Data Storage

- What is stored: Schemas, Relation metadata (e.g. indexes, statistical info), Log files

Secondary Storage

- DBMS storage includes:
    - Data: Stored in disk blocks/pages
    - File layer: Organisation and data retrieval
    - Buffer manager: Reading/writing of disk pages
    - Disk space manager: Keeps track of pages used by file layer

Magnetic Hard-Disk Drive (HDD)

- Disk Access time:
    1. Command Processing time (negligible)
    2. Seek time: Move disk head on track
    3. Rotational delay: Rotate to put head on start of correct sector
        a. Avg rotational delay = time for ½ revolution = ½ × $\frac{60}{\text{RPM}}$ s
    4. Transfer time: Rotate along the correct sector(s) to move data to/from disk = $\frac{\#\text{ sectors read}}{\text{Total }\#\text{ sectors on track}} \times \frac{60}{\text{RPM}}$ s
- Sequential I/O:
    - Pages stored contiguously on one track, then move on to next surface of the cylinder (i.e. same track across different surfaces), then move on to next cylinder
- **Solid-State Drive (SDD)**
    - Per block: Avg seek time + Avg rotational delay + Transfer time

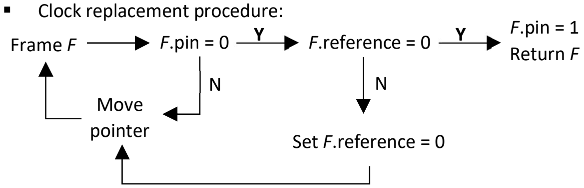
Buffer Manager

- Maintains a buffer pool in RAM (memory allocated for DBMS) for caching. Each unit of memory is called a **frame**
    - Each frame has:
    1. Pin count: # clients using the page
    2. Dirty flag: Whether page has been modified but not updated on disk
- Disk pages are fetched into/release from the buffer pool
- Page request procedure:
    1. Client requests page *P*
    2. Is page *P* already in memory?
        a. Yes: Pin frame *F*. Return address of *F*. **End**
        b. No: continue to 3
    3. Find free frame or evict a page if buffer pool is full (i.e. Find some frame *F*.**pin = 0**)
    4. Pin frame *F*
    5. Write frame *F* into disk if *F* is dirty
    6. Return address of frame *F*

Page Replacement Policies

Clock Replacement Policy

- Each frame has an additional reference bit
- Pointer moves in FIFO manner
- Only frame with reference bit = 0 AND pin = 0 will be replaced
- Reference bit set to 0 when pin drops to 0
- Clock replacement procedure:

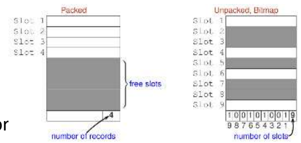


- **DO NOT** increment pointer after each successful page replacement

Page/Record Format

Fixed-length Records

- Each record is of fixed length.
- Each record is idenfitied by a RID = (page id, slot #)
- Packed: Contiguous; Unpacked: Not Contiguous
- Record Format: Fields in a recor



Variable-length Records (Slotted Page Organisation)

- Slot directory:
    - Pointer to start of free space: Memory location for next record to be inserted
    - Number of slots: Increments whenever a new record is inserted
    - Pointers to each record: Address to the start of a record in the page + size of the record (byte offset)
- Record Format:
    - Delimit fields with special symbols (e.g. F1 $ F2 $ .... $ Fn)
    - Use an array of field offsets

Tree-based Indexing

- **Index**: Data structure to speed up data retrieval; stored as a file.)
- **Search key:** Sequence of *k* data attributes e.g. (name, age)
- **Unique index**: Search key is a candidate key
- **Data entry**: (Key Value, RID), stored in index

B+ Tree Index

- **Leaf nodes** store data entries. Leaf nodes are doubly-linked and all of them are on the same level
- **Height** h: # levels of internal nodes (root is level 0). Leaf nodes are at level h
- **Internal nodes** store index entries in the form (p_0, k_1, p_1, k_2, p_2, ...p_n)
- **Order** of a B+ Tree, *d*: Root node must contain [1, 2d] entries and non-root nodes must contain [d, 2d] entries
- A B+ Tree with n level of internal nodes has with order d has $2(d+1)^{n-1} \leq$ # leafs $\leq (2d+1)^n$

