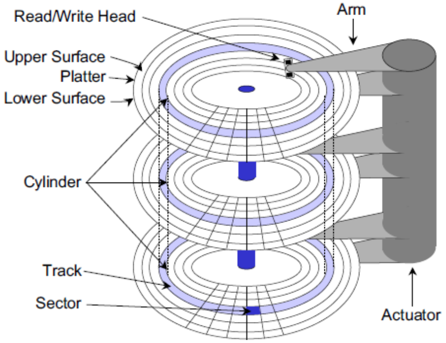
CS3223 DATABASE SYSTEMS IMPLEMENTATION - MIDTERM CHEATSHEET

Lecture 1: Data Storage



Frequent, random access to small file → Middle Tracks

Seq. scans of large files → Outer tracks

Random access to large file via index → File & Index on inner tracks

Seq. scan of small file → Inner half (as good as random IO)

Command processing time (*negligible*): interpret access cmd by disk controller; **Seek time**: moving arms to position disk head on track *(5-6ms avg)*.; **Rotational delay**: waiting for block to rotate under head; **Transfer time**: actually moving data to/from disk surface

$\boldsymbol{Access\ time\ =\ seek\ time\ +\ rotational\ delay\ +\ transfer\ time}$

$\boldsymbol{Response\ time\ for\ disk\ access\ =\ queuing\ delay\ +\ access\ time}$

$\boldsymbol{Average\ rotational\ delay\ =\ time\ for\ ½\ revolution}$

Example: For 10000 RPM, avg. rotational delay = 0.5(60/10000) = 3ms

$\boldsymbol{transfer\ time\ =\ n\ *\ (time\ for\ 1\ rev\ /\ [\#\ of\ sectors/track])}$

n = # of requested sectors on same track

Storage Manager Components

Data is stored & retrieved in units called **disk blocks** (or **pages**) where each block = sequence of one or more contiguous sectors. **File & access methods layer** – deals with organization and retrieval of data. **Buffer Manager** – controls reading/writing of disk pages. **Disk Space Manager** – keeps track of pages used by file layer. **Buffer Pool** = Main memory allocated for DBMS. Buffer pool is partitioned into block-sized pages called **frames**. Clients can request for a disk page to be fetched into buffer pool or release a disk page in buffer pool. A page in the buffer is **dirty** if it has been modified & not updated on disk. Two variables are maintained for each frame in buffer pool: **pin count** (number of clients using page) and **dirty flag** (whether page is dirty).

Handling a request for page p: if p is in frame f: increment pin count & ret addr(f), else: choose frame f' for replacement and increment pin count of f'. If f' was dirty write to disk. Read p into f' and return addr(f').

Incrementing/decrementing the pincount is called **pinning/unpinning** the page. When unpinning, set dirty flag if page is dirty. Page in buffer can only be replaced if the pincount=0. Before replacing a page, write to disk if dirty flag is true. Buffer manager coordinates with tx manager to ensure data correctness and recoverability.

CLOCK – a variant of LRU. current variable points to some buffer frame. Each frame has a referenced bit – turns on when pincount becomes 0. Replace a page that has ref bit off & pincount = 0. LRU & Clock are not good when user requires sequential scans of the data.

Heap File Implementations: Linked List of pages with free space & full pages; Page directory implementation.

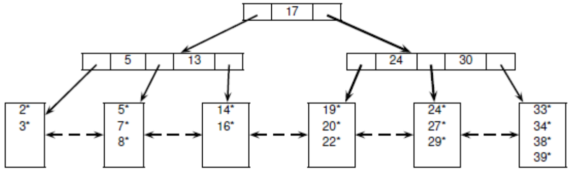
Page Formats

RID = (page id, slot number). For fixed-length records, **packed organisation**: store records in contiguous slots, **unpacked organisation**: use bit array to maintain free slots. Fixed-length-records: store fields consecutively. Variable-length-records: delimit fields with special symbols or use an array of field offsets.

