

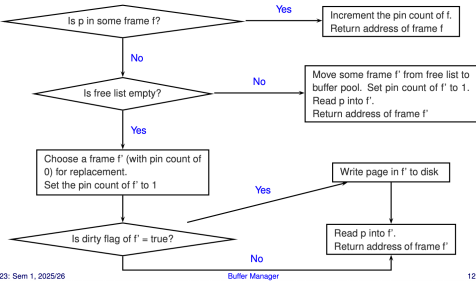
Storage

- Parts of disk
 - Platter has 2 surfaces
 - Surface has many tracks
 - Each track is broken up into sectors
 - Cylinder is the same tracks across all surfaces
 - Block comprises of multiple sectors

- **Disk Access Time:** Seek time + Rotational Delay + Transfer Time
 - **Seek Time:** Move arms to position disk head
 - **Rotational Delay:** $\frac{1}{2} \frac{60}{\text{RPM}}$
 - **Transfer time**(for n sectors): $n \times \frac{\text{time for 1 revolution}}{\text{sectors per track}}$
 - n is requested sectors on track

- Access Order
 1. Contiguous Blocks within same track (same surface)
 2. Cylinder track within same cylinder
 3. next cylinder

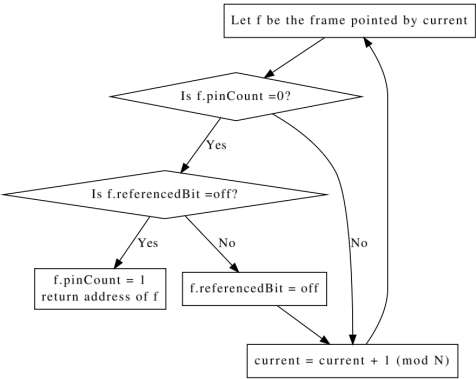
Buffer Manager



- Data stored in block sized pages called frames
- Each frame maintains pin count(PC) and dirty flag

Replacement Policies

- Decide which un-pinned page to replace
- **LRU:** queue of pointers to frames with PC = 0
- **clock:** LRU variant
 - **Reference bit:** turns on when PC = 0
 - Replace a page when ref bit off and PC = 0



Files

- Heap File Implementation
 - Linked List
 - 2 linked lists, 1 of free pages, 1 of data pages
 - Page Directory Implementation
 - Directory structure, 1 entry per page.
 - to insert, scan directory to find page with space to store record

Page Formats

- **RID** = (page id, slot number)
- Fixed Length records
 - Packed Organization: Store records in contiguous slots (requires swapping last item to deleted location during deletion)
 - Unpacked organization: Use bit array to maintain free slots
- **Variable Length Records:** Slotted page organization

Record Formats

- Fixed Length Records: Stored consecutively
- Variable length Records
 - Delimit fields with special symbols (F1, \$, F2 \$, F3)
 - Array of field offsets ($o_1, o_2, o_3, F1, F2, F3$)

Data Entry Formats

1. $k *$ is an actual data record (with search key value k)
2. $k *$ is of the form **(k , rid)**
3. $k *$ is of the form **(k , rid-list)** list of rids of data with key k

B+ Tree index

- **Search key** is sequence of k data attributes $k \geq 1$
- **Composite search key** if $k > 1$
- **unique key** if search key contains *candidate* key of table
- index is stored as file
- **Clustered index:** Ordering of data is same as data entries
 - key is known as **clustering key**
 - Format 1 index is clustered index (Assume format 2 and 3 to be unclustered)

Tree based Index

- **root node** at level 0
- Height of tree = no of levels of internal node
- **Leaf nodes**
 - level h , where h is height of tree
- **internal nodes** store entries in form $(p_0, k_1, p_1, k_2, p_2, \dots, p_n)$
 - $k_1 < k_2 < \dots < k_n$
 - p_i = disk page address
- **Order** of index tree
 - Each non-root node has $m \in [d, 2d]$ entries
 - Root node has $m \in [1, 2d]$ entries

