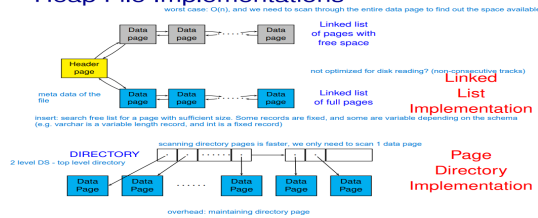


CS3223

AY22/23 Sem 2

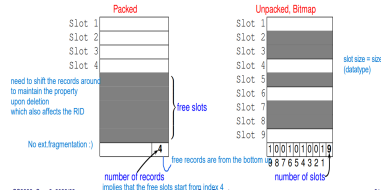
github.com/SeekSaveServe

Heap File Implementations



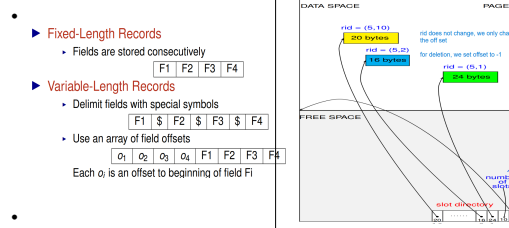
Page Formats: Fixed Length Records

- **Packed Organisation** Store records in contiguous slots
- **Unpacked Organisation** Uses a bit array to maintain free slots



Page Formats: Slotted Page (variable length record)

- Store records in slots of (record offset, record length)
- Record Offset: Offset of the record from the start of the page



L2 And L3 - Indexing

- A search key is a sequence of k attributes. If $k \geq 1$, composite key
- A search key is an unique index if it is a candidate key
- An index is stored as a file

Format of data entries

- Format 1: k^* is an actual data record with search value k
- Format 2: k^* is the form (k, rid)
- Format 3: k^* is the form (k, rid-list*)
- Note: Different formats affects the number of data entries stored in a page

Clustered Vs Unclustered

- **Clustered**: Order of data entries is the same as the order of data records. Can only be built on ordered field (e.g. primary key)
- **Unclustered**: Order of data entries does not correspond to the order of data records
- The implication is that we can read an entire clustered page with 1 I/O
- B+ Tree: Format 1 is clustered, Format 2 and 3 can be clustered if data records are sorted on the search key
- Hash: Only format 1 is clustered since hashing do not store data entries in search key order

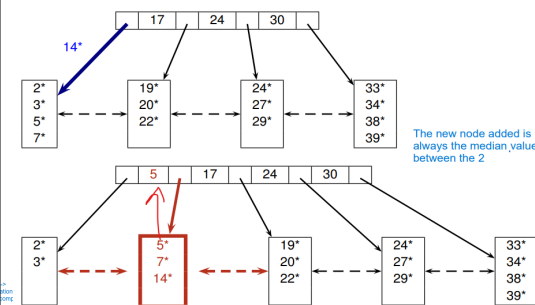
Tree Based Index - B+ Tree

- Leaf nodes are doubly linked and store Data Entries

- Internal nodes store index entries (p0, k, p1 ... pk, k, pk+1)
- Internal nodes contains m entries, $m \in [d, 2d]$ → space utilisation $\geq 50\%$
- Root contains m entries, $m \in [1, 2d]$