[F1,F2,F3,F4] *[F1,$,F2,$,F3,$,F4]* *[o1,o2,o3,o4,F1,F2,F3,F4]*

Lecture 2: Indexing

An **index** is a data structure to speedup retrieval of data records based on some search key. A search key is a **composite search key** if it has >1 attributes. An index is a **unique index** if its search key is a candidate key. An index is stored as a file & records in an index file are referred to as **data entries**.



Leaf nodes store **sorted** data entries. Leaf nodes are doubly-linked. Internal nodes store index entries of the form $(p_0, k_1, p_1, \ldots, p_n)$. $k_1 < k_2 < \cdots < k_n$. p_i = disk page address (root node of an index subtree T_i). For each data entry k* in T_0, $k < k_1$. For each data entry k* in T_i $\left(i \in [1, n)\right)$, $k \in [k_i, k_{i+1})$. Each (k_i, p_i) is an index entry; k_i serves as a separator between the node contents pointed by p_{i-1} & p_i.

Order of index tree, $d \in Z^+$

1. Controls space utilization of index nodes
2. Each **non-root node contains** $[\boldsymbol{d}, \boldsymbol{2d}]$ entries
3. The **root node contains** $[\mathbf{1}, \boldsymbol{2d}]$ entries

To find the starting key: at each internal node N, find the largest key k_i in N s.t. $k \geq k_i$. If k_i exists, then search subtree at p_i, otherwise search subtree at p_0.

Format of Data Entries: Format 1: k*: **actual data record** (with search key value k) **Format 2**: k*: **(k, rid)**, where rid is the record identifier of a data record with search key value k; **Format 3**: k*: **(k, rid-list)** rid-list is a list of rid

Insertion: Splitting of overflowed node: Split overflowed leaf node by distributing d+1 entries to new leaf node. Create a new index entry using smallest key in new leaf node. Insert new index entry into parent node of overflowed node. **Propagation of node splits:** When splitting internal node, middle key is pushed up to parent node. **Redistribution:** Node split can be avoided by distributing entries from overflowed node to non-full adjacent sibling node: place entry into full leaf node N, then insert the last entry into N' (sibling node), then update index entry of N'. [Check right then left].

Deletion: Redistribution: an underflowed node could be balanced using adjacent sibling's entry (make sure each leaf node has $[d, 2d]$ entries). **Merging of nodes:** an underflowed node needs to be merged if each of its adjacent sibling node has exactly d entries. **Propagation of node merges:** pull down appropriate key from parent node to form merged node.

Bulk Loading a B+ Tree: Efficient construction algorithm & leaf pages are allocated sequentially.

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

Lecture 3: Hashing

Used for **equality queries** but not for range queries.

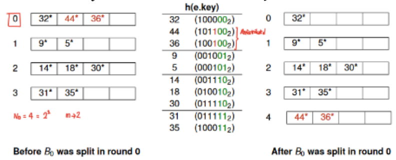
Static Hashing: Data is stored in N buckets B_0, B_1, B_{N-1} (N fixed at creation time). Record with search key k is inserted into bucket B_i where $i = h(k)\ mod\ N$. **Each bucket consists of one primary data page & a chain of zero or more overflow data pages.** v* denotes a data entry e with h(e.key)=v. e.key denotes the index's search key value of e.

Linear Hashing: Dynamic hashing technique; Hash file grows linearly by systematic splitting of buckets; Overflow pages are needed since an overflowed bucket might not be split immediately; **An insertion into a bucket Bi overflows if all the pages in Bi (i.e., primary & overflow pages) are full**. Assume initial file size of N_0 buckets. File grows linearly by **splitting buckets** in rounds. Let N_i denote the file size at the beginning of round i, $N_i = 2^i N_0$. At the end of round I, N_i new buckets are added: $B_{N_i}, B_{N_i+1}, \ldots, B_{2N_i-1}$. In round I, the split image of B_j is $B_{N_i+j} \mid j \in [0, N_i - 1]$.

