|  |  |  |  |
| --- | --- | --- | --- |
| DBMS | Stores relations & system catalog/data dictionary (contains metadata abt relations, e.g. schemas, constraints, triggers, view definitions, indexes, statistical info abt relations for use by query optimizer) & log files (for data recovery)  Memory Hierarchy: 1) Primary memory: registers, static RAM (caches), dynamic RAM  2) Secondary memory: magnetic disks (HDD), solid-state disks/drive (SSD). 3) Tertiary memory: optical disks, tapes  Tradeoffs: capcaity, cost, access speed, volatile vs non-volatile  DBMS stores data on non-volatile disk for persistence; processes data in main memory (RAM)  Disk access operations: 1) read: transfer data from disk to RAM. 2) write: transfer data from RAM to disk | | |
| HDD | Diagram of a diagram of a structure  Description automatically generatedMagnetic Hard-Disk Drive (HDD)  Disk access time = seek time + rotational delay + transfer time (command processing time considered negligible)  - Command processing time: interpreting access command by disk controller  - Seek time: moving arms to position disk head on track (≈ 5-6ms)  - Rotational delay: waiting for block to rotate under head  [rotation speed measured in rotations per minute; RPM  average rotational delay = time for 1/2 revolution]  - Transfer time: actually moving data to/from disk surface  [n = num of requested sector on same track. Avg sector transfer time ≈ 100-200 Transfer time = ]  Response time for disk access = queuing delay + access time  Sequential vs random I/O, Sequential read faster than random reads | | |
| SSD | Built w NAND flash memory w/o any moving parts  Random I/O: 100X faster than HDD Sequential I/O: slightly faster than HDD | Pros: Lower power consumption. Cons: - updates to a page requires erasure of multiple pages (≈ 5ms) before overwriting page  - Limited num of times a page can be erased (≈ 105 - 106) | |
| Storage Manager Components | Data is stored & Retrieved in units called disk blocks or pages. Each block = seq of 1 or more contiguous sectors  1) Files & access methods layer (aka file layer): deals w organization and retrieval of data  2) Buffer Manager: controls reading/writing of disk pages. 3) Disk Space Manager: keeps track of pages used by file layer | | |
| Buffer Manager | Buffer pool = main memory allocated for DBMS. Buffer pool is partitioned into block-sized pages called frames  Clients of buffer pool can - request for a disk page to be fetched into buffer pool OR - release a disk page in buffer pool  Page in buffer is dirty = has been modified & not updated on disk  For each frame in buffer pool, keep track of: 1) pin count (num of. clients using page; initialized to 0). 2) dirty flag (whether page is dirty)  Pinning the requested page in its frame = incrementing pin count. Unpinning page = decrementing pin count  When unpinning a page, its dirty flag should be updated to true if the page is dirty. A page in buffer can be replaced only if pin count is 0  Replacement Policy: decide which unpinned page to replace: Random; FIFO; Most Recently used (MRU), Least Recently Used (LRU), Clock  Buffer manager coordinates with transaction manager to ensure data correctness and recoverability | | |
| - Requesting for page  + Replacement policy | if <is requested page p already in some frame f>:  Increment the pin count of f; Return address of frame f  else:  if <have free frame f'>:  Read p into f'. Increment pin count of f' Return address of frame f'  else: { Choose a frame f' for replacement. Increment the pin count of f'  if <is dirty flag of f' true>: {Write page in f' to disk}  Read p into f'. Return address of frame f' }  Clock is a variant of LRU: - current var (points to some buffer frame). - Each frame has a referenced bit (turns on when its pin count becomes 0)  Clock Unpin frame f: Change f.pinCount = 0. Change f.referencedBit = on | | Clock Replacement Policy  while true {  Let f be frame pointed by current  if <is f.pinCount = 0>:  if <is f.referencedBit = off>:  f.pinCount = 1; return address of f  else:  f.referencedBit = off  current = current + 1 (mod N)  }  N = num of frames in buffer pool |
| LRU | Use a queue of pointers to frames w pin count = 0. Frame is added to end of queue when it becomes a candidate for replacement (when pin\_count goes to 0). Page chosen for replacement is the one in the frame at the head of the queue | | |
| Files | File abstraction: - Each relation is a file of records. - Each record has a unique record identifier (RID/TID)  Common file ops: - create/delete file, - insert a record, - get/delete a record w a given RID, - scan all records  File organization: mtd of arranging data records in a file that is stored on disk  1) Heap file: unordered file. 2) Sorted file: records ordered on some search key. 3) Hashed file: records located in blocks via a hash fn | | |
| Heap File Implemen-tation | A diagram of a data page  Description automatically generatedLinked list implementation  Page Directory Implementation | | |
| Page Formats | A diagram of a space  Description automatically generatedOrganizing records within a page. RID = (page id, slot number) Fixed-Length Records: - Packed organization (store records in contiguous slots). - Unpacked organization: Uses a bit array to maintain free slots  A screenshot of a graph  Description automatically generatedA graph of a number of records  Description automatically generated  Variable-Length Records: Slotted page organization | | |
| Records Formats | Organize fields within a record  Fixed-Length Records: Firelds are stored consecutively  Variable-Length Records: Delimit fields w special symbols.  OR Use an array of field offsets. Each oI is an offset to beginning of field Fi | | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index | Data structure to speed up retrieval of data records based on some search key. Search key = seq of k data attributes, k ≥ 1  Composite search key = search key with k > 1. Index is a unique index if its search key is a candidate key; otherwise, it is a non-unique index  Index is stored as a file. Records in an index file = data entries. Index file: (attr, RID1) -> (attr, RID2) -> ....  So when searching based on index, can just scan index file, filter based on index attr, just RID and find the actual data in a data file | | | | | | | |
| Index Types | Tree-based index: Based on sorting of search key values  Hash-based index: Data entries accessed using hashing fn | | | | Need to consider search performance (for equality search or range search)  Storage overhead. Update performance | | | |
| B+-tree Index | A diagram of a diagram  Description automatically generatedEach node either a leaf node or internal node  Top most internal node is root node at level 0  Height = num of levels of internal nodes (e.g. 2)  Leaf nodes are at level h where h = height of tree  Nodes at same level = sibling nodes if they have same parent node  - Leaf nodes store sorted data entries: k\* denote a data entry of form (k, RID) where k = search key value  Leaf nodes are doubly-linked  - Internal nodes store index entries of the form (p0, k1, p1, k2, p2, …, pn) where k1 < … < kn. pi = disk page addr (root node of an index subtree Ti)  Each (ki, pi) is an index entry; ki serves as a separator btw the node contents pointed to by pi-1 & pi. Note k\* ≥ k except for leftmost leaf node | | | | | | | |
| Properties | | | Dynamic index structure; adapts to data updates gracefully. Height-balanced index structure  Order of index tree, d : 1. Controls space utilization of index nodes. In above e.g. order of tree = 2  2. Each non-root node contains m entries, where m [d, 2d]. 3. Root node contains m entries, where m [1, 2d] | | | | | |