Search

1. At each internal node, find largest k_i s.t. target $k_i \leq k$.
    a. Search subtree at p_i . if k_i exists
    b. Otherwise, search subtree at p_0
2. Continue until leaf node and return all entries with search key = *k*.
    a. If range search, traverse along leaf nodes and return all entries within the bound

Insert (Handling overflows)

- **Node splitting**
    - **Overflowing leaf node**
        1. Distribute the $d+1$ largest entries into new leaf node
        2. Create and insert new index entry using smallest key in new leaf node into parent node
        3. If parent node overflows, split parent node
    - **Overflowing internal node**
        1. Split at the middle key, and push it up to the parent node
        2. Propagate node splitting until no overflows/reached root
- **Redistribution of data entries**
    1. Redistribute entries in overflowed leaf node *N* by putting the largest /smallest entry (among the $2d+1$ entries) into adjacent right/left sibling N'
    2. Then, update the separating key in parent

Delete (Handling underflows)

- **Node Merging**
    - **Underflowing leaf node**
    1. Merge underflowed leaf node N with adjacent sibling N' by moving all entries from N' to N.
    2. Then, delete N' and the separating key in parent
    3. Update parent index if needed
- **Underflowing internal node** (Pre-condition: N' must have d entries)
    1. Merge underflowed internal node N with adjacent sibling N' by pulling down separating key in parent, combining N and N'
    2. Propagate node merging until no underflows/reached root
- **Redistribution of data entries** (Pre-condition: N' must have > d entries)
    1. Redistribute entries by moving the data entry with the smallest/largest key from right/left sibling N' to underflowing leaf node N
    2. Update separating key in parent with the smallest key in N'
- **Redistribution of internal entries**
    1. Merge leaf nodes, causing internal node N to be underflowed
    2. "Pull down" the separating parent key K between N and sibling N' and join with N
    3. Then, replace the K in the affected index entry in parent node with $N'.k_i$ (i.e. left/right-most key in right/left sibiling)
    4. Remove k_i from sibling

Data Formats

- **Format-1**: Leaves store data records
- **Format-2**: Leaves store (k, rid)
- **Format-3**: Leaves store (k, rid-list)

Bulk Loading

1. Sort data entries to be inserted by search key
2. Load the leaf pages with those sorted entries
3. Initialize the 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 the B+ tree

- Advantages:
    - Efficient construction
    - Leaf pages are allocated sequentially

Hash-based Indexing

- Numbers in the buckets are the **HASH VALUES**, NOT THE KEY VALUES

Linear Hashing

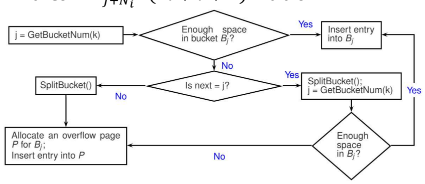
- N_0 = initial # of buckets = 2^m
- N_i = # buckets in at start of level $i = 2^i N_0 = \mathbf{2^{m+i}}$
- **Split image** of $B_j = B_{j+N_i}$

Insert

- Hash function: $h_i(k) = h(k)\ mod\ \boldsymbol{N_i}$ (look at the **last $\boldsymbol{m+i}$ bits**)

$$\text{Bucket \#} = \begin{cases} h_i(k), & h_i(k) \geq next \\ h_{i+1}(k), & otherwise \end{cases}$$

- **Splitting**: occurs when any bucket overflows:
  - Split the bucket B_j pointed to by '*next*'
  - Redistribute entries:
    - Entries in $B_j : (m+i+1)^{\text{th}}$ bit is **0**
    - Entries in $B_{j+N_i} : (m+i+1)^{\text{th}}$ bit is **1**



Delete

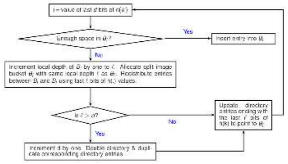
- If last bucket B_{N_i+next} is empty:
  - If next > 0
    1. Decrement next
    2. Delete last bucket
  - If next = 0 and level > 0
    1. Decrement level
    2. Update next to point to last bucket in previous level ($B_{N_{i-1}-1}$)
- Delete overflow pages that become empty after redistribution

Performance

- Best case: No overflow pages – 1 disk I/O per insertion
- Worst case: All hashed to the same bucket – linear I/O cost
- Average: 1.2 disk I/O per insertion