Dynamic Hashing

$N_0 = 2^m,\ N_i = 2^i N_0 = 2^{m+i}.\ h_i(k) = h(k) mod N_i = last\ (m+i) bits\ of\ h(k)$

Consider **splitting of $\boldsymbol{B_j}$ in round i**: Before splitting: all entries in B_j have $h_i(e.key) = j$ (same last m+i bits); After splitting: e remains in B_j iff the last (m+i+1) bit of h(e.key) is 0.



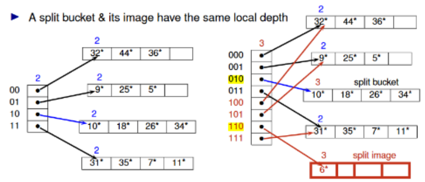
Splitting Buckets: *Next* specifies the next bucket to be split. Initialize next to 0 at the start of each round. Buckets that have split uses h_{i+1}, yet to split uses h_i, split images uses h_{i+1}. Assume that a bucket split is triggered whenever some bucket overflows. Overflow pages are needed since an overflowed bucket might not be split immediately.

Deletion: Case 1: If next > 0, decrement next by one ; i.e. if last bucket becomes empty, it can be removed. **Case 2**: If (next =0) and (level > 0), update next to point to last bucket in previous level, then decrement level by one. E.g. [level=1, N0=2, next=0] -> [level=0, N0=2, next=1].

Linear Hashing Performance: One disk I/O unless the bucket has overflow pages. On average: 1.2 disk IO for uniform data distribution. Worst Case: IO is linear in the number of data entries. ***Poor space utilization with skewed data distribution***.

Extendible Hashing

Case 1: Split bucket's local depth = global depth: double the directory, increment d, increment l by one whenever the bucket splits, split & image have same l. When dir is doubled, each new dir entry points to the same bucket as its corresponding entry.



Case 2: Split bucket's local depth < global depth: Directory is not expanded (repoint the dir ptr).

Deletion: Locate bucket B_i containing entry & delete entry. If B_i becomes empty, B_i can be merged with bucket B_j where i & j differs only in the lth bit. $delete(B_i); l{-}{=}1, update\ dir\ entries\ from\ B_i \rightarrow B_j$. More generally, Bi & Bj can merge if their entries can fit within a bucket. If each pair of corresponding entries point to the same bucket, directory can be halved (d--).

Extendible Hashing Performance: At most 2 disk IO for equality selection (atmost 1 disk IO if directory fits in main memory). Overflow pages needed when number of collisions exceed page capacity.

Lecture 4: Sorting & Selection

Used to produce a sorted table of results, bulk loading a B+tree index, implement other algebra operators such as projection & join.

External Merge Sort *Suppose file size of N pages and B number of buffer pages.*

Pass 0: Creation of sorted runs

- Read in and sort B pages at a time
- Number of sorted runs created = $\lceil N/B \rceil$
- Size of each sorted run = B pages (except possible the last run)

Pass i, $\boldsymbol{i \geq 1}$: Merging of sorted runs

- Use B-1 Buffer pages for input & one buffer page for output
- Perform (B-1) way merge

Analysis:

- N_0= number of sorted runs created in pass 0 = $\lceil N/B \rceil$
- 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 & writes N pages

Optimization with Blocked I/O *R/W in units of **buffer blocks of b pages**.*

- Allocate one block (b pages) for output
- Remainder space can ammodate $\left\lfloor\frac{B-b}{b}\right\rfloor$ blocks for input
- Can merge at most $\left\lfloor\frac{B-b}{b}\right\rfloor$ sorted runs in each merge pass

Analysis:

- N_0= number of initial sorted runs $= \lceil N/B \rceil$
- F = number of runs that can be merged at each merge pass $= \left\lfloor\frac{B}{b}\right\rfloor - 1$
- Number of passes $= \lceil \log_F N_0 \rceil + 1$

Sorting using B+ trees

When table to be sorted has a B+ tree index on sorting attribute → Format 1: sequentially scan leaf pages of B+ tree; Format 2/3: Sequentially scan leaf pages and for each page visited, retrieve data records using RIDs. An index is a **clustered index** if the order of its data entries is the same as or 'close to' the order of the data records. An **index using format 1 is a clustered index**. There is at most one clustered index for each relation.