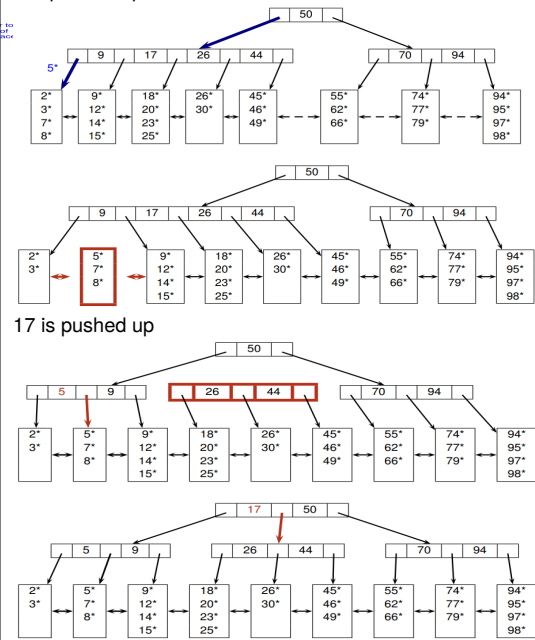
B+ Tree - Split Overflow Nodes

- Distribute d+1 entries to the new leaf node
- Create new entry index using smallest key in the new node (middle key)
- Insert new entry into parent node of overflowed node



B+ Tree - Overflow Propagation

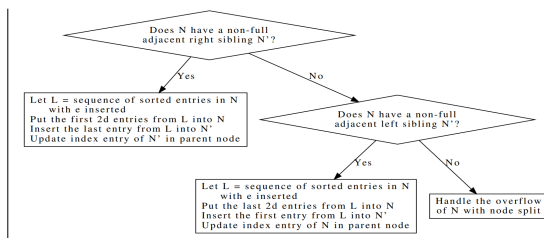
5 is pushed up



Excess middle node is pushed updated to parent node

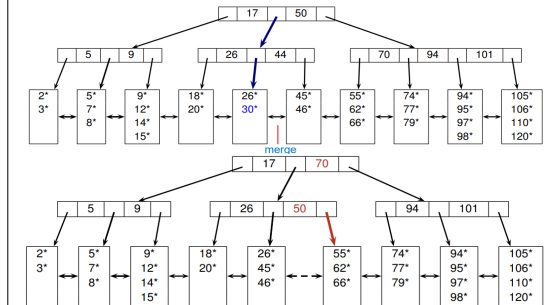
B+ Tree - Redistribution of data entries

Two nodes are siblings if they have the same parent node



B+ Tree - Underflow

- Underflow occurs when a node has less than d entries
- Underflow is resolved by redistributing entries between siblings
- An underflow node is merged if each of its adjacent siblings have exactly d entries



B+ Tree - Bulk Loading

- Initializing a B+ tree by insertion is expensive (need to traverse tree n times)
- 1. Sort all data entries by search key
- 2. Initialise B+ tree with an empty root page
- 3. Load data entries into leaf pages
- 4. In asc order, insert the index entry of each leaf page into the rightmost parent node

Hash Based Index

- Does not support range search, only equality queries

Static Hashing

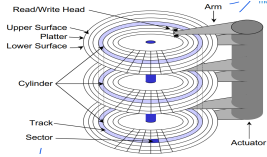
- N buckets, each bucket has 1 primary page and ≥ 0 overflow pages
- To maintain performance, we need to routinely construct bigger hash tables and redistribute data entries

Dynamic Hashing - Extendible Hashing

- No overflow pages! A bucket can be thought of as a page
- At most 2 Disk I/Os for equality search (at most 1 if directory and bucket fits in memory)
- Instead of maintaining data entries, we maintain pointers to data entries in buckets
- Instead of maintaining buckets, maintain a directory of pointers to buckets
- The directory has 2^d buckets, where d is the global depth – large overhead if hashing is uniform
- Each director entry differs by a unique d-bit address
- Two directories are corresponding iff their addresses differ only in the dth bit
- All entries with the same local depth (l) have the same last l bits in h(k)

L1 - Data Storage

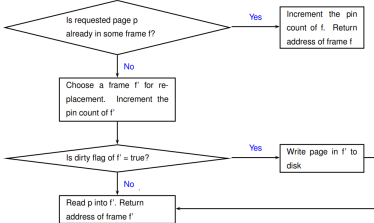
Magnetic Disks



- **Disk Access Time** Seek time + Rotational Latency + Transfer time
- **Response time** Queueing delay + Disk access time
- **Rotational Delay** $\frac{1}{2} \frac{60s}{RPM}$
- **Transfer Time** sectors on the same track * $\frac{TimePerRevolution}{SectorsPerTrack}$