| Search | Equality search: At each internal node N, find largest key ki in N s.t. k ≥ ki. If ki exists, then search subtree at pi  Else, search subtree at p0 | | | | Range search: Same as equality search. But at leaf nodes, traverse to other leaf nodes to continue search if needed. Can go left or right | | | |
| Format of data entries | Format 1) k\* is an actual data record (with search key value k)  Format 2) k\* is of form (k, rid), where rid = record identifier of a data record w search key value k  Format 3) k\* is of form (k, rid-list), where rid-list = list of record identifiers of data records w search key value k (i.e. index not unique) | | | | | | | |
| Insert | def InsertInternalNode(N, e):  If N is null:  Create a new root node N = (p0) where p0 = addr of current root node  Insert e into N  elif N is not full/overflowed: Insert e into N  else:  Allocate new leaf node N'. Let L = seq of sorted entries in N w e inserted  Put first d entries from L into N. Put last d + 1 entries from L into N'  Update sibling pointers in N & N'  Let e' = (k', p') be a new index entry where k' = min key value in N' & p' = addr of N'  InsertInternalNode(P, e') where P = parent node of N | | | | | | Insert with node split  Start w index page w no index entry  e = new data entry to be inserted into index  N = leaf node where e is to be inserted into  d = order of tree  Overflowed node = node is full (i.e. with 2d entries) and new entry is to be inserted into it  Parent node of a root node is null  When splitting an internal node, the middle key is pushed up to parent node | |
| Redistribute of date entries (in leaf nodes) | | | | def InsertInternalNode(N, e):  if N have a non-full adjacent right sibling N':  Let L = seq of serted entries in N w e inserted. Put first 2d entries from L into N  Insert last entry from L into N'. Update index entry of N' in parent node  elif N have a non-full adjacent left sibling N':  Let L = seq of serted entries in N w e inserted. Put last 2d entries from L into N  Insert first entry from L into N'. Update index entry of N in parent node  else: { Handle overflow of N with node split } | | | Sometimes can avoid node split by distributing entries from overflowed node to a non-full adjacent sibling node | |
| Delete | N = non-root leaf node that underflows after deletion of data entry. Assume redistribution is attempted whenever possible  P = parent node of N. Nl = adjacent left sibling of N. Nr = adjacent right sibling of N.  Underflowed node = non-root node w d entries, and an existing entry is to be deleted from that node | | | | | | | |
| if Nr has > d entries:  Move 1st entry from Nr to N. Update index entry of Nr in P  elif Nl has > d entries:  Move last entry from Nl to N. Update index entry of N in P  elif Nr exist:  Move all entries from Nr to N. Deallocate Nr  Update pointers btw N & Nr's right adjacent node.  Delete index entry of Nr in P  If P becomes underflowed non-root node:  UnderflowInternalNode(P)  else:  Move all entries from N to Nl. Deallocate N  Update right pointer in Nl to null. Delete index entry of N in P  If P becomes underflowed non-root node:  UnderflowInternalNode(P)  To merge 2 adjacent internal nodes, pull down the separating key from parent node to from merged node | | | | | UndeflowInternalNode(N)  if Nr has > d entries:  Let e' = (k', p') be index entry of Nr in P. Nr = (p0, k1, p1, …)  Insert (k', p0) as last entry into N  Update e' in P to (k1, p'). Remove (p0, k1) from Nr  elif Nl has > d entries:  Let e' = (k', p') be index entry of N in P. Nl = (p0, …, kn, pn)  Insert (pn, k') into N to precede its 1st entry  Update e' in P to (kn, p'). Remove (kn, pn) from Nl  elif Nr exist:  Let e' = (k', p') be index entry of Nr in P  Append k' & all entries from Nr to N. Deallocate Nr & delete e' from P  If P becomes underflowed non-root node: UnderflowInternalNode(P)  Else if P becomes empty: Deallocate P & N becomes root node  else:  Let e' = (k', p') be index entry of N in P  Append k' & all entries from N to Nl. Deallocate N & delete e' from P  If P becomes underflowed non-root node: UnderflowInternalNode(P)  Else if P becomes empty: Deallocate P & Nl becomes root node | | |
| Bulk loading a B+-tree | | 1. Sort data entries to be inserted by search key. 2. Load leaf pages of B+-tree w sorted entries  3. Initialize B+-tree w 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 (fill left most node w max entries) | | | | | | Efficient construction algo  Leaf pages are allocated sequentially |
| Index types | Clustered index = order of index data entries is same as or 'close to' order of data records; otherwise unclustered index  Index using Format 1 for its data entries is a clustered index | | | | Dense index = index w index record for every search key value in data; otherwise sparse index  Unclustered index must be dense. Clustered can be sparse or dense | | | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hashed-Based Indexing | | | Used for equality queries but not for range queries.  Hashing techniques: Static hashing, Dynamic hashing (Linear hashing, Extendible hashing, ...) | | | | | |
| Static Hashing | Data is stored in N buckets, B0, …, BN-1. N is fixed at creation time. Hashing fn h(k) is used to identiy bucket to store a record.  Record w search key k inserted into bucket Bi, where i = h(k) mod N. h(k) maps search key value into a bit string  Each bucket consists of 1 primary data page & a chain of 0 or more overflow data pages (think linked list) | | | | | | | |
| Linear Hashing | Hash file grows linearly by systematic splitting of buckets. Each bucket consists of 1 primary data page & a chain of 0 or more overflow pages.  Insertion into bucket Bi overflows if all pages in Bi (i.e. primary & overflow pages) are full. Assume initial file size of N0 buckets, B0, …,  File grows linearly by splitting buckets in rounds: At each round, buckets are split sequentially (Bi split before Bi+1)  To split a bucket Bi: Add new bucket Bj (aka split image of Bi). Redistribute entries in Bi btw Biand Bj  File size incr by 1 bucket after each bucket split. At end of 1 round of splitting (i.e. every bucket at start of round splitted), file size is doubled  Let Ni = file size at beginning of round i. Ni = 2iN0, i ≥ 0.  At end of round i, Ni new buckets are added: . In round i, split image of Bj is for j [0, Ni - 1] | | | | | | | |
| Dynamic Hashing | Recall v\* denotes data entry e w h(e.key) = v.  Keep track of which bucket to be split next using var *next*  Initialize *next* to 0 at start of each round | | | | | Each round uses 2 hash fn: fn hi and hi+1 for round i. hi(k) = h(k) mod Ni.  Bx is bucket for search key k if Bx had not been split, where x = hi(k)  By = bucket for search key k if Bx had been split, where y = hi+1(k) | | |
| Redistri-buting Entries E.g. | A screenshot of a math test  Description automatically generatedA screenshot of a test  Description automatically generatedLet N0 = 2m for m {0, 1, …}. Ni = 2iN0 = 2m+i  hi(k) = h(k) mod Ni = value of last (m+i+) bits of h(k) In round i:  - before splitting, all entries e in Bj have hi(e.key) = j (i.e. same last (m+i) bits)  - After splitting, e remains in Bj iff the last (m+i+1) bit of h(e.key) is 0  h(e.key):  32 = (100 0 002). 44 = (101 1 00). 36 = (100 1 00)  9 = (0010 01). 5 = (0001 01)  14 = (0011 10). 18 = (0100 10). 30 = (0111 10)  31 = (0111 11). 35 = (1000 11) | | | | | | | |
| Splitting bucket | Split whenever some bucket overflows OR whenever space utilization of file > some threshold OR ...  Overflow pages are needed for an overflowed bucket if it is not the next bucket to be split  Assume bucket split is triggered whenever some bucket overflows due to insertion. Use level to denote splitting round number | | | | | | | |