- **Equality search:** At each *internal* node N , find largest key k_i in N , such that $k_i \leq k$
 - if k_i exists, go subtree p_i , else p_0
- **Range search:** First matching record, and traverse doubly linked list
- **Min nodes at level** i is $2 \times (d + 1)^{i-1}, i \geq 1$
- **Max nodes at level** i is $(2d + 1)^i$

Operations (Right sibling first, then left)

Insertion

1. **Leaf node Overflow**
 - Redistribute and then split
 - **Split:** Create a new leaf N with $d + 1$ entries. Create a new index entry $(k, \blacksquare)$ where k is smallest key in N
 - **Redistribute:** If sibling is not full, take from it. If given right, update right's parent pointer, else current node's parent pointer
2. **Internal node Overflow**
 - Node has $2d + 1$ keys.
 - Push middle $(d + 1)$ -th key up to parent.

Deletion

1. **Leaf node**
 - Redistribute then merge
 - **Redistribution**
 - Sibling must have $> d$ recordsto borrow
 - Update parent pointers to right sibling's smallest key)
 - **Merge**
 - If sibling has d entries, then merge
 - Combine with sibling, and then remove parent node
2. **Internal Node Underflow**
 - Let N' be adjacent *sibling* node of N with $l, l > d$ entries
 - Insert $(K, N'.p_i)$ into N , where i is the leftmost(0) or rightmost entry(l)
 - Replace K in parent node with $N'.k_i$
 - Remove (p_i, k_i) entry from N'

Bulk Loading

1. Sort entries by search keys.
2. Load leaf pages with $2d$ entries
3. For each leaf page, insert index entry to rightmost parent page

Hash based Index

Static Hashing

- Data stored in N buckets, where hash function $h(\cdot)$ is used to id bucket
 - record with key k is inserted into B_i , where $i = h(k) \bmod N$
- Bucket is primary data page with 0+ overflow data pages

Linear Hashing

- Grows linearly by splitting buckets

- Systematic splitting: Bucket B_i is split before B_{i+1}
- Let $N_i = 2^i N_0$ be file size at beginning of round i
- How to split bucket B_i
 - Add bucket B_j (split image of B_i)
 - Redistribute entries in B_i between B_i and B_j
 - next++; if next == NLevel: (level++; next = 0)

Performance

- Average: 1.2 IO for uniform data
- Worst Case: Linear in number of entries

Extensible Hashing

- Overflowed bucket is resolved by splitting overflowed bucket
- No overflow pages, and order in which buckets are split is random
- Directory of pointers to buckets, directory has 2^d entries
 - d is global depth of hashed file
 - Each bucket maintains a local depth $l \in [0, d]$
 - Entries in a bucket of local depth l : same last l bits

Bucket Overflow

- Number of directory entries could be more than number of buckets
- Number of dir entries pointing to bucket = 2^{d-l}
- When bucket B with depth l overflows,
 - Increment local depth of B to $l + 1$
 - Allocate split image B'
 - Redistribute entries between B and B' using $(l + 1)$ th bit
- if $l + 1 >$ global depth d
 - Directory is doubled in size, , global depth to $d + 1$
 - New entries point to same bucket as corresponding entry
- if $l + 1 \leq$ global depth d
 - Update dir entry corresponding to split bucket's directory entry to point to split image

Bucket Deletion

- B_i & B_j (with same local depth l and differ only in l th bit) can be merged if entries fit bin bucket
 - B_i is deallocated, B_j 's local depth decremented by 1. Directory entries that point to B_i points to B_j

Performance

- At most 2 disk IOs for equality selection
- Collisions: If they have same hashed value.
 - Need overflow pages if collisions exceed page capacity

Sorting

Notation

$ r $	pages for R
$\ r\ $	tuples in r
$\pi_L(R)$	project column by list L from R
$\pi_L^*(R)$	project with duplicates
b_d	Data records that can fit on page
b_i	Data entries that can fit on page
b_r	RIDs that can fit on page