Buffer Manager

- **Buffer pool** Main memory allocated for DBMS
- **pin count** is incremented upon pinning
- **dirty bit** is updated when the page is unpinned (if modified)
- Replacement is only possible if pin count == 0

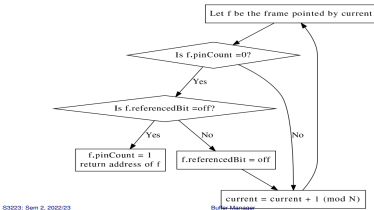


Replacement Policies LRU Policy

- Maintains a queue of pointers to frames with pin count = 0

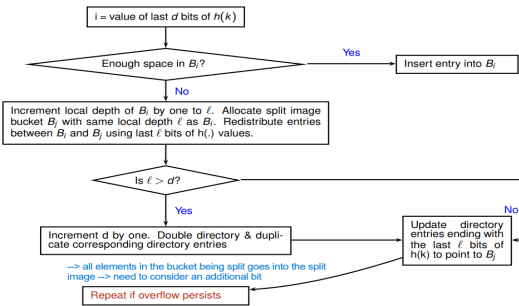
Clock Replacement Policy

N = number of frames in buffer pool



- Simplifies LRU with a second chance round robin system
- Each frame has a **reference bit** that is turned on when pin count reaches 0
- Replaces a page when referenced bit is off and pin count is 0

File Organisation



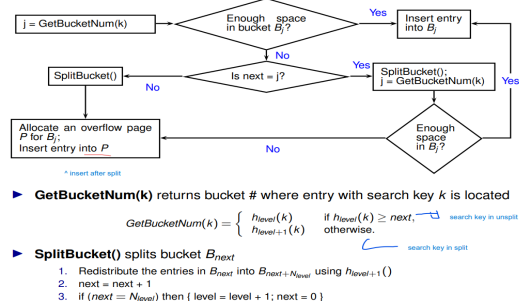
Extendible Hashing - Split, Double

- Split and doubling is checked every time a bucket is full
- Doubling only happens if local depth = global depth
- The split image has the same depth as the split bucket
- Other than the split image of the split bucket, split image of other buckets points to the same corresponding bucket
- Each bucket is pointed by $2^{(d-l)}$ directories

Extendible Hashing - Deletion

- B_i is deallocated
- l decrement by 1
- Directory Entries that point to B_i points to its corresponding bucket

Dynamic Hashing - Linear Hashing



- **GetBucketNum(k)** returns bucket # where entry with search key k is located
- $GetBucketNum(k) = \begin{cases} h_{next}(k) & \text{if } h_{next}(k) \geq next, \\ h_{next+1}(k) & \text{otherwise.} \end{cases}$ (search key in unsplit)
- **SplitBucket()** splits bucket B_{next}
 1. Redistribute the entries in B_{next} into B_{next+1} and B_{next+2} using $h_{next+1}()$
 2. $next = next + 1$
 3. If $(next == N_{next})$ then $\{level = level + 1; next = 0\}$
- One I/O for equality search (more per number of overflow pages in bucket)
- Performs worse than extendible hashing if distribution is skewed
- Does not require a directory
- Higher average space utilisation, but longer overflow chains
- Has a family of hash functions, with each having a range twice of its predecessor
- N_0 : initial number of buckets
- $N_i = 2^i N_0$: number of buckets at start of round i
- $next$: the next bucket to be split, this is incremented every time split happens
- $h_{i+1} = h(k) \bmod N_{i+1}$: hash function for round i, if the bucket $> next$ (already split)
- $h_i = h(k) \bmod N_i$: hash function for round i+1, if the bucket $> next$
- Split Criteria: By default, split when a bucket overflows
- **Linear Hashing - Deletion**
 - Essentially the inverse of insertion
 - If the last bucket is empty \rightarrow delete it, $next-$
 - If $next$ is 0, set it to $M/2 - 1$, and we can decrement level by 1 (half of buckets have been deleted if $next$ is 0)