Access Path: refers to a way of accessing data records/entries. **Table scan** = scan all data pages, **Index scan** = scan index pages, **Index intersection** = combine results from multiple index scans (e.g. intersect, union). **Selectivity of access path =** number of index & data pages retrieved to access data records. Most selective path retrieves least pages. An index I is a **covering index** for query Q if all attributes references in Q are part of the key of I [Q can be evaluated using I without any RID lookup → **index-only plan**].

CNF Predicates: A **term** is of the form $R.A\ op\ c$ or $R.A_i\ op\ R.A_j$. A **conjunct** consists of one or more terms connected by ∨. A conjunct that contains ∨ is said to be **disjunctive**. A **conjunctive normal form (CNF)** predicate consists of one or more conjuncts connected by ∧.

B+ Tree matching predicates: B+ tree index $I = (K_1, \ldots, K_n)$. Non-disjunctive CNF predicate p. I matches p if p is of the form : $(\boldsymbol{K_1 = c_1}) \wedge \ldots \wedge (\boldsymbol{K_{i-1} = c_{i-1}}) \wedge (\boldsymbol{K_i\ op_i\ c_i}), \boldsymbol{i \in [1, n]}$ where $(K_1, \ldots, K_i)$ is a prefix of the key of I, and there is atmost one non-equality comparison operator which must be on the last attribute of the prefix K_i. **Hash Index matching predicates:** I matches p if p is of the form : $(\boldsymbol{K_1 = c_1}) \wedge (\boldsymbol{K_2 = c_2}) \wedge \ldots \wedge (\boldsymbol{K_n = c_n})$

Primary Conjuncts: The subset of conjuncts in p that I matches
Covered Conjuncts: The subset of conjuncts in p that are covered by I. Each attribute in covered conjuncts appears in the key of I. $primary\ conjunct \subseteq convered\ conjuncts$.

Cost of B+tree Index evaluation of p: Let p' = primary conjuncts, pc = covered conjuncts:

1. Navigate internal nodes to locate first leaf page $Cost_{internal} = \left\lceil \log_F\left(\left\lceil\frac{||R||}{b_d}\right\rceil\right)\right\rceil$
2. Scan leaf pages to access all qualifying entry $Cost_{leaf} = \left\lceil\frac{||\sigma_p'(R)||}{b_d}\right\rceil$
3. Retrieve qualified data records via RID lookup $Cost_{RID} = 0\ or\ \left|\left|\sigma_{pc}(R)\right|\right|$, cost is zero if I is a covering or format-1 index. Cost of RID lookups could be reduced by first sorting the RIDs: $\left\lceil\frac{||\sigma_{pc}(R)||}{b_d}\right\rceil \le Cost_{rid} \le \min\left\{\left|\left|\sigma_{pc}(R)\right|\right|, |R|\right\}$

Cost of hash index evaluation of p:

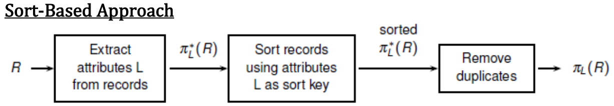
Format-1 Index: at least $\left\lceil\frac{||\sigma_p'(R)||}{b_d}\right\rceil$. Format-2 Index: at least $\left\lceil\frac{||\sigma_p'(R)||}{b_d}\right\rceil$.