External Merge Sort

- **File size:** N pages
- Memory pages available: B
- **Pass 0:** Create sorted runs
 - Read and sort B pages at a time
- **Pass i:** Use $B - 1$ pages for input, 1 for output, performing $B - 1$ -way merge sort
- **Analysis**
 - Sorted runs: $N_0 = \lceil \frac{N}{B} \rceil$
 - Total passes: $\lceil \log_{B-1}(N_0) \rceil + 1$
 - Total I/O: $2N(\lceil \log_{B-1}(N_0) \rceil + 1)$

Optimized Merge Sort

- Read and write in blocks of b pages
 - Allocate 1 Block for output
 - Remaining memory for input: $\lfloor \frac{B}{b} \rfloor - 1$ blocks
- **Analysis**
 - sorted runs: $N_0 = \lceil \frac{N}{B} \rceil$
 - Runs Merged at each pass $F = \lfloor \frac{B}{b} \rfloor - 1$
 - No of merge passes: $\lceil \log_F(N_0) \rceil$ (+1 for total)
 - Total IO: $2N(\lceil \log_F(N_0) \rceil + 1)$
- **Sorting with B+ Trees:** IO Cost: h + Scan of leaf pages + Heap access (If not covering index)

Projection

Sort based approach

- Extract attributes, Sort attributes, remove duplicates
- **Analysis**
 1. Extract Attributes: $|R|(\text{scan}) + |\pi_L^*(R)|$ (output)
 2. Sort Attributes:
 - $N_0 = \lceil \frac{|\pi_L^*(R)|}{B} \rceil$
 - Merging Passes: $\log_{B-1}(N_0)$
 - Total IO: $2 |\pi_L^*(R)| (\log_{B-1}(N_0) + 1)$
 3. Remove Duplicates: $|\pi_L^*(R)|$

Optimized approach

- Merge Split step 2 into Creating and Merging sorted runs, and merge into step 1 and 3 respectively
- **Analysis**
 - **Step 1**
 - $B - 1$ pages for initial sorted run
 - Sorted Runs: $N_0 = \lceil \frac{|\pi_L^*(R)|}{B-1} \rceil$

- Create sorted run = $|R| + |\pi_L^*(R)|$
- **Step 2**
 - Merging passes: $\lceil \log_{B-1}(N_0) \rceil$
 - Cost of merging: $2 |\pi_L^*(R)| \lceil \log_{B-1}(N_0) \rceil$
 - Cost of merging excluding IO output: $(2 \lceil \log_{B-1}(N_0) \rceil - 1) |\pi_L^*(R)|$

Hash based approach

- **Partitioning**
 - Allocate 1 page for input, $B - 1$ page for output.
 - Read 1 page at a time, for each tuple, create projection, hash(h) to distribute to $B - 1$ buffers
 - Flush to disk when full.
- **Duplicate Elimination**
 - For each partition R_i , create hash table, hash each tuple with hash function $h' \neq h$ to bucket B_j if $t \notin B_j$

- **Partition Overflow:** hash table for $\pi_L^*(R_i)$ is larger than memory pages allocated for $\pi_L(R)$

- **Analysis**
 - IO Cost (no partition overflow): $|R| + 2|\pi_L^*(R)|$
 - Partitioning Phase: $|R| + |\pi_L^*(R)|$
 - Duplicate Elimination: $|\pi_L^*(R)|$
 - To Avoid partition overflows:
 - $|R_i| = \frac{|\pi_L^*(R)|}{B-1}$
 - $B >$ size of hash table, $|R_i| \times f$
 - $B > \sqrt{f \times |\pi_L^*(R)|}$

Selection

- **Conjunct:** $1 \geq$ terms connected by $\vee$
- **CNF predicate:** $1 \geq$ conjuncts connected by $\wedge$
- **Covered Conjunct** - predicate p_i is covered conjunct if each attribute in p_i is in key K or include column of Index I
 - $p = (\text{age} > 5) \wedge (\text{height} = 180) \wedge (\text{level} = 3)$
 - I_1 key = (level, weight, height)
 - p_c wrt $I_1 = (\text{height} = 180) \wedge (\text{level} = 3)$
- **Primary Conjunct**
 - I matches p if attributes in p form prefix of K and all comparison operators are equality except last
 - p_p is largest subset of conjuncts in p such that I matches p_p
- $\sigma_p(R)$: Select rows from R that satisfy predicate p
- Access Path: way of accessing data records / entries
 - **Table Scan:** Scan all data pages (Cost: $|R|$)
 - **Index Scan:** Scan index pages
 - **Index Combination:** Combine from multiple index scans
 - Scan/Combination can be followed by RID lookup to retrieve data
- **Index only plan:** Query where it does not need to access any data tuples in R