Extendible Hashing

- Global depth (directory) = d; # directory entries = 2^d
- Local depth (bucket) = $l \leq d$
- Directory entry # = **last $\boldsymbol{d}$ bits** of $h(k)$; points to bucket
- All entries in a bucket have same l bits in their $h(k)$
- Corresponding entries: differ only by the d^{th} bit (indexed 1)
- # directory entries pointing to a bucket = 2^{d-l}
- **Splitting**: occurs when target bucket B_i overflows
  1. Increment l of B_i
  2. Allocate new bucket B_j (split image) with same l
  3. Redistribute:
     - Entries in $B_i : l^{th}$ bit is 0
     - Entries in $B_j : l^{th}$ bit is 1
  4. Using the last l bits, redistribute pointer(s) between B_i and B_j
- Bucket B_i overfows:
  - If $l = d$
    1. Increment d and l of B_i
    2. Double the number of directory entries
    3. Split B_i; redistribute
    4. Redistribute pointer(s)
  - If $l \leq d$
    1. Increment l of B_i
    2. Split B_i; redistribute
    3. Redistribute pointer(s)



Delete

- If entries in B_i and B_j (corresponding entry) can fit into 1 bucket:
  1. Merge B_i and B_j into one bucket e.g. B_i
  2. Delete B_j
  3. Decrement l of B_i
  4. Move directory entries pointing to B_j to point to B_i
  5. If each pair of corresponding entries point to the same bucket, decrement d and halve the directory

Performance

- At most 2 disk I/O per insertion

Index Formats

Clustered vs Unclustered

- Clustered: Order of data records is close to order of the data entries
  - Format-1 index is a clustered index
  - Cost of RID lookups become 0 (format-1) or $\|\sigma(R)\|\ /\ b_d$
- Each relation can have at most one clustered index

Dense vs Sparse

- Dense: There is an index record for every search key value; otherwise, it is sparse
- Unclustered index must be dense
- Format-1 B+ Tree index is sparse

Query Evaluation: Sort

External Mergesort

- Assume N data records need to be sorted
- Assume that data records are on the disk
- Only B memory pages are allocated for sorting
  - 1 page is allocated for **output**
  - $B-1$ pages are allocated for **input**

1. Read the N data records into $\lceil N/B \rceil$ initial sorted runs
2. Recursively merge the sorted runs in a $B-1$ way merge
   a. With $B-1$ input pages, we can sort $B-1$ sorted runs by allocating one page for each of the $B-1$ sorted runs
   b. Each input pages has a pointer to the current smallest value
   c. When output page is full, write the page back to disk

Analysis

- N_0 = # sorted runs in initial pass (pass 0) = $\lceil N/B \rceil$
- Total # passes = $\lceil log_{B-1}(N_0) \rceil + 1$
- Total # I/O = $2N(\lceil log_{B-1}(N_0) \rceil + 1)$
  - Each pass has N reads and writes

External Mergesort with Blocked I/O

- Instead or reading/writing one page at a time, read/write in units of **buffer blocks** of b_i and b_o pages respectively
  - This allows for sequential I/O over random I/O
- Assume B memory pages are allocated for sorting
  - **Output** block size = b_o pages; **Input** block size = b_i pages
  - 1 block is allocated for **output**; $\left\lfloor \frac{B-b_o}{b_i} \right\rfloor$ blocks are allocated for **input**

1. Read the N data records into $\lceil N/B \rceil$ sorted runs
2. Recursively merge the sorted runs in a $\left\lfloor \frac{B-b_o}{b_i} \right\rfloor$ way merge
   a. With $\left\lfloor \frac{B-b_o}{b_i} \right\rfloor$ input blocks, we can sort $\left\lfloor \frac{B-b_o}{b_i} \right\rfloor$ sorted runs by allocating one block for each of the sorted runs
   b. When output block is full, write the block back to disk

Analysis

- N_0 = # sorted runs in initial pass (pass 0) = $\lceil N/B \rceil$
- F = # sorted runs that can be merged at each merge pass = $\left\lfloor \frac{B-b_o}{b_i} \right\rfloor$
- Total # passes = $\lceil log_F(N_0) \rceil + 1$
- Reduced merge factor, but more sequential I/O