| Linear Hashing Insertion | # returns bucket num where entry w search key k is located  GetBucketNum(k) =  def SplitBucket():  Redistribute entries in Bnext into using hlevel+1()  next += 1  if (next == Nlevel) then {level = level + 1; next = 0} | | | | | | j = GetBucketNum(k)  if (enough space in bucket Bj): {Insert entry into Bj}  elif (Is next == j): {SplitBucket(); j = GetBucketNum(k)  if (Enough space in Bj): {Insert entry into Bj} }  else: {SplitBucket() }  Allocate an overflow page P for Bj  Insert entry into P | |
| Linear Hashing Deletion | Locate bucket and delete entry  if (next > 0): {next -= 1} # If last bucket, becomes empty, it can be removed. Case 1  elif (next == 0) and (level > 0): { Update next to point to last bucket in previous level ; level -= 1 } # Case 2 | | | | | | | |
| Linear Hashing Summary | | | | 1 disk I/O unless bucket has overflow pages. On avg 1.2 disk I/O for uniform data dist. Worst case: I/O cost is linear in num of data entries. Poor space utilization w skewed data dist | | | | |
| Extendible Hashing | | Similar to linear hashing: num of buckets grow dynamically. Uses some num of least significant bits of h(k) as bucket addr for search key k  A diagram of a number  Description automatically generatedBut add new bucket (as split image) whenever an existing bucket overflows. Then no overflow pages (except when num of collisions exceed page capacity). - 2 data entries collide if they have the same h(k) value  Uses a directory of pointers to buckets. Directory expands dynamically as buckets overflow  Directory has 2d entries where d = global depth of hash file. Each directory entry has a unique d-bit addr bdbd-1…b2b1  2 directory entries correspond if their addresses differ only in the dth bit (i.e. bd); such entries aka corresponding entries  Each bucket maintains a local depth, . All entries in a bucket w local depth have same last bits in h(k) | | | | | | |
| Extendible Hashing Insertion | | i = value of last d bits of h(k)  while (not enough space in Bi): {  Increment local depth of Bi by 1 to . Allocate split image bucket Bj w same local depth as Bi.  Redistribute entries btw Bi and Bj using last bits of h(k) values  if ( > d): {Increment d by one. Double directory & duplicate corresponding entries}  Update directory entries ending w last bit of h(k) to point to Bj  i = value of last d bits of h(k)  }  Insert entry into Bi | | | | | | Split bucket & its image have same local depth  When directory is doubled, each new directory entry (except for entry for split image) points to same bucket as its corresponding entry  Num of directory entries pointing to a bucket = 2d-l |
| Extendible Hashing Deletion | | Locate bucket Bi containing entry & delete entry  If Bi becomes empty, Bi can be merged w bucket Bj where both buckets have same local depth and i & j differs only in the bit  - Bi is deallocated. - Bj's local depth is decremented by 1. - Directory entries that point to Bi are updated to point to Bj  More generally, Bi & Bj (w same local depth and i & j differs only in the bit) can be merged if their entries can fit within a bucket  If each pair of corresponding entries point to the same bucket, directoy can be halved (i.e. d -= 1) | | | | | | |
| Extendible Hashing Summary | | | | | At most 2 disk I/Os for equality selection: At most 1 disk I/O if directory fits in main memory  Handling collisions: Overflow pages are needed when num of collisions exceeds page capacity | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Query Processing | SQL Query Query Parser & Translator Relational Algebra (RA) Expr Query Optimizer Query Plan Query Evaluator Query Result  Query Plans: 1) Logical plan - high level overview. Expressed in Relational algebra. Include high level details on order of operation  2) Physical plan - low level. Include details on order of op, choice of algo (sort-merge or hash join, hash-based projection) and access path | | | | |
| External Merge Sort | A white rectangular sign with black numbers  Description automatically generatedE.g. Sort 11-page data R using 3 memory pages. Each page can contain 2 data records  1) Read first 3 data pages (6 data records) into allocated/main memory. 2) Sort the data records in main memory  3) Write the sorted data records to a file on disk (aka sorted run): R1 4) Repeat process for next 3, 3, 2 data pages: R2, R3, R4  A green and black rectangular sign with black numbers and blue text  Description automatically generatedA close-up of a memory  Description automatically generatedA screenshot of a memory  Description automatically generated5) Merge R1& R2 to create a longer sorted run: Merge 1 page of R1 & R2 at a time. Use 2 memory pages for input records. Use 1 memory page for output records (Read 1 page of R1 & R2 into main memory -> Put smallest records in bottom page of main memory -> When bottom page full, write to disk (R5) -> Continue sorting -> When all records read in are used, read next page of R1 or R2 -> Eventually get final R5)  6) Merge R3 & R4 to create R6. 7) Finally merge R5 & R6 to create final sorted run | | | | |
| File size of N pages. Use B num of buffer pages.  Pass 0: Creation of sorted runs: - read in and sort B pages at a time. - Num of sorted runs created = . Size of each sorted run = B pages (except possibly for last run)  Pass i, i ≥ 1: Merging of sorted runs: - Use B-1 buffer pages for input & 1 buffer page for output. - Performs (B-1) way merge  Analysis: N0 = num of sorted runs created in pass 0 = . Num of passes = (+1 for pass 0)  Total num of I/O = (each pass reads N pages & writes N pages) | | | | |
| Optimiza-tion w Blocked I/O | Read and write in units of buffer blocks of b pages. Reduce num of I/O. If b = 1, then same as default  Given an allocation of B buffer pages for sorting: - Allocate 1 block (b pages) for output. - Remainder space can accomodate blocks for input. - Thus can merge at most sorted runs in each merge pass  Analysis: b = num of pages of each buffer block. N0 = num of initial sorted runs =  F = num of runs that can be merged at each merged pass = . Num of passes = | | | | |
| Sorting using B+-trees | When table to be sorted has a B+-tree index on sorting attributes  1) Format 1: Sequentially scan leaf pages of B+-tree  2) Format 2 or 3: Sequentially scan leaf pages of B+-tree. For each leaf page visited, retrieve data records using RIDs | | | | |
| Access Path | Access path = way of accessing data records/entries: 1) Table scan = scan all data pages. 2) Index scan = scan index pages  3) Index intersection = combine results from multiple index scans (e.g. intersect, union)  Selectivity of an access path = num of index & data pages retrieved to access data records/entries  Most selective path (i.e. w smallest selectivity) retrieves the fewest pages | | | | |
| B+-tree: Include Columns | E.g. B+-tree index on Student name: *CREATE INDEX stu\_name\_index ON Student (name) INCLUDE (major, year)*  Useful for query like: *SELECT major FROM Student WHERE name = 'Lucy'*  An index I is a covering index for a query Q if all the attrs referenced in Q are part of the key or include column(s) of I: - Q can be evaluated using I w/o any RID lookup - Such an evaluation plan is known as index-only plan  If index not a covering index, then have to do Index Scan + RID lookups | | | | |
| B+-tree: Index Intersection | | | E.g. *SELECT height, weight FROM Student WHERE height BETWEEN 164 and 170 AND weight BETWEEN 50 AND 59*  If I1 = (height), I2 = (weight). Then get relevant RIDs from each index and find their intersection | | |