Format 2: Cost to retrieve data records = 0 if I is covering index, otherwise $\left|\left|\sigma_p'(R)\right|\right|$.

Lecture 5: Projection & Join

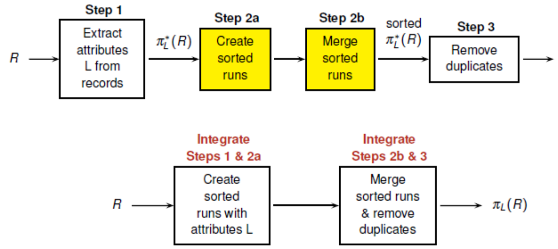
$\pi_L(R)$ projects columns given by list L from relation R. $\pi_L^*(R)$ preserves duplicates. Projection involves: 1. removing unwanted attributes, 2. eliminate any duplicate tuples produced.

Sort-Based Approach



Step 1:	Cost to scan record $= \|R\|$ Cost to output temporary result $= \|\pi_L^*(R)\|$
Step 2:	Cost to sort records $= 2\|\pi_L^*(R)\|(\log_m(N_0) + 1)$ N_0 = number of initial sorted runs, $m = B - 1$ = merge factor
Step 3:	Cost to scan records $= \|\pi_L^*(R)\|$

Optimized Sort-Based Approach



Hash-Based Approach: Build a main-memory hashtable to detect & remove duplicates.

Phase 1 **Partitioning phase**: partitions R into $R_1, R_2, \ldots, R_{B-1}$.

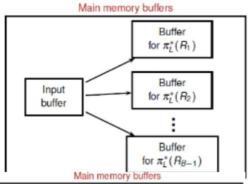
- Hash on $\pi_L(t)$ for each tuple $t \in R$.
- $R = R_1 \cup R_2 \cup \ldots \cup R_{B-1}$
- $\pi_L^*(R_i) \cap \pi_L^*(R_j) = \emptyset$ for each pair R_i & $R_j, i \ne j$

Phase 2 **Duplicate elimination phase**: eliminates duplicates from each $\pi_L^*(R_i)$

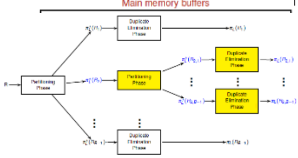
$\pi_L(R)$ = duplicate free union of $\pi_L(R_1), \pi_L(R_2) \ldots, \pi_L(R_{B-1})$

> Initialize an empty hashtable T
> For each tuple t in R:
> - Apply hash function h on $\pi_L(t)$
> - Let t be hashed to bucket B_i in T
> - if $\pi_L(t)$ is not in B_i:
>   - Insert $\pi_L(t)$ into B_i
>
> Return all entries in T

Partitioning Phase: Use **one buffer for input** & **(B-1) buffers for output**. Read R one page at a time into input buffer. For each tuple t in input tuple: project out unwanted attributes from t to form t'. Apply a hash function h on t' to distribute t' into one output buffer. Flush the output buffer to disk whenever buffer is full.



Duplicate Elimination Phase: For each partition R_i, initialize an in-memory hashtable. Read $\pi_L^*(R_i)$ one page at a time, for each tuple t read, hash t into bucket B_j with hash function $h'(h' \ne h)$. Insert t into B_j if $t \notin B_j$. Write out tuples in hashtable to results.



Partition Overflow: Partition overflow problem: Hash table for $\pi_L^*(R_i)$ is larger than available memory buffers. **Solution**: recursively apply hash-based partitioning to the overflowed partition.

Hash-Based Approach: Analysis: Approach is effective if B is large relative to |R|. Assuming that h distributes tuples in R uniformly, Each R_i has $\frac{|\pi_L^*(R)|}{B-1}$ pages. Size of hash table for each $R_i = \frac{|\pi_L^*(R)|}{B-1} \times f$. Fudge factor is a small value that increases the number of partitions. To avoid partition overflow, $B > \frac{|\pi_L^*(R)|}{B-1} \times f$ or approximately $B > \sqrt{f \times |\pi_L^*(R)|}$.