Query Evaluation: Select $\boldsymbol{\sigma}$

- Access paths:
  - Table scan: Scan all data pages
  - Index scan: Scan index pages + RID lookup (if needed)
  - Index intersection: Combine results from multiple index scans + RID lookup (if needed)
- More selective access path → fewer pages need to be accessed
- **Include Columns**: Specify the attributes whose values are also stored in the data entries of the index, on top of the index key. Can be used to avoid RID lookups
- **Covering Index**: Index I is a covering index for query Q if all attributes rein Q are part of the key or include columns(s) of I
  - Then, Q is evaluated with index-only plan

Matching Predicates

- **B+ Tree**: $I = (K_1, K_2, \ldots, K_n)$. I matches predicate p if p is in the form:
  $$(K_1 = c_1) \wedge \ \ldots \ \wedge (K_{i-1} = c_{i-1}) \wedge (K_i\ op_i\ c_i), \quad i \in [1, n]$$
  - Prefix key with all equality operator, and at most one non-equality operator on last attribute of prefix
- **Hash Index**: $I = (K_1, K_2, \ldots, K_n)$. I matches predicate p if p is in the form:
  $$(K_1 = c_1) \wedge (K_2 = c_2) \wedge \ \ldots \ \wedge (K_n = c_n)$$
  - All attributes must appear in p and only equality operator

Primary Conjuncts p'

- The subset of conjuncts in p that I matches

Covered Conjuncts p_c

- The subset of conjuncts p_c in p such that all attributes in p_c appear in the key or include column(s) of I

Cost Evaluation

B+ Tree Index

$$Cost_{internal} = \begin{cases} \left\lceil log_F \left\lceil \frac{\|R\|}{b_d} \right\rceil \right\rceil, & \text{format} - 1 \\ \left\lceil log_F \left\lceil \frac{\|R\|}{b_i} \right\rceil \right\rceil, & \text{otherwise} \end{cases}$$

$$Cost_{leaf} = \begin{cases} \left\lceil \frac{\|\sigma_{p'}(R)\|}{b_d} \right\rceil, & \text{format} - 1 \\ \left\lceil \frac{\|\sigma_{p'}(R)\|}{b_i} \right\rceil, & \text{otherwise} \end{cases}$$

$$Cost_{RID} = \begin{cases} 0, & \text{format} - 1 \text{ or covering} \\ \|\sigma_{p_c}(R)\|, & \text{otherwise (worst case)} \\ \frac{\|\sigma_{p_c}(R)\|}{b_d}, & \text{otherwise (clustered)} \end{cases}$$

$$Cost_{total} = Cost_{internal} + Cost_{leaf} + Cost_{RID}$$

Hash Index

- Ranged query: Table scan = $|R|$
- Format 1
  - $Cost_{total} = Cost_{records} \geq \left\lceil \frac{\|\sigma_{p'}(R)\|}{b_d} \right\rceil$
- Format 2
  - $Cost_{entries} \geq \left\lceil \frac{\|\sigma_{p'}(R)\|}{b_i} \right\rceil$, due to possible long overflow chain
  - $Cost_{records} = \begin{cases} 0, & I \text{ is covering index} \\ \|\sigma_{p_c}(R)\|, & \text{otherwise} \end{cases}$
  - $Cost_{total} = Cost_{records} + Cost_{entries}$

Query Evaluation: Project π

- $\pi_L(R)$ – No duplicates (select DISTINCT) ; $\pi_L^*(R)$ – Keep duplicates

Sort-Based Projection

Unoptimized Approach

1. Extract attributes L from records R → $\pi_L^*(R)$
2. Sort $\pi_L^*(R)$ using L as sort key → sorted $\pi_L^*(R)$ [External Mergesort]
3. Scan $\pi_L^*(R)$ to remove duplicates → $\pi_L(R)$

- **Analysis**

1. Read I/O (table scan) = $|R|$; Write I/O = $|\pi_L^*(R)|$ → $|R| + |\pi_L^*(R)|$
2. Given B buffer pages for sorting:
   - Merge factor = $B - 1$; $N_0 = \lceil \pi_L^*(R)/B \rceil$
   - Step 2 total = $2|\pi_L^*(R)|(log_{B-1}N_0 + 1)$
3. Step 3 total = $|\pi_L^*(R)|$ (ignore write I/O as cost)
4. Total = $|R| + 2|\pi_L^*(R)|(log_{B-1}N_0 + 2)$