- Merging with corresponding bucket is optional

L4: Query Evaluation - Sort, Select

Sorting - External Merge Sort

Projection, join, bulk loading etc all require sorting

- Uses 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 possibly for last run)
- **Pass i, $i \geq 1$** : Merging of sorted runs
 - Use $B-1$ buffer pages for input & one buffer page for output
 - Performs (B-1)-way merge
- **Analysis**:
 - N_0 = 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 N pages & writes N pages

External Merge Sort - Bocked I/O

- Read and write in blocks of **b** buffer pages (replace b with 1 for unoptimised)
- $\lfloor \frac{B-b}{b} \rfloor$ blocks for input, 1 block for output
- Can merge at most $\lfloor \frac{B-b}{b} \rfloor$ sorted runs in each merge pass
- $F = \lfloor \frac{B}{b} \rfloor - 1$ runs can be merged at each pass
- Num passes = $\log_F N_0$

B+ tree sort

- B+ Tree is sorted by key
- Format 1 (clustered): Sequential Scan
- Format 2/3: Retrieve data using RID for each data entry
- Unclustered implies more I/Os

Access Path refers to the different ways to retrieve tuples from a relation. It is either a **file scan** or a **index plus matching selection condition**. The more **selective** the access paths, the fewer pages are read from the disk.

- Table scan: scan all data pages
- Index scan: scan all index pages
- Table intersection: combine results from multiple index scans (union, intersect). Find RIDs of each predicate and get the intersection

Query: Selection Covering Index

- I is a covering index of $query_Q$ if I contains all attributes of Q
- No RID lookup is needed, Index-only plan
- If data is unclustered, unsorted, no index - i best way is to collect all entries and sort by RID before doing I/O

CNF Predicate

- Find RIDs of each predicate and get the intersection
- Conjuncts are in the form (R.A op c v R.a op R.b)
- CNF are conjuncts (or terms) connected by \wedge

Matching Predicates - B+ Tree

- Non-disjunctive CNF (no \vee)
- At most one non-equality comparison operator which must be on the **last attribute in the CNF**
- $(k_1 = c_1) \wedge (k_2 = c_2) \wedge \dots k_i op c_i I = (k_1, k_2 \dots k_n)$
- The order of k matters, and there cannot be missing K_i in the middle of the CNF
- Having inequality operator before equality operator makes the query to be less selective

Matching Predicates - Hasing

- No inequality operators

$$(k_1 = c_1) \wedge \dots k_i = c_n \mid I = (k_1, k_2 \dots k_n)$$

- Unlike B+ tree, **all predicates must match**

$$l = (\text{age, weight, height}), p = (\text{age} \geq 20 \wedge \text{age} \geq 18 \text{weight} = 50 \wedge \text{height} = 150 \wedge \text{level} = 3)$$

Primary Conjuncts: The subset of conjuncts in p that I matches

Primary Conjuncts: $\text{age} \geq 20 \wedge \text{age} \geq 18$

Covered Conjuncts: The subset of conjuncts in p that I covers (conjuncts that appear in I). Primary conjunct \subseteq covered conjunct

Covered Conjuncts: $\text{age} \geq 20 \wedge \text{age} \geq 18 \wedge \text{height} = 150$

Notation	Meaning
r	relational algebra expression
$ r $	number of tuples in output of r
$ r $	number of pages in output of r
b_d	number of data records that can fit on a page
b_i	number of data entries that can fit on a page
F	average fanout of B ⁺ -tree index (i.e., number of pointers to child nodes)
h	height of B ⁺ -tree index (i.e., number of levels of internal nodes)
$h = \lceil \log_F(\lceil \frac{ R }{b} \rceil) \rceil$	if format-2 index on table R
B	number of available buffer pages