| Conjunctive Normal Form  (CNF) Predicates | | A term is of the form *R.A op c* or *R.Ai op R.Aj*  A conjunct consists of 1 or more terms connected by  A conjunct that contains is disjunctive (or contains a disjunction)  A CNF predicate consists of 1 or more conjuncts connected by | | | |
| B+-tree: Matching Predicates | B+-tree index I = (K1, K2, …, Kn). Non-disjunctive CNF predicate p  I matches p if p is of the form:  where (K1, …, Ki) is a prefix of the key of I and there is at most 1 non-equality comparison operator which must be on the last attr of the prefix (i.e. Ki) | | | | |
| Hash Index: Matching predicates | | | | Hash index I = (K1, K2, …, Kn). Non-disjunctive CNF predicate p  I matches p if p is of the form: (K1 = c1) (K2 = c2) … (Kn = cn) | |
| Primary Conjuncts | The subset of conjuncts in selection predicate p that an index I matches are called primary conjuncts  E.g. B+-tree index I = (age, weight, height). Predicate p = (age ≥ 18) (age ≤ 20) (weight = 65) (level = 3)  Primary conjuncts: (age ≥ 18) (age ≤ 20) (BETWEEN counted as 1 op?). Non-primary conjuncts: (weight = 65) (level = 3) | | | | |
| Covered Conjuncts | Given a predicate p and an index I, a conjunct C in p is a covered conjunct if all the attrs in C appear in the key or include column(s) of I  Primary conjuncts Covered conjuncts  E.g. B+-tree index I = (age, weight, height). Predicate p = (age ≥ 18) (age ≤ 20) (height = 180) (level = 3)  Covered conjuncts: (age ≥ 18) (age ≤ 20) (height = 180). If I has an include column (level), then covered conjuncts becomes the entire p | | | | |
| Notation | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | r |  |  | bd | bi | F | h | B | | RA expr | Num of tuples in output of r | Num of pages in output of r | Num of data records that can fit on a page | Num of data entries that can fit on a page | average fanout of B+-tree index (num of pointers to child nodes) | height of B+-tree index  h = if format-2 index on table R | Num of available buffer pages | | | | | |
| Cost of B+-tree index evaluation of p | | | Let p' = primary conjuncts of p, pc = covered conjuncts of p  1. Navigate internal nodes to locate first leaf page =  2. Scan leaf pages to access all qualifying data entries =  3. Retrieve qualified data records via RID lookups =  Cost of RID lookups could be reduced by first sorting RIDs: | | |
| Cost of hash index evaluation of p | | | Let p' = primary conjuncts of p  For format-1 index: cost to retrieve data records = at least | | For format-2 index: - cost to retrieve data entries: at least  - cost to retrieve data records = |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Projection | | = project cols given by list L from relation R. = same as but preserves duplicates  Projection involves: 1) remove unwanted attributes. 2) Eliminate any duplicate tuples produced. Can project based on sorting OR hashing | | | | | | | |
| Sort-based Approach | | R -> 1) Extract attrs L from records -> -> 2) Sort records using attrs L as sort key -> Sorted -> 3) Remove duplicates ->  Cost analysis: Step 1) Cost to scan records = |R|. Cost to output temporary result = ||  Step 2) Cost to sort records = 2||(logm(N0) + 1), where N0 = num of initial sorted runs, m = merge factor  Step 3) Cost to scan records = || | | | | | | | |
| Optimized Sort-based Approach: Split step 2 into 2a) Create sorted runs + 2b) Merge sorted runs  R -> 1) + 2a) Create sorted runs w attrs L -> 2b) + 3) Merge sorted runs & remove duplicates -> | | | | | | | |
| Hash based approach | | Consider . Build a main-memory hash table T to detect & remove duplicates:  Consists of 2 phases:  1) Partitioning phase: partitions R into R1, R2, …, RB-1  - Hash of for each tuple t R. - R = R1 R2 … RB-1  - for each pair Ri & Rj, i ≠ j  2) Duplicate elimination phase: eliminates duplicates from each  = duplicate-free union of , , …, | | | | | | *initialize empty hash table T;*  *for each tuple t in R {*  *apply hash function h on ;*  *let t be hashed to bucket Bi in T;*  *if ( not in Bi) then insert into Bi ;*  *} output all entries in T;*  Cost = |R| if T fits in main memory | |
| Partitioning Phase: - Use 1 buffer for input & (B-1) buffers for output. - Read R 1 page at a time into input buffer  - For each tuple t in input buffer: project out unwanted attrs from t to form t'. apply hash fn on t' to dist t' into 1 output buffer. flush output buffer to disk whenever buffer is full  Duplicate Elimination Phase: For each partition Ri: - initialize an in-memory hash table. - Read 1 page at a time; for each tuple t read,  -- Hash t into bucket Bj w hash function h' (h' ≠ h). -- Insert t into Bj if t Bj. -- Output tuples in hash table | | | | | | | |
| Partition overflow problem: Hash table for is larger than available memory buffers.  Soln: Recursively apply hash-based partitioning to the overflowed partition | | | | | | | |
| Analysis: Approach is effective if B is large relative to |R|  Assume that h dustributes tuples in R uniformly. Each Ri has pages. Size of hash table for each Ri = , where f = fudge factor  To avoid partition overflow, B > . Approximately, B >  Suppose theres no partition overflow (i.e. B > ):  - Cost of partitioning phase: |R| + . - Cost of duplicate elimination phase: . - Total cost = |R| + 2 | | | | | | | |
| Comparison | | | Sort based is good if there are many duplicates or if dist of hashed values is non-uniform  If B > : Num of initial sorted runs N0 = . Num of merging passes = logB-1(N0) ≈ 1  Cost = pass 0 + pass 1 = [ |R| + ] + which is the same I/O for hash based | | | | | | |
| Using Indexes | | If there is an index whose search key (tgt w any include cols) contains all wanted attrs: replace table scan w index scan  If index is ordered (e.g. B+-tree) whose search key includes wanted attrs as a prefix: - scan data entries in order. - compare adjacent data entries for duplicates | | | | | | | |
| Joins | | Algos: 1) Iteration-based: block nested loop. 2) Index-based: index nested loop. 3) Partition-based: sort-merge join, hash join  Things to consider: 1) Type of join predicates: - equality predicates, or - inequality predicates. 2) Sizes of join operands.  3) Available buffer space. 4) Available access mtds. Given a join R S. R = outer relation and S = inner relation | | | | | | | |
| Tuple-based Nested Loop Join | | | | I/O Cost Analysis: |R| + \* |S| | *for each tuple r R*  *for each tuple s S { if (r matches s) then output (r,s) to result }* | | | | |
| Page-based Nested Loop Join | | | | I/O Cost: |R| + |R| \* |S| | *for each page PR of R { for each page PS of S*  *for each tuple r PR { for each tuple s PS { if (r matches s) then output (r,s) to result } } }* | | | | |
| Block Nested Loop Join | | To better exploit buffer space to minimize num of I/Os  Assume |R| ≤ |S|. Choose R as outer & S as inner  Buffer space allocation: Allocate 1 page for S, 1 page for output & remaining pages for R  I/O Cost: |R| + | | | | | *while (scan of R is not done)*  *read next (B-2) pages of R into buffer*  *for each page PS of S*  *read PS into buffer*  *for each tuple r of R in buffer and each tuple s PS*  *{ if (r matches s) then output (r,s) to result }* | | |
| Index Nested Loop Join | | Analysis: Let R.Ai = S.Bj be join condition  - Assume Uniform dist: each R-tuple joins w num of S-tuples  - For format-1 B+-tree index on S, I/O Cost = |R| + \* J, where  J = = search index's internal nodes + search index's leaf nodes | | | | | | | Precondition: there is an index on the join attribute(s) of inner relation S  *for each tuple r R*  *use r to probe S's index to find matching tuples* |