Optimized Approach

1. Create sorted runs with attributes L
   - Write only attributes L to output page
2. Merge sorted runs and remove duplicates simultaneously

- **Analysis**

1. Read I/O = $|R|$; Write I/O = $|\pi_L^*(R)|$→ Step 1 total = $|R| + |\pi_L^*(R)|$
2. Given B buffer pages for sorting:
   - Merge factor = $B - 1$; $N_0 = \lceil \pi_L^*(R)/B \rceil$
   - Step 2 total = $2|\pi_L^*(R)|(log_{B-1}N_0) - |\pi_L^*(R)|$
3. Total = $|R| + 2|\pi_L^*(R)|(log_{B-1}N_0)$

- If $B > \sqrt{|\pi_L^*(R)|}$,
  - # initial sorted runs = $\left\lceil \frac{|R|}{B} \right\rceil \approx \sqrt{|\pi_L^*(R)|}$
  - # merging passes = $log_{B-1}\left(\sqrt{|\pi_L^*(R)|}\right) \approx 1$
  - Total = $|R| + 2|\pi_L^*(t)|$

Hash-Based Projection

1. **Partition** all the tuples in R into $R_1, R_2, \dots, R_{B-1}$
   - Only B memory pages are allocated for hash table
     - 1 page is allocated for **input** buffer
     - $B - 1$ pages for **output** buffers/partitions $R_1, R_2, \dots, R_{B-1}$
   - For each t in R,
     1.1 $h(\pi_L(t))$, then output $\pi_L(t)$ into output buffer $R_{h(\pi_L(t))}$
     1.2 If $R_{h(\pi_L(t))}$ is full, flush to disk.
   - Output of partition phase is $\pi_L^*(R_1), \pi_L^*(R_2), \dots, \pi_L^*(R_{B-1})$
2. **Duplicate Elimination**: For each $\pi_L^*(R_i)$,
   2.1 Initialize hash table T of size $B - 1$
   2.2 For each tuple t in $\pi_L^*(R_i)$:
      2.2.1 Perform $h'(t) = j$ (NOTE: $h' \neq h$)
      2.2.2 If t not in B_j, then insert t into B_j
   2.3 Output all the tuples in T as $\pi_L(R_i)$
3. Combine all $\pi_L(R_i)$

- Entire T must fit in main memory. If not, recursively partition $\pi_L^*(R_i)$ into $\pi_L^*(R_{i_1}), \dots, \pi_L^*(R_{i_{B-1}})$

Analysis

- Assuming no partition overflow:
  1. Partitioning = $|R| + |\pi_L^*(R)|$
     - Read I/O (table scan) = $|R|$.
     - Each t is projected before writing. So, write I/O = $|\pi_L^*(R)|$.
  2. Duplicate elimination = $|\pi_L^*(R)|$
     - Each of the projected tuples is read once. Ignore write I/O
  3. Total = $|R| + 2|\pi_L^*(t)|$
- If $B > \sqrt{f|\pi_L^*(R)|}$, then there will not be partition overflow
  1. Assume h hashes every t in R uniformly
  2. Each R_i will have $\approx \frac{|\pi_L^*(R)|}{B-1}$ pages → $B > f\frac{|\pi_L^*(R)|}{B-1} \approx \sqrt{f|\pi_L^*(R)|}$

Indexing

- If there is a covering index I for the projection, then replace table scan (i.e. read I/O = $|R|$) in the both projection schemes with index scan
- If index is ordered (e.g. B+-tree) whose search key (e.g. (A, B, C) includes wanted attributes (e.g. $\pi_{A,B}(R)$) as a prefix
  1. Scan all the data entries in order
  2. Compare adjacent data entries for duplicates

Query Evaluation: Join ⋈

- In general, the smaller relation should be the outer relation

Tuple-based Nested Loop Join

- For each <u>tuple</u> r in R, for each <u>tuple</u> s in S, check if r matches s

Analysis

- Scan R = $|R|$; Scan S = $\|R\| \times |S|$
- Total = $|R| + \|R\| \times |S|$

Page-based Nested Loop Join

- For each <u>page</u> P_r in R, for each <u>page</u> P_s in S, for each <u>tuple</u> r in P_r, for each <u>tuple</u> s in P_s, check if r matches s