Cost of B+ tree index evaluation of p

Let p '=primary conjuncts of p — p_c =covered conjuncts of p

1. Navigate internal nodes to locate the first leaf page

$$Cost_{internal} = \begin{cases} \lceil \log_F(\lceil \frac{\lceil |R| \rceil}{b_d} \rceil) \rceil |Format1| \\ \lceil \log_F(\lceil \frac{\lceil |R| \rceil}{b_i} \rceil) \rceil |Otherwise| \end{cases}$$

- 1.1 This is traversing the height of B+ tree
2. Scan leaf pages to access all qualifying data entries

$$Cost_{leaf} = \begin{cases} \lceil \frac{||\sigma_{p'}(R)||}{b_d} \rceil |Format1| \\ \lceil \frac{||\sigma_{p'}(R)||}{b_i} \rceil |Otherwise| \end{cases}$$

- 2.1 This is the cost of reading qualifying conjuncts
- 2.2 Using p_c would be wrong since covering conjuncts may be non-matching which results in more reads from the leaves
- 3 Retrieve qualified data records using RID lookups. 0 if I is covering OR format 1 index. $||\sigma_{p_c}(R)||$ otherwise

Cost of RID lookups could be reduced by first sorting the RIDs

$$\frac{||\sigma_{p_c}(R)||}{b_d} \leq Cost_{rid} \leq \min\{||\sigma_{p_c}(R)||, |R|\}$$

Cost of Hash index evaluation of p

- Format 1: cost to retrieve **data entries** is at least $\lceil \frac{||\sigma_{p'}(R)||}{b_d} \rceil$
- Format 2: cost to retrieve **data entries** is at least $\lceil \frac{||\sigma_{p'}(R)||}{b_i} \rceil$
- Format 2: Cost to retrieve **data records** is 0 if it is a covering index (all information in data entry) OR $||\sigma_{p'}(R)||$ otherwise

L5: Query Evaluation - Projection and Join

- $\pi^*(R)$ refers to projection without removing duplicates
- $\pi(R)$ involves 1. Removing unwanted attributes 2. Removing duplicates
- Sorting is better if we have many duplicates or if the distribution is nonuniform (overflow more likely for hashing partitions)

- Sorting allows results to be sorted
- If $B > \sqrt{|\pi_L^*(R)|}$, then both sorting and hashing has similar I/O costs ($\lceil \frac{|R|}{B} \rceil \rightarrow |R| + 2 * |\pi_L^*(R)|$)

Approach 1: project based on sorting

- **Naive**: Extract attributes L from records $\rightarrow \pi_L^*(R) \rightarrow$ Sort attributes \rightarrow Remove duplicates
- Cost: Cost to scan records ($|R|$) + Cost to output to temporary result ($|\pi_L^*(R)|$) \rightarrow cost to sort records ($2|\pi_L^*(R)| \log_m(N_0) + 1$) \rightarrow Cost to scan data records $|\pi_L^*(R)|$
- **Optimisation**: Create Sorted runs with attributes L only (Pass 0) \rightarrow Merge sorted runs and remove duplicates $\rightarrow \pi_L(R)$

Approach 2: project based on hashing

- Build a main-memory hash table to detect and remove duplicates. Insert to the hashtable if then entry is not already in it.
- 1. Partition R into $R_1, R_2 \dots R_{B-1}$, hash on $\pi_L(t)$ for $t \in R \leftarrow (\pi_L^*(R_i))$ does not intersect $\pi_L^*(R_j), i \neq j$
- 1.1 Use 1 buffer for input and (B-1) for output
- 1.2 Read R 1 page at a time, and hash tuples into B-1 partitions
- 1.3 Flush output buffer to disk when full
- 2. Eliminate duplicates in each partition $\pi_L^*(R_i)$
- $\pi_L(R) = \cup_i^{B-1} (\pi_L(R_i))$
- 2.1 For each partition, Initialise an in-memory hash table and insert each tuple into B_j if $t \notin B_j$

Partition overflow: Hash table $\pi_L^*(R_i)$ is larger than available memory buffers.