| Sort-Merge Join | Sort both relations based on join attributes & merge them  Sorted relation R consists of partitions Ri of records where r, r' Ri iff r and r' have the same values for the join attribute(s)  Merging phase: Suppose R is outer relation & S is inner  Each tuple in R-partition merges w all tuples in matching S-partition  Pointer is mainted for each sorted join operand  Each pointer initialized to first tuple in sorted operand  Search for matching partitions by advancing pointer that is pointing to a "smaller" tuple  Need to rmb position of first tuple in matching S-partition to enable rewinding of S-pointer  I/O Cost = Cost to sort R + Cost to sort S + Merging Cost  Cost to sort R = 2|R|() if using external merge sort, where NR = num of initial sorted runs of R, m = merge factor  Cost to sort S = 2|R|() if using external merge sort  If each S partition is scanned at most once during merging, merging cost = |R| + |S|  Worst case when each tuple of R requires scanning entire S, merging cost = |R| + \* |S| | | | | | | | | *if (R not sorted) then sort R*  *if (S not sorted) then sort S*  *tr = first tuple in R; ts = first tuple in S*  *ps = first tuple in S partition*  *while (tr ≠ NULL) and (ps ≠ NULL) {*  *while (tr.Ai < ps.Bj) { tr = next tuple in R after tr }*  *while (tr.Ai > ps.Bj) { ps = next tuple in S after ps }*  *ts = ps*  *while (tr.Ai = ps.Bj) {*  *ts = ps*  *while (ts.Bj = tr.Ai) {*  *add (tr , ts) to result;*  *ts = next tuple in S after ts }*  *tr = next tuple in R after tp }*  *ps = ts }* |
| Optimized Sort-Merge Join | | Optimization: Combine merge phase of sorting & merge phase of join. B ≥ , which can be simplified to ->  - If B > N(R, i) + N(S, j) for some i & j, sorting of R and S can stop, where N(R,i) = total num of sorted runs of R at end of pass i of sorting R   |  |  |  |  | | --- | --- | --- | --- | | Sort-Merge Join | Sort R | Sort S | Join R and S | | Conventional | create sorted runs of R; merge sorted runs of R | create sorted runs of S; merge sorted runs of S | merge sorted R & sorted S | | Optimized | create sorted runs of R; merge sorted runs of R partially | create sorted runs of S; merge sorted runs of S partially | Merge remaining sorted runs of R & S and join them at the same time |   Analysis: Assume |R| ≤ |S|. If B > : - num of initial sorted runs of S < . - total num of initial sorted runs of R and S <  - 1 pass is sufficient to merge and join initial sorted runs R & S. - I/O Cost = [ 2 \* (|R| + |S|) ] + (|R| + |S|) = pass 0 + pass 1 = 3(|R| + |S|) | | | | | | | |
| Hash Join | | For RS. Partition R and S into k partitions using some hash fn h: - R = R1 R2 … Rk, t Ri iff h(t.A) = i  - S = S1 S2 … Sk, t Si iff h(t.B) = i. - for each pair Ri & Sj, i ≠ j  Joins corresponding pair of partitions R S = (R1 S1) (R2 S2) … (Rk Sk). Algos: Grace hash join & Hybrid hash join (not covered) | | | | | | | |
| Grace Hash Join | | R is called the build (inner, smaller) relation. S is called the probe (outer) relation. 1) Partition R into R1, …, Rk. 2) Partition S into S1, …, Sk.  3) Probing phase: probes each Ri with Si: - Read Ri to build hash table. - Read Si to probe hash table | | | | | | | |
| Partitioning phases:  *initialize hash table T w k buckets*  *for each tuple r R { insert r into bucket h(r.A) of T }*  *write each bucket Ri of T to disk*  *initialize hash table T w k buckets*  *for each tuple s S { insert s into bucket h(s.B) of T }*  *write each bucket Si of T to disk* | | | | Probing phase  *for i = 1 to k*  *initialise hash table T*  *for each tuple r in partition Ri { insert r into bucket h'(r.A) of T }*  *for each tuple s in partition Si {*  *for each tuple r in bucket h'(s.B) of T { if r and s matches then ouput (r,s) }* | | | |
| Analysis: To minimize size of each partition of Ri: Set k = B-1 given B buffer pages  Assume uniform hash dist: - size of each partition Ri is |R|/(B-1). - size of hash table for Ri is f \* |R| / (B-1), where f = fudge factor  - During probing phase, B > f \* |R| / (B-1) + 2 (w 1 input buffer for Si & 1 output buffer). - Approximately, B >  Can assume B > [ size of each partition of min(R,S) + 1 + 1 ]. f ≈ 1.2  Partition overflow problem: Hash table for Ri don't fit in memory. Soln: recursively apply partitioning to overflow partitions  I/O Cost = cost of partitioning phases + cost of probing phase = [ 2 \* (|R| + |S|) ] + [ |R| + |S| ] = 3 \* (|R| + |S|) (if no partition overflow) | | | | | | | |
| General Join conditions | | Multiple equality-join conditions: (R.A = S.A) and (R.B = S.B). Algos: 1) Index nested loop join: use index on all or some of join attrs. 2) Sort-merge join: need to sort on combinations of attrs. 3) Other algos unchanged  Inequality-join conditions: (R.A < S.A). Algos: 1) Index Nested Loop Join: requires B+-tree index. 2) Sort-Merge Join & Hash-based Joins: N/A. 3) Other algos unchanged | | | | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Set Operations | | Cross-product, Intersection are just special cases of join. Union & Difference can use sorting-based or hashing-based approach  Sorting approach for R S: Sort R using all attrs. Sort S using all attrs. Merge sorted operands to combine and discard duplicates  Hashing for R S: Partition R into {R1, …, Rk} using hash fn h on all attrs. Partition S into {S1, …, Sk} using hash fn h on all attrs.  For i = 1 to k: { - Build an in-memory hash table Ti (using hash fn h') for Si. - Scan Ri: for each tuple t Ri, probe Ti. Discard t if t is in Til otherwise, insert t into Ti. - Write Ti to disk }  Algos for R - S are similar to those for R S | | | |
| Aggregation | | Sorting Approach: - Sort relation on grouping attr(s). - Scan sorted relation to compute aggregate for each group  Hashing: - Scan relation to build hash table on grouping attr(s). - For each group, maintain (grouping-value, running-info)  Avoid table scan: If there's a covering index for query, aggregation op can be computed from index's data entries instead of data records  Ordered scan using index: For group-by aggregation, if set of group-by attrs forms a prefix of a B+-tree index's search key, the data entries (and data records if necessary) for each group can be retrieved w/o an explicit sorting | | | |
| Query Evaluation | Logical -> Physical plan (with choice of algos)  1) Materialized evaluation: - Operator is evaluated only when each of its operands has been completely evaluated or materialized  - Intermediate results are materialized to disk. - Bottom up approach  2) Pipelined evaluation: - Output produced by a operator is passed directly to its parent operator. - Execution of operators is interleaved.  - Top down approach. - Operator O is a blocking operator if O may not be able to produce any output until it has received all the input tuples from its child operator(s) (e.g. external merge sort, sort-merge join, grace hash join)  - Typically have 3 functions: 1) open: initialize state of iterator. Allocate resources for op. Initializes operator's arguments (selection conditions). 2) getNext: generate next output tuple. Return null if all output tuples have been generated. 3) close: deallocates state info  Can combine pipelined evaluation w partial materialization | | | | |