- **Covering Index:** I is covering index if all of R s attribute in query is part of the key / include columns of I

B+ Trees

- For Index Scan + RID Lookup, many matching RIDs could refer to same page
 - Sort matching RIDs before performing lookup: Avoid retrieving same page

Analysis

Cost of index scan = $N_{\text{internal}} + N_{\text{leaf}} + N_{\text{lookup}}$

- N_{internal} : No of internal nodes accessed
 - Height of B+ tree index
 - $$\text{height}(\text{est}) = \begin{cases} \left\lceil \log_F \left(\left\lceil \frac{|R|}{b_d} \right\rceil \right) \right\rceil & \text{if index is clustered} \\ \left\lceil \log_F \left(\left\lceil \frac{|R|}{b_i} \right\rceil \right) \right\rceil & \text{otherwise} \end{cases}$$
- N_{lookup} : Data pages accessed for RID lookups
 - If I is covering index for $\sigma_p(R)$, $N_{\text{lookup}} = 0$
 - else $N_{\text{lookup}} = \|\sigma_{p_c}(R)\|$
 - If matching RIDs are sorted before RID lookup
 - $N_{\text{lookup}} = N_{\text{sort}} + \min\{\|\sigma_{p_c}(R)\|, |R|\}$
- N_{sort} : sorting matching RIDs
 - $N_{\text{sort}} = 0$ if $\left\lceil \frac{\|\sigma_{p_c}(R)\|}{b_r} \right\rceil \leq B$ (if RIDs can fit into B)
 - $$N_{\text{sort}} = 2 \left\lceil \frac{\|\sigma_{p_c}(R)\|}{b_r} \right\rceil \lceil \log_{B-1}(N_0) \rceil, N_0 = \left\lceil \frac{\left\lceil \frac{\|\sigma_{p_c}(R)\|}{b_r} \right\rceil}{B} \right\rceil$$
 - Sorting with External Merge Sort
 - N_{sort} doesn't include read IO for pass 0 as its included in N_{internal} and N_{leaf}
 - N_{sort} doesn't include write IO for final merging pass as RID is used for lookup
- N_{leaf} : Leaf pages scanned for evaluating $\sigma_p(R)$
 - $N_{\text{leaf}} = \left\lceil \frac{\|\sigma_{p_p}(R)\|}{b_d} \right\rceil$ if clustered
 - $N_{\text{leaf}} = \left\lceil \frac{\|\sigma_{p_p}(R)\|}{b_d} \right\rceil$ if unclustered
- **Index Combination**
- Cost = $N_{\text{internal}}^p + N_{\text{leaf}}^p + N_{\text{internal}}^q + N_{\text{leaf}}^q + N_{\text{combine}} + N_{\text{lookup}}$
 - N_{combine} : IO cost to compute join of $\pi_p \pi_q$
 - If $\min\{|\pi_{X_p}(S_p)|, |\pi_{X_q}(S_q)|\} \leq B$
 - One of the join operands can fit in mem, then $N_{\text{combine}} = 0$

Hash based Index Scan

- Cost: $N_{\text{dir}} + N_{\text{bucket}} + N_{\text{lookup}}$
 - N_{dir} : no of directory pages accessed (1 if extensible hash, 0 otherwise)
 - N_{bucket} : max no of index's primary/overflow pages accessed
 - $N_{\text{lookup}} = N_{\text{sort}} + \min\{\|\sigma_{p_c}(R)\|, |R|\}$ if I is not covering index for $\sigma_p(R)$