Solution: Recursively apply hash-based partitioning to overflowed partitions.

Analysis: Effective (no overflow) when B

$$> \frac{|\pi_L^*(R)|}{B-1} * f \approx \sqrt{f * \pi_L^*(R)}$$

If no partition overflow: (partition) $|R| + \pi_L^*(R)|$ + (duplicate elimination) $|\pi_L^*(R)|$

Index based projection: Do index scan if the wanted attributes \subseteq search key

Join $R \bowtie_{\theta} S$, where R is the outer relation and S is the inner relation

- **Tuple-based**
 - Cost: $|R| + |R| * |S|$
 - for each tuple r in R
 - for each tuple s in S
 - if (r matches s) then output (r, s) 4 to result
- **Page-based**
 - Load P_R and P_S to main memory
 - Cost: $|R| + |R| * |S|$
 - for each page P_R in R
 - for each page P_S in S
 - for each tuple $r \in P_R$
 - for each tuple $s \in P_S$
 - if (r matches s) then output (r, s) 4 to result
- **Block nested-loop**
 - Allocate 1 page for S, 1 for output, B-2 for R
 - $|R| \leq |S|$
 - Cost: $|R| + (\lceil \frac{|R|}{B-2} \rceil * |S|)$
 - $|R| \leq |S|$
 - while Scanning R
 - read next (B-2) pages of R to buffer
 - for P_S in S
 - read P_S into Buffer
 - for $r \in buffer \wedge s \in P_S$
 - if (r matches s) then output (r, s) 4 to result
 - Without materialisation: $\lceil \frac{|R|}{B-2} \rceil * |T|$
- **Index Nested Loop Join**
 - There is an index on the join attributes of S
 - Uniform distribution: r joins $\lceil \frac{|S|}{|\pi_{B_j}(S)|} \rceil$ tuples in S
 - format 1
 - B+Tree: $|R| + |R| * J$

- $J = \log_F(\lceil \frac{|S|}{b_d} \rceil)$ (tree traversal)+ $\lceil \frac{|S|}{b_d |\pi_{B_j}(S)|} \rceil$ (search leaf nodes)
- for $r \in R$
- use r to probe S's index to find matching tuples
- **Sort-Merge Join**
 - Sort R and S on join attributes and merge
 - Cost: $2|\pi_L^*(R)|(\log_{B-1}(\sqrt{\frac{|R|}{B}})+1) + 2|\pi_L^*(S)|(\log_{B-1}(\frac{|S|}{B})+1) + |\pi_L^*(R)| + |\pi_L^*(S)|$
 - merging cost is $|R| + |R| * |S|$ if each tuple of R requires a full scan of S
 - Optimisation: $B \geq N(R, i) + N(S, i) + 1 \geq \sqrt{|R| + |S|}$
 - We can choose which relation to partition again if this is not met
 - Cost: cost of getting R, S + k(write out $(|R| + |S|)$) + m(merge $(|R| + |S|)$)
 - If sorted on join column: $|R| + |S|$
- **Grace Hash Join**
 - Partition into $B - 1$ partitions
 - If no partition overflow ($B > \sqrt{f * |S|}$):
 - k(Cost to partition R, S) + Cost of probe phase
 - Partition cost = cost of getting R + cost to write partitions($|R|$)
 - Probe cost = $|R| + |S|$

L6: Query Optimisation

1. Search space: Queries considered

- **Search Place** Queries being considered
- **Linear** if at least one operand for each join is a base relation, bushy otherwise
- **Left-deep** if every right join operand is a base relation
- **Right-deep** if every left join operand is a base relation