| Hash-based Projection (w operand R)  **open()**  Allocate memory for op. Initialize hash table HT  R.open(); r = R.getNext()  while (r ≠ NULL)  Hash on r & write r to appropriate output page  r = R.getNext()  Let R be partitioned into R1, …, Rn  **getNext()**  while (i ≤ n) { Initialize hash table HT for partition Ri  Hash each tuple r from Ri; Insert r into HT if not duplicate  while (HT not empty) { return next tuple from HT }  i = i + 1  return null }  **close()** {R.close(); Deallocate memory} | | | Nested Loop Join (w operands L & R)  **open()**  Allocate memory for op;  L.open(); R.open()  **getNext()**  l = L.getNext()  while (l ≠ NULL)  r = R.getNext()  while (r ≠ null)  if (l & r satisfy p) then return (l, r)  r = R.getNext()  R.rewind(); l = L.getNext()  return NULL  **close()** { L.close(); R.close();  Deallocate memory } | Table scan  (w operand R & predicate p)  **open()** {R.open()}  **getNext()**  r = R.getNext()  while (r ≠ NULL)  if (r satisfies p) then return r  r = R.getNext()  return NULL  **close()** {R.close()} |
| Query plans | | A query generally has many equivalent logical query plans. Each logical plan can be implemented by many physical query plans  For join plan notation: outer relation is left child, inner relation is right child. For hash join: probe relation is left child, build relation is right | | | |
| RA Equivalence | | attributes(R) = set of attrs in schema of relation R. attributes(p) = set of attrs in predicate p  1. Commutativity of binary operators: 1.1) . 1.2)  2. Associativity of binary operators: 2.1) . 2.2)  3. Idempotence of unary operators: 3.1) . 3.2)  4. Commutating selection w projection: 4.1)  5. Commutating selection w binary operators: 5.1) (if attrs(p) attrs(R)).  5.2) (if attrs(p) attrs(R)). 5.3)  6. Commutating projection w binary operators: Let L = LR LS, where LR attributes(R) and LS attributes(S)  6.1) . 6.2) . (if attrs(p)attrs(R) LR and attrs(p)attrs(S) LS)  6.3) | | | |
| Query Optimiza-tion | | A diagram of a tree  Description automatically generatedA diagram of a tree  Description automatically generated1) Search space: what is the space of query plans being considered.  2) Plan enumeration: how to enumerate the space of query plans  3) Cost model: how to estimate cost of query plan  Types of query plan trees: 1) Linear: if at least 1 operand for each join operation is a base relation; otherwise plan is bushy  2) Left-deep: if every right join operand is a base relation. 3) Right-deep: if every left join operand is a base relation | | | |
| Query plan enumera-tion | | Dynamic programming formulation  Input: A SPJ query q on relations R1, R2, …, Rn  Output: An optimal query plan for q  O(3n) | *for i = 1 to n: { optPlan({Ri}) = best access plan for Ri }*  *for i = 2 to n: { for each S {R1, …, Rn}, |S| = i: {*  *bestPlan = dummy plan w cost(bestPlan) = ∞*  *for each Sj, Sk, |Sj| [1,i), S = Sj Sk: { p = best way to join optPlan(Sj) and optPlan(Sk)*  *if (cost(p) ≤ cost(bestPlan)): { bestPlan = p } }*  *optPlan(S) = bestPlan } return optPlan({R1, …, Rn}) }* | | |
| System R Optimizer | | Uses heuristics to prune search space: - enumerates only left deep query plans. - avoids cross-product query plans. - considers early &  Uses enhance dynamic programming approach that considers sort order of query plan's output:  - maintains optPlan(Si, oi) instead of optPlan(Si). - oi captures the sort order of output produced by query plan w.r.t Si  - oi = NULL if output is unordered or a seq of attrs. - optPlan(Si, oi) = cheapest query plan for relations Si w output ordered by oi if oi ≠ NULL | | | |
| Cost Estimation of Query Plans | | Depends on size of input operands, available buffer pages, available indexes, ...  Assumptions: - Uniformity: uniform dist of attr values. - Independence: indep dist of values in diff attrs.  - Inclusion: For R S, if , then  DB statistics: - relation cardinality, - num of distinct values & max & min in each col, - freq values, - col group statistics, - histograms, ... | | | |
| Size estimation. Consider query q = , where p = t1 t2 … tn, and e = R1 R2 … Rm  . Each term ti potentially filters out some tuples in e.  Reduction factor of a term ti = rf(ti) = fraction of tuples in e that satisfy ti = = aka selectivity factor  Assuming terms in p are statistically independent,  Consider relation R(A,…) w = 45 and = 15. Using uniformity assumption, rf(A = c) ≈ 1/. So estimated to be 3 | | | |
| Join selectivity factor = selectivity factor for join predicates. rf(R.A = S.B) =  Inclusion assumption: If , then  Join selectivity estimation: assume . By inclusion assumption, every R-tuple joins w some S-tuple  By uniformity assumption, there are S-tuples corresponding to each S.B value  Thus, each R-tuple joins w S-tuples. So . rf(R.A = S.B) ≈ 1/max{, } | | | |
| Estimation using Histograms. Main idea: - partition attribute's domain into buckets. Assume value dist within each bucket is uniform  1) Equiwidth histograms: each bucket has (almost) same num of values  2) Equidepth histograms: each bucket has (almost) same num of tuples. Sub-ranges of adjacent buckets might overlap | | | |
| Improved Histogram Estimation w MCV (most common values). Track freq of top-k MCV and exclude MCV from histogram's buckets | | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Transaction (Xact) | Abstraction representing a logical unit of work  Ensures 4 properties of Xacts to maintain data in the face of concurrent access and system failures  1) Atomicity: either all or none of the actions in Xact happen  2) Consistency: If each Xact is consistent, and DB starts consistent, DB ends up consistent  3) Isolation: Execution of 1 Xact is isolated from other Xacts  4) Durability: If a Xact commits, its effects persist  Concurrency control manager component ensures 3). Recovery manager ensures 1) and 4). | | | | | BEGIN TRANSACTION  SELECT balance INTO :Balx FROM Account WHERE accountId = :x; ...  If (:Balx < amount) then ABORT;  UPDATE Account SET balance = :Baly + :amount WHERE accountId = :y; ...  COMMIT; |
| Xact Ti can be viewed as a seq of actions: 1) Ri(O) = Ti reads an object O. 2) Wi(O) = Ti writes an object O  3) Commiti = Ti terminates successfully. 4) Aborti = Ti terminates unsuccessfully  Each Xact must end w either commit or abort action. An active Xact = Xact that is still in progress (not yet terminated) | | | | | |
| Xact Schedules | Schedule = list of actions from a set of Xacts, where order of actions within each Xact is preserved  A serial schedule = schedule where actions of Xacts are not interleaved | | | | | |
| - Tj reads O from Ti in a schedule S if last write action on O before Rj(O) in S is Wi(O). - Tj reads from Ti if Tj has read some object from Ti  - Ti performs final write on O in a schedule S if last write action on O in S is Wi(O)  An interleaved Xact execution schedule is correct if it is "equivalent" to some serial schedule over the same set of Xacts | | | | | |
| 2 schedules S and S' (over the same set of Xacts) are view equivalent (S S') if they satisfy:  1. If Ti reads A from Tj in S, then Ti must also read A from Tj in S'  2. For each data obj A, the Xact (if any) that performs the final write on A in S must also perform the final write on A in S'  View Serializable Schedule (VSS) = a schedule S that is some serial schedule over the same set of Xacts | | | | | |
| Testing for View Serializability | | Given schedule S, construct a directed graph, VSG(S) to capture read-from and final-write relations among transactions in S  Nodes in VSG(S) = Xacts, edges = precedence relations among Xacts. Edge (Tj, Ti) VSG(S) if:  - If Ti reads from Tj. - If both Ti & Tj updates the same obj O & Ti performs final write on O.  - If Tj reads obj O from Tk & Ti update obj O, then either (Ti, Tk) VSG(S) or (Tj, Ti) VSG(S)  If VSG cyclic, S not VSS. If VSG acyclic, then S is VSS iff a serializable schedule produced from a topological ordering of VSG that is S | | | | |
| Conflicting Actions | | | | 2 actions on the same object conflict if 1) at least 1 of them is a write action, and 2) the actions are from diff Xacts | | |