Analysis

- Scan R = $|R|$; Scan S = $|R| \times |S|$
- Total = $|R| + |R| \times |S|$

Block Nested Loop Join

- Allocate 1 buffer page for output, 1 buffer page for S and $B - 2$ buffer pages for R. Read pages from R in blocks of $B - 2$
- For each <u>block</u> B_i from R, for each <u>page</u> P_s in S, for each <u>tuple</u> r in B_i, for each <u>tuple</u> s in P_s, check if r matches s

Analysis

- Scan R = $|R|$; Scan S = $\left\lceil \frac{|R|}{B-2} \right\rceil \times |S|$
- Total = $|R| + \left\lceil \frac{|R|}{B-2} \right\rceil \times |S|$

Index Nested Loop Join

- Precondition: There is an index on the join attribute(s) of **inner** relation
- For each tuple r in R, use r to probe S's index to find matching tuples

Analysis

- Assuming uniform distribution, one tuple in R matches with $\left\lceil \frac{\|S\|}{\|\pi_{B(S)}\|} \right\rceil$ tuples
- Scan R = $|R|$; Scan S = $\|R\| \times \left(log_F \left\lceil \frac{\|S\|}{b_d} \right\rceil + \left\lceil \frac{\|S\|}{b_d \|\pi_{B(S)}\|} \right\rceil \right)$
- Total = $|R| + \|R\| \times \left(log_F \left\lceil \frac{\|S\|}{b_d} \right\rceil + \left\lceil \frac{\|S\|}{b_d \|\pi_{B(S)}\|} \right\rceil \right)$

Sort-Merge Join

- Assume $|R| \leq |S|$; R is outer relation

Unoptimized Approach

1. **Sort**: Sort both R and S based on ⋈ attributes
   - This partitions R into $R_1, \dots R_k$ and S into $S_1, \dots S_l$ each containing tupls with the same join attribute value(s)
2. **Merge**:
   2.1. Initialize pointers p_r and p_s for R and S, each pointing to the first tuple. Let r and s be the tuples pointed to by p_r and p_s
   2.2. r and s do not match: Advance smaller pointer
      2.2.1 If p_r is advanced and r matches s at p_s', **rewind**: p_s ← p_s'
   2.3. r and s match:
      2.3.1 Remember position p_s : p_s' ← p_s
      2.3.2 Output $r \bowtie s$ and advance p_s until r and s do not match

- **Analysis**
  - Sort R and S:
    - External Mergesort = $2|R|(log_m(N_R) + 1) + 2|S|(log_m(N_S) + 1)$
    - Internal Mergesort = $|R| + |S|$
  - Merge:
    - Best case: No rewinds = $|R| + |S|$
    - Worst case: All tuples in R and S match = $|R| + \|R\| \times |S|$

Optimized Approach

- Start merging as soon as sorted runs from R and S can fit into memory
  - $B > N(R, i) + N(S, j)$, i, j = # passes from sorting R, S

1. Create sorted runs of R and merge partially to get (R, i)
2. Create sorted runs of S and merge partially to get (S, j)
3. Merge (R, i) and (S, j)

- **Analysis:** If $B > \sqrt{2|S|}$
  - # initial sorted runs of $S < \sqrt{\frac{|S|}{2}}$
  - Total # initial sorted runs of R and $S < \sqrt{2|S|}$
  - 1 pass is sufficient to merge and join the initial sorted runs
  - Total cost = $2(|R| + |S|) + (|R| + |S|) = 3(|R| + |S|)$

Grace Hash Join

- Assume R = build relation, S = probe relation

1. Partition R into $R_1, \ldots, R_{B-1}$ by h
2. Partition S using same h into $S_1, \ldots, S_{B-1}$
3. For each R_i, **build hashtable**
   - 3.1. Allocate 1 buffer page for input, 1 page for output and $B - 2$ pages for hashtable T
   - 3.2. For each tuple r in R_i, read it into input buffer and hash it into T using h', $h \neq h'$
   - 3.3. For each tuple s in S_i, read it into input buffer and **probe** T: If s matches with any r in bucket $h'(s)$, write $r \bowtie s$ into output buffer

Analysis

- By UHA: $|R_i| = \frac{|R|}{B-1}$. Let size of T be $f \times \frac{|R|}{B-1}$, f = fudge factor
  - Hence, $B > \frac{f|R|}{B-1} + 2 \approx \sqrt{f|R|}$ (1 input buffer, 1 output buffer) to prevent partition overflow
- Assuming no partition overflow:
  1. Partitioning = $2(|R| + |S|)$
  2. Probing = $|R| + |S|$
     - Read each page of R_i to build T
     - Read each page of S_i to probe
  3. Total = $3(|R| + |S|)$