2. Plan enumeration - for joins between 2 tables

Input: A SPJ query q on relations R_1, R_2, \dots, R_n
Output: An optimal query plan for q

```

01.  for i = 1 to n do
02.      optPlan( $\{R_i\}$ ) = best access plan for  $R_i$ 
03.  for i = 2 to n do
04.      for each  $S \subseteq \{R_1, \dots, R_n\}, |S| = i$  do
05.          bestPlan = dummy plan with cost(bestPlan) =  $\infty$ 
06.          for each  $S_j, S_k, |S_j| \in [1, i), S = S_j \cup S_k$  do
07.               $p$  = best way to join optPlan( $S_j$ ) and optPlan( $S_k$ )
08.              if  $cost(p) \leq cost(bestplan)$  then
09.                  bestPlan =  $p$ 
10.          optPlan( $S$ ) = bestPlan
11.  return optPlan( $\{R_1, \dots, R_n\}$ )

```

- Decide on the plan for the 2 operands
- Decide on the plan to join: Block nested loop, sort merge join, grace hash join

3. Cost Model

- **Uniformity:** uniform distribution of all values
- **Independence:** Independent distribution of values in different attributes
- **Inclusion:** for $R \bowtie S, if |\pi_A(R)| \leq |\pi_B(S)|$ then $\pi_A(R) \subseteq \pi_B(S)$
- **Plans-no join, 1 table**
 - **Table scan** Scan the entire table. Cost: $|R|$
 - **Index scan** Scan the index. Cost: $2 + |\text{leaf pages satisfying the predicate}| + |\text{entries satisfying predicate}|$ (unclustered)
 - **Index intersection with $I_a I_b$** Cost to partition predicate1(R) + Cost to partition predicate2(R) + cost to intersect partitions 1,2 + cost to RID lookup
 - cost to partition: Scan index for matching pages + cost to write partitions from matching entries

Histogram

- **Equiwidth** Each bucket has equal number of values
- Estimate: $\frac{1}{|bucket|} * ||bucket||$
- **Equidepth** Each bucket has equal number of tuples
- Sub-ranges can overlap, tuples of the same value can be in 2 adjacent buckets
- $\frac{1}{|bucket_A|} * ||bucket_A|| + \frac{1}{|bucket_B|} * ||bucket_B|| + \dots$
- **MCV** Separately track the top-k MCV and exclude them from the bucket

Size of query

- **Join** $||R|| * ||S|| * \frac{1}{max(|\pi_b(R)|, |\pi_b(S)|)}$
- **Select - OR** $(1 - (p(a = x) * p(b = y))) * ||R||$
- **Select - AND** $p(a = x) * p(b = y) * ||R||$

L7: Transaction Management

View Equivalent

- If T_i reads A from T_j in S, then T_i must also read A from T_j in S'
- For each data object A, the Xact (if any) that performs the final write on A in S must also perform the final write on A in S'

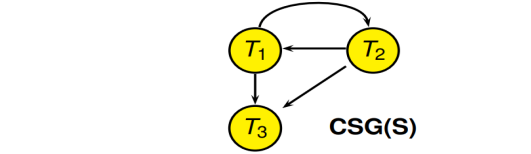
Conflicting actions - WW, WR

- **Dirty Read** T2 read uncommitted write from T1
- **Unrepeatable Read** T2 updates an object that T1 has previously read and T2 commits while T1 is still in progress \rightarrow T1 can get a different value from read
- **Lost Update** T2 overwrites the value of an object that has been modified by T1 while T1 is still in progress
- View serializable prevents These
- **Blind write** $R_1(X), W_2(Y), W_1(X)$ Blind write: $W_2(Y)$
- **Conflict Serializable** Conflict equivalent to serial schedule, view serializable and not blind write
- **Non Conflict Serializable** find conflicting action pairs($R1(x) W1(x)$), ($R2(x) W1(x)$)