| Anomalies w Interleaved Xact Executions | 1) Dirty read problem (due to WR conflict): - T2 reads an obj that has been modified by T1 and T1 has not yet committed  - T2 could see an inconsistent DB state  2) Unrepeatable read problem (RW conflicts): - T2 updates an obj that T1 has previously read and T2 commits while T1 is still in progress.  - T1 could get a diff value if it reads the obj again  3) Lost update (WW conflict): - T2 overwrites value of an obj that has been modified by T1 while T1 still in progress. - T1's update is lost | | | | | |
| Conflict Serializable Schedules | 2 schedules S and S' (over the same set of Xacts) are conflict equivalent (S S') if they order every pair of conflicting actions of 2 commited Xacts in the same way  A schedule is a conflict serializable schedule (CSS) if it is conflict equivalent to a serial schedule over the same set of Xacts | | | | | |
| A conflict serializability graph for a schedule S, CSG(S) is a directed graph G = (V, E) s.t. V = node for each commited Xact in S, and E contains (Ti, Tj) if an action in Ti precedes and conflict w one of Tj's actions  S = R1(A), W2(A), Commit2, W1(A), Commit1, W3(A), Commit3  Thrm 1: A schedule is conflict serializable iff its conflict serializability graph is acyclic  Thrm 2: A schedule that is conflict serializable is also view serializable.  For convenience, use serializable to mean conflict serializable | | | | | |
| Blind Writes | Blind write = A write on obj O by Ti where Ti did not read O prior to the write. E.g. R1(x), W2(y), W1(x). W2(y) is a blind write. W1(x) is not  Thrm 3: If S is view serializable and S has no blind writes (WW), then S is also conflict serializable | | | | | |
| Cascading Aborts | | | | | For correctness, if Ti has read from Tj, then Ti must abort if Tj aborts. Recursive aborting process = cascading abort | |
| Recoverable Schedules | | | | | Recoverable schedule = for every Xact T that commits in schedule S, T must commit after T' if T reads from T' | |
| Cascadeless Schedules | While recoverable schedules guarantee that committed Xacts will not be aborted, cascading aborts of active Xacts are possible  Cascading aborts are undesirable due to cost of bookkeeping to identify them and performance penalty incurred  To avoid cascading aborts / to be cascadeless, DBMS must permit reads only from committed Xacts  Cascadeless schedule = whenever Ti reads from Tj in schedule S, Commitj must precede this read action  Thrm 4: A cascadeless schedule is also a recoverable schedule | | | | | |
| Recovery using Before-Images | | | An efficient approach to undo actions of aborted Xacts is to restore before-images for writes. E.g. W1(A), W2(A), Abort2  However, before-image recovery doesn't always work. E.g. W1(A), W2(A), Abort1. Will undo effect of W2 as well | | | |
| Strict Schedules | Strict schedule = for every Wi(O) in schedule S, O is not read or written by another Xact until Ti either aborts or commits  Strict schedules enables use of before-images for recovery. Tradeoff: - recovery more efficient. - concurrent executions more restrictive  Thrm 5: A strict schedule is also a cascadeless schedule | | | | | |
| Serial schedule must be strict & conflict serializable. Cascadeless if all read operations are non-dirty. Strict if all read and write operations are non-dirty | | | | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Concurrency Control | For each input action (read, write, commit, abort) to the transaction scheduler, the scheduler performs one of the following:  - output the action to the schedule, - postpone the action by blocking the transaction or - reject the action and abort the transaction  Concurrency control algos: 1) Lock-based. 2) Timestamp-based. 3) Multiversion. 4) Optimistic | | |
| 1) Lock-Based Concurrency Control | Each Xact needs to request for an appropriate lock on an obj before the Xact can access the obj  Locking modes: - Shared (S) locks for reading objects. - Exclusive (X) locks for writing objects  Let Si(O) = Xact Ti is requesting S-lock on obj O. Xi(O) similarly defined  Ui(O) = Xact Ti releases lock on obj O | | |  |  |  |  | | --- | --- | --- | --- | | Lock Requested | Lock Held | | | | - | S | X | | S | granted | granted | blocked | | X | granted | blocked | blocked |   Lock compatibility: |
| 1) To read an obj O, a Xact must request for a shared/exclusive lock on O  2) To update an obj O, a Xact must request for an exclusive lock on O  3) A lock request is granted on O if requesting lock mode is compatible w lock modes of existing locks on O  4) If T's lock request is not granted on O, T becomes blocked: its execution is suspended & T is added to O's request queue  5) When a lock is released on O, lock manager checks the request of the 1st Xact T in the request queue for O. If T's request can be granted, T acquires its lock on O and resumes execution after its removal from the queue  6) When a Xact commits/aborts, all its locks are released & T is removed from any request queue it's in | | |
| Two Phase Locking (2PL) Protocol | 1) To read an obj O, a Xact must hold a S-lock or X-lock on O. 2) To write to an obj O, a Xact must hold a X-lock on O  3) Once a Xact releases a lock, the Xact can't request any more locks  Xacts using 2PL can be split into 2 phases: 1) Growing phase: before releasing 1st lock. 2) Shrinking phase: after releasing 1st lock  Thrm 1: 2PL schedules are conflict serializable | | |
| Strict 2PL Protocol: 1) and 2) are the same. 3) A Xact must hold on to locks until Xact commits or aborts  Thrm 2: Strict 2PL schedules are strict & conflict serializable | | |
| Deadlocks | Cycle of Xacts waiting for locks to be released by each other. To deal w deadlocks: either detect deadlock or prevent deadlock | | |
| To detect deadlocks: use Waits-for graph (WFG): - Nodes = active Xacts. - Edge Ti Tj if Ti waiting for Tj to release a lock  Lock manager: - adds an edge when it queues a lock request. - updates edges when it grants a lock request  Deadlock is detected if WFG has a cycle. Breaks a deadlock by aborting a Xact in cycle  Alternative to WFG: timeout mechanism | | |
| To prevent deadlocks: Assume older Xacts have higher priority than younger Xacts  - Each Xact is assigned a timestamp when it starts. - Older Xact has smaller timestamp   |  |  |  | | --- | --- | --- | | Prevention Policy | Ti has higher priority | Ti has lower priority | | Wait-die | Ti waits for Tj | Ti aborts | | Wound-wait | Tj aborts | Ti waits for Tj |   Suppose Ti requests for a lock that conflicts w a lock held by Tj  1) Wait-die policy: lower priority Xacts never wait for higher priority Xacts  - non-preemptive: only a Xact requesting for a lock can get aborted  - a younger Xact may get repeatedly aborted  - a Xact that has all the locks it needs is never aborted  2) Wound-wait policy: higher priority Xacts never wait for lower priority Xacts. - preemptive  To avoid starvation, a restarted Xact must use its original timestamp | | |
| Lock conversion | Increase concurrency by allowing lock conversions. Interleaved executions become possible w lock upgrading  UGi(A) = Ti upgrades its S-lock on obj A to X-lock. - Upgrade request blocked if another Xact is holding a shared lock on A  - Upgrade request allowed if Ti has not released any lock  DGi(A): Ti downgrades its X-lock on obj A to S-lock. - Downgrade request allowed if Ti has not modified A & Ti has not released any lock | | |
