil + 1,000)$, where $b_i$ is the number of leaf node entries per leaf node in the B$^+$-tree on $R$. Thus, the total I/O cost is at least 10,031 (with $\lceil \frac{1000}{b_i} \rceil = 1$).
   - Index Nested Loop with R as the outer relation: Due to the large value of $||R||$, the total I/O cost will be higher than that of Index Nested Loop with $S$ as the outer relation.

(i) For 319-way merges, only 2 merging passes are needed. The first pass will produce $\lceil 31250/319 \rceil = 98$ sorted runs; these can then be merged in the next pass. Every page is read and written individually, at a cost of 16ms per read or write, in each of these two passes. The cost of these merging passes is therefore $2 \times 2 \times 16 \times 10000000 = 640,000,000 ms$. (The formula can be read as number of passes times cost of read and write per page times number of pages in file.)

passes   r/w   per r/w   pages

(ii) With 256-way merges, only 2 merging passes are needed. Every page in the file is read and written in each pass, but the effect of blocking is different on reads and writes. For reading, each page is read individually at a cost of 16ms. Thus, the cost of reads (over both passes) is $2 \times 16 \times 10000000 = 320000000 ms$. For writing, pages are written out in blocks of 64 pages. The I/O cost per block is $10 + 5 + 1 \times 64 = 79 ms$. The number of blocks written out per pass is $10000000/64 = 156250$, and the cost per pass is $156250 \times 79 = 12343750 ms$.

seek   rot   pages

1. num merging passes
2. read per block × num blocks × passes)
3. write per block × num blocks × passes)
4. sum 2,3

If block size = 1 page it's just page read