Query Evaluation: Misc Operations

Set Operations

- **Intersection** $R \cap S$
  - Join with join predicate involving all columns of R and S
- **Cross Product** $R \times S$
  - Join with join predicate = true (trivial)
- **Union** $R \cup S$
  - Sorting approach:
    1. Sort R, sort S (on all attrs)
    2. Combine R and S and removing duplicates
  - Hashing approach: ≈ Grace Hash Join
    1. Partition (on all attrs) R into $R_1, \ldots, R_{B-1}$,
    2. Partition S into $S_1, \ldots, S_{B-1}$
    3. Build hash table T_i for each R_i (suppose R is build relation)
    4. For each $t \in S_i$, probe T_i and insert only if t not in T_i
- **Difference** $R - S$
  - Sorting approach: ≈ Sort-Merge Join using R as outer relation
    1. Sort R, sort S (on all attrs)
    2. Remove $t \in R$ if $t \in S$
  - Hashing approach: ≈ Grace Hash, using R as build relation
    1. Partition (on all attrs) R into $R_1, \ldots, R_{B-1}$,
    2. Partition S into $S_1, \ldots, S_{B-1}$
    3. Build hash table T_i for each R_i
    4. For each $t \in S_i$, probe T_i and discard t from T_i if t in T_i

Aggregate Operations

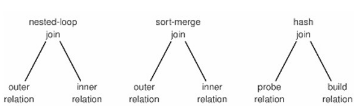
- **Simple Aggregation**: Maintain running info while scanning table
  - Valid for SUM, COUNT, AVG, MIN, MAX
- **Group-by Aggregation**
  - Sorting approach
    1. Sort relation by 'GROUP BY' attributes
    2. Scan relation and compute aggregate for each group
  - Hashing approach
    1. Scan relation to build hash table on 'GROUP BY' attributes
    2. Maintain running info for each group
- **Using Index**
  - Use index I over table scan if I is a covering index for aggregation operation

Query Evaluation Approaches

- **Materialized evaluation**
  - An operator is evaluated only when all of its operands has been completely evaluated/materialized
  - Materialize intermediate results to disk
- **Pipelined evaluation**
  - Pass the output directly to its parent operator (no materialize)
  - Execution of operators is interleaved
  - Blocking operator: Operator that is unable to produce any output until it has received all the tuples from its child operators
- **Iterator Interface of Pipelined evaluation**
  1. open – initialization; allocates resources and operators' args
  2. getNext – generates next output tuple/null if no more output
  3. close: deallocate state information

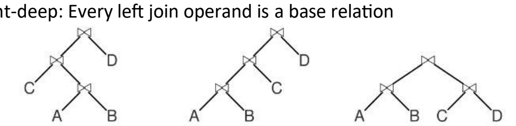
Query Optimization

- Join Plan Notation:



Commutative 1.1. $R \times S \equiv S \times R$ 1.2. $R \bowtie S \equiv S \bowtie R$	**Commutating σ with π** 4.1. $\pi_L(\sigma_p(R)) \equiv \pi_L(\sigma_p(\pi_{L \cup \text{attr}(p)}(R)))$
Associative 2.1. $(R \times S) \times T \equiv R \times (S \times T)$ 2.2. $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$	**Commutating σ with $\times/\bowtie$** 5.1. $\sigma_p(R \times S) \equiv \sigma_p(R) \times S$, $attr(p) \subseteq \text{attr}(R)$ 5.2. $\sigma_p(R \bowtie_q S) \equiv \sigma_p(R) \bowtie_q S$, $attr(p) \subseteq \text{attr}(R)$ 5.3. $\sigma_p(R \cup S) \equiv \sigma_p(R) \cup \sigma_p(S)$
Idempotence 3.1. $\pi_A(\pi_B(R)) \equiv \pi_A(R)$, $A \subseteq B \subseteq \text{attr}(R)$ 3.2. $\sigma_{p_1}\left(\sigma_{p_2}(R)\right) \equiv \sigma_{p_1 \wedge p_2}(R)$	**Commutating π with $\times/\bowtie$** 6.1. $\pi_L(R \times S) \equiv \pi_{L_R}(R) \times \pi_{L_S}(S)$ 6.2. $\pi_L(R \bowtie_p S) \equiv \pi_{L_R}(R) \bowtie_p \pi_{L_S}(S)$, $\text{attr}(p) \cap \text{attr}(R) \subseteq L_R$ & $\text{attr}(p) \cap \text{attr}(S) \subseteq L_S$ 6.3. $\pi_L(R \cup S) \equiv \pi_L(R) \cup \pi_L(S)$