Assume there's no partition overflow,

- Cost of partitioning phase: $|R| + |\pi_L^*(R)|$ *Read |R|, output projected R**
- Cost of duplicate elimination phase: $|\pi_L^*(R)|$ *Read projected R**
- **Total cost =** $\boldsymbol{|R| + 2|\pi_L^*(R)|}$

Sort-Based vs Hash-Based: Sort-based output is sorted. Its good if there are many duplicates or if distribution of hashed values are non-uniform. If $B > \sqrt{|\pi_L^*(R)|}$,

- Number of initial sorted runs $N_0 = \left\lceil\left(\frac{|R|}{B}\right)\right\rceil \approx \sqrt{|\pi_L^*(R)|}$
- Number of merging passes $= \log_{(B-1)} N_0 \approx 1$
- Sort-based approach requires 2 passes for sorting
- Both hash-based & sort-based methods have same IO cost.

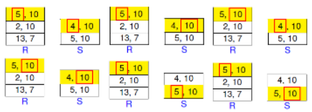
Join Algorithms: Things to consider when choosing an algorithm: **types of join predicates** (equality/inequality), **sizes of join operands, available buffer space, available access methods.** Given $R \bowtie_\theta S$, R is the **outer** relation and S is the inner relation.

Tuple-based Nested Loop Join → **I/O Cost Analysis:** $\boldsymbol{|R| + ||R|| \times |S|}$

For each tuple $r \in R$:
  For each tuple $s \in S$:
    if (r matches s): *output (r, s) to result*

Page-based Nested Loop Join → **I/O Cost Analysis:** $\boldsymbol{|R| + |R| \times |S|}$

For each page P_R of R:
  For each page P_S of S:
    For each tuple $r \in P_R$:
      For each tuple $s \in P_S$:
        if (r matches s): *output (r, s) to result*



Block Nested Loop Join: → **I/O Cost Analysis:** $|R| + \left(\left\lceil\frac{|R|}{B-2}\right\rceil \times |S|\right)$

Exploit buffer space to minimize number of I/Os. **Assume** $\boldsymbol{|R| \le |S|}$. Buffer space allocation: **Allocate one page for S, one page for output & remaining pages for R.**

while (scan of R is not done):
  read next (B-2) pages of R into buffer
  for each page P_S of S:
    read P_S info buffer
    for each tuple r of R in buffer and each tuple $s \in P_S$:
      if (r matches s): *output (r, s) to result*

Projection operating using Indexes: If there is an index whose search key contains all the wanted attributes, we can replace table scan with index scan! If the index is ordered (e.g. B+tree) whose **search key includes wanted attributes as a prefix**, we can scan data entries in order & compare adjacent data entries for duplicates. *Example: Use B+tree index on R with key (A,B) to evaluate query* $\pi_A(R)$.

Index Nested Loop Join: Consider $R(A, B) \bowtie_A S(A, C)$. Assume that theres a B+tree index on $S.A$.

Precondition: there is an index on the join attribute(s) of inner relation.

Idea: for each tuple $r \in R$: use r to probe S index to find matching tuples

Analysis: Let $R.A_i = S.B_j$ be the join condition. Assume uniform distribution, each R-tuple joins with $\left\lceil\frac{||S||}{||\pi_{B_j}(S)||}\right\rceil$ number of S-tuples.

For a format-1 B+tree index on S: **IO Cost =** $\boldsymbol{|R| + ||R|| \times J}$

$$\boldsymbol{J = \log_F\left(\left\lceil\frac{||S||}{b_d}\right\rceil\right) + \left\lceil\frac{||S||}{b_d\left|\left|\pi_{B_j}(S)\right|\right|}\right\rceil}$$

Search index's internal nodes Search index's leaf nodes