Conflict Serializability Graph

- V contains a node for each committed Xact in S
- E contains (T_i, T_j) if an action precedes and conflicts with one of T_j 's actions

$R_1(A), W_2(A), Commit_2, W_1(A), Commit_1, W_3(A), Commit_3$



Schedules

- **Cascading aborts** T_i read from $T_j \rightarrow T_j$ aborts $\rightarrow T_i$ aborts
- **Recoverable** $\forall T \in S$ T2 must commit after T1 if T2 reads from T1
- **Cascadeless** Whenever T_i reads from T_j in S, Commit must precede this action
- Theorem 4: Cascadeless \rightarrow Recoverable (not iff)
- **Strict** to use before-images, $\forall W_i(O) \in S, O$ is not read or written by another Xact until Ti either aborts or commits
- Theorem 5: Strict \rightarrow Cascadless (not iff)

2PL

- To read an object O, a Xact must hold a S-lock or X-lock on O
- To write to an object O, a Xact must hold a X-lock on O
- Once a Xact releases a lock, the Xact can't request any more locks
- Theorem 1: 2PL is conflict serializable

Strict 2PL

- To read an object O, a Xact must hold a S-lock or X-lock on O
- To write to an object O, a Xact must hold a X-lock on O
- A Xact must hold on to locks until Xact commits or aborts
- Theorem 2: Strict 2PL is strict and conflict serializable
- Strict 2PL prevents cascading rollback and deadlock and ensures recoverability

Detect deadlocks

- Waits-for graph (WFG) \rightarrow Deadlock is detected if WFG has a cycle
- Breaks a deadlock by aborting a Xact in cycle
- **Deadlock Prevention**
 - Each Xact is assigned a timestamp when it starts
 - Assume older (smaller time stamp) Xacts have higher priority than younger Xacts
 - Suppose T_i requests for a lock that conflicts with a lock held by T_j
 - Two possible deadlock prevention policies:
 - **Wait-die policy:** lower-priority Xacts never wait for higher-priority Xacts
 - **Wound-wait policy:** higher-priority Xacts never wait for lower-priority Xacts

Prevention Policy	T_i has higher priority	T_i has lower priority
Wait-die	T_i waits for T_j	T_i aborts
Wound-wait	T_j aborts	T_i waits for T_j

L8: MVCC

Multi Version Serializable Schedle (MVSS)

- **multiversion view equivalent** if S and S' have the same set of read-from relationships
- i.e. $R_i(x_j)$ occurs in S iff $R_i(x_j)$ occurs in S'
- **Monoversion Schedule** each read action returns the most recently created object version
- **MVSS** if there exists a serial Monoversion schedule that is multiversion view equivalent to S
- Note that a MVSS is not necessarily conflict serializable schedule if it is not a valid monoversion schedule
- E.g. $W1(x1), R2(x0), R2(y0), W1(y1), C1, C2$ is MVSS with (T2, T1) but contains conflicting actions $W1(x1)$ and $R2(X0)$

Snapshot Isolation (SI)

- Each Xact has a snapshot of the database at the start of the Xact and sees only versions from that snapshot and **its own writes**
- **FUW** T needs to acquire X-lock on O (if not - wait), and if O has been updated by a concurrent T' then T aborts
- **FCW** (no locks) before committing T checks if O has been updated, abort if it has been updated
- Write-skew anomaly, not MVSS: $R_1(X_0), R_2(X_0), R_1(Y_1), R_2(Y_2), W_1(X_1), Commit1, W_2(Y_2), Commit2$

Transaction Dependencies

- **WW** from T1 to T2: T1 writes some version of X and T2 writes the immediate successor
- **WR** from T1 to T2: T1 writes some version of X which is read by T2
- **RW** from T1 to T2: T1 reads some version of X and T2 writes the immediate successor
- **DSG** V = xacts, E = Dependencies, use $-->$ for concurrent transactions and \rightarrow for non-concurrent