- Query Plan Trees:
  - Linear: At least 1 operand for each join operation is a base relation
  - Bushy: There is a join operation where no operand is a base relation
  - Left-deep: Every right join operand is a base relation
  - Right-deep: Every left join operand is a base relation



Linear tree　Left-deep tree　Bushy tree

Query Plan Enumeration

- Uses a bottom-up Dynamic Programming approach starting with size-1 relations and memoizing the best plan
- For every possible combination S of R_i, for every possible pair of paritions S_i and S_j in S, find the optPlan(S) of joining S_i and S_j using the memoized optPlan(S_i) and optPlan(S_j)

```
Input: A SPJ query q on relations R1, R2, ···, Rn
Output: An optimal query plan for q
01.  for i = 1 to n do
02.      optPlan({Ri}) = best access plan for Ri
03.  for i = 2 to n do
04.      for each S ⊆ {R1, ···, Rn}, |S| = i do
05.          bestPlan = dummy plan with cost(bestPlan) = ∞
06.          for each Sj, Sk, |Sj| ∈ [1, i), S = Sj ∪ Sk do
07.              p = best way to join optPlan(Sj) and optPlan(Sk)
08.              if (cost(p) ≤ cost(bestplan)) then
09.                  bestPlan = p
10.          optPlan(S) = bestPlan
11.  return optPlan({R1, ···, Rn})
```

System R Optimizer

- Uses an enhanced Dynamic Programming approach that also considers sort order of query plan's output
  - Maintains optPlan(S_i, o_i), o_i = sort order of output by query plan wrt S_i
  - $o_i = \text{null}$ if output is unordered or a sequence of attributes
- Prunes search space:
  - Considers only left-deep query plans
  - Avoids cross-product query plans
  - Considers early selections and projections

Query Plan Cost Estimation

- Reduction factor $rf(t_i)$: fraction of tuples in e that satisfy t_i i.e. $rf(t_i) = \frac{\|\sigma_{t_i}(e)\|}{\|e\|}$

Assumptions		
Uniformity	Independence	Inclusion
Uniform distribution of attribute values	Independent distribution of values in diff attributes	For $R \bowtie_{R.A=S.B} S$, if $\|\pi_A(R)\| \leq \|\pi_B(S)\|$, then $\pi_A(R) \subseteq \pi_B(S)$
$rf(A = c) \approx \frac{1}{\|\pi_A(R)\|}$	$rf(t_i \wedge t_j) \approx rf(t_i) \times rf(t_j)$	$rf(R.A = S.B) \approx \frac{1}{max\{\|\pi_A(R)\|, \|\pi_B(S)\|\}}$

Equiwidth Histogram

- Each bucket has almost equal number of values
- All buckets have the same width/range size of B
- $\|\sigma_{A=c}(R)\| = \frac{\|b_i\|}{B}$
- $\left\|\sigma_{A\in[x,y]}(R)\right\| = \frac{f_1\|b_i\|}{B} + \|b_{i+1}\| + \cdots + \frac{f_2\|b_{i+k}\|}{B}$

Equidepth Histogram

- Each bucket has almost equal number of tuples. Let B_i denote the width of bucket b_i
- A value can be contained in multiple buckets
- $\|\sigma_{A=c}(R)\| = \frac{f_1\|b_i\|}{B_i} + \frac{f_2\|b_{i+1}\|}{B_{i+1}} + \cdots + \frac{f_k\|b_{i+k}\|}{B_{i+k}}$, for all b_j containing c
- $\left\|\sigma_{A\in[x,y]}(R)\right\| = \frac{f_1\|b_i\|}{B_i} + \frac{f_2\|b_{i+1}\|}{B_{i+1}} + \cdots + \frac{f_k\|b_{i+k}\|}{B_{i+k}}$

Histogram with MCV

- Separately keep tract of the frequencies of the top-k most common values and exclude them from the histogram