| Performance of Locking | Resolve Xact conflicts by using blocking & aborting mechanisms. Blocking causes delays in other waiting Xacts  Aborting and restarting a Xact wastes work done by Xact. | To incr sys throughput: 1) Reduce locking granularity.  2) Reduce time a lock is held.  3) Reduce hot spots = DB obj that is frequently accessed and modified | |
| Concurrency Control Anomalies & Locking | Dirty read: W1(x), R2(x). Unrepeatable read: R1(x), W2(x), Commit2, R1(x). Lost update: R1(x), R2(x), W1(x), W2(x)  Phantom read problem: A Xact re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently committed transaction. Similar to unrepeatable read  Let R(p) = generalization of read action R(x) where p = selection predicate on some relation. R(p) reads all objs that satisfies p  R(p) and W(x) conflicts if obj x satisfies p. Phantom Read: R1(p), W2(x), Commit2, R1(p), Commit1  Phantom problem can be prevented by predicate locking. In practice, phantom problem is prevented via index locking | | |
| ANSI SQL Isolation Levels  In many DBMSs, default isolation level = READ COMMITTED | |  |  |  |  |  | | --- | --- | --- | --- | --- | | Isolation Level | READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE | | Dirty Read | possible | not possible | not possible | not possible | | Unrepeatable Read | possible | possible | not possible | not possible | | Phantom Read | possible | possible | possible | not possible |   BEGIN TRANSACTION; SET TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED | ... | SERIALIZABLE }; ... COMMIT;   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Degree | Isolation level | Write Locks | Read Locks | Predicate Locking | | 0 | READ UNCOMMITTED | long duration | none | none | | 1 | READ COMMITTED | long duration | short duration | none | | 2 | REPEATABLE READ | long duration | long duration | none | | 3 | SERIALIZABLE | long duration | long duration | yes |   Short duration lock: lock acquired for an operation could be released after end of operation before Xact commits/aborts  Lond duration: lock acquired for an operation is held until Xact commits/aborts | | |
| Locking Granularity | Locking granularity = size of data items being locked. Allow multi-granular lock instead of fixed granule locking  Highest (coarsest) granularity = DB -> relation -> page -> tuple = Lowest (finest) granularity  If Xact T holds a lock mode M on a data granule D, then T implicitly also holds lock mode M on granules finer than D | | |
| A diagram of a diagram  Description automatically generatedMultigranularity Locking: Use a new intention lock (I-lock) mode   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Lock Requested | Lock Held | | | | | - | I | S | X | | I | granted | granted | blocked | blocked | | S | granted | blocked | granted | blocked | | X | granted | blocked | blocked | blocked |   Before acquiring S-lock/X-lock on a data granule G, need to acquire I-locks on granules coarser than G in a top-down manner  E.g. Xact T1 wants to request X-lock on tuple t4  T1 request for I-lock on D, R, P2.  T1 request for X-lock on t4.  If T2 wants to read P2, its I-lock  requests on D & R will be granted, but its S-lock request on P2 will be blocked | | |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Lock Requested | Lock Held | | | | | | - | IS | IX | S | X | | IS | granted | granted | granted | granted | blocked | | IX | granted | granted | granted | blocked | blocked | | S | granted | granted | blocked | granted | blocked | | X | granted | blocked | blocked | blocked | blocked |   Problem: limited concurrency w lock modes I, S and X. So further refine intention lock idea  - Intention Shared (IS): intent to set S-locks at finer granularity  - Intention exclustive(IX): intent to set X-locks at finer granularity  Locks are acquired in top-down order  To obtain S or IS lock on a node, must already hold IS or IX lock on its parent node  To obtain X or IX lock on a node, must already hold IX lock on its parent node. Locks are released in bottom-up order | | |

|  |  |  |
| --- | --- | --- |
| Multiversion Concurrency Control | Maintain multiple versions of each obj. Wi(O) creates a new version of obj O. Ri(O) reads an appropriate version of O  Pros: - Read-only Xacts not blocked by update Xacts. - Update Xacts not blocked by read-only Xacts. - Read-only Xacts never aborted  Wi(x) creates a new version of x denoted by xi. For each obj x, its initial version is denoted by x0 | |
| E.g. R2(x), W1(y), R2(y). R2(y) is not blocked and return value of the version before W1(y)  If there are multiple versions of an obj x, a read action on x could return any version  Thus, an interleaved execution could correspond to diff multiversion schedules depending on the MVCC protocol | |
| 2 schedules, S and S', over the same set of transactions are Multiversion View Equivalent (S S') if they have the same set of read-from relationships. i.e. Ri(xj) occurs in S iff Ri(xj) occurs in S' | |
| Monoversion schedule = a multiversion schedule S where each read action in S returns the most recently created obj version  Serial monoversion schedule = a monoversion schedule where it is also a serial schedule (i.e. no interleaving) | |
| Multiversion view serializable schedule (MVSS) = a multiversion schedule S where there exists a serial monoversion schedule (over the same set of Xacts) that is multiversion view equivalent to S  Thrm 1: A view serializable schedule (VSS) is also a multiversion view serializable schedule (MVSS). MVSS not necessarily VSS  CSS -> VSS -> MVSS. Not MVSS -> Not CSS | |
| Test for MVSS: use MVSG(S) to capture read-from relations among transactions in S (similar to VSG). Cyclic = not MVSS. Acyclic = MVSS  Edge (Tj, Ti) MVSG(S) if: - Ti reads from Tj. - Tj reads obj O from Tk & Ti update obj O, then either (Ti, Tk) MVSG(S) or (Tj, Ti) MVSG(S) | |
| MVCC Protocols | 1) Multiversion two-phase locking. 2) Multiversion timestamp ordering. 3) Snapshot isolation (SI; used by Oracle, PostgreSQL, SQL server) | |
| Each Xact T sees a snapshot of DB that consists of updates by Xacts that committed before T starts  Each Xact T is associated w 2 timestamps: start(T) = time that T starts. commit(T) = time that T commits | |
| 2 Xacts T and T' are concurrent if they overlap, i.e. [start(T), commit(T)] [start(T'), commit(T')] ≠ | |
| SI: - Wi(O) creates a version of O denoted by Oi. Oi is a more recent/newer version compared to Oj if commit(Ti) > commit(Tj)  - Ri(O) reads either its own update (if Wi(O) precedes Ri(O)) or the latest version of O created by a Xact that committed before Ti started,  - Concurrent Update Property: If multiple concurrent Xacts updated the same obj, only 1 of Xacts is allowed to commit. If not, schedule may not be serializable. To enforce this property: 1) First Committer Wins (FCW) rule. 2) First Updater Wins (FUW) rule | |
| FCW: Before commiting a Xact T, system checks if there exists a committed concurrent Xact T' that has updated some obj that T has also updated. If T' exists, then T aborts. Otherwise, T commits | |
| FUW: Whenever a Xact T needs to update an obj O, T requests for a X-lock on O.  If X-lock not held by any concurrent Xact, then - T is granted X-lock on O. - If O has been updated by any concurrent Xact, then T aborts.  - Else T proceeds with its execution  Otherwise, if X-lock is held by some concurrent Xact T', then T waits until T' aborts or commits.  If T' aborts, then: - assume T is granted X-lock on O. - If O has been updated by any concurrent Xact, then T aborts.  - Else T proceeds with its execution.  If T' commits, then T is aborted | |
| Garbage Collection | A version Oi of obj O may be deleted if there exists a newer version Oj (i.e. commit (Ti) < commit(Tj)) s.t. for every active Xact Tk that started after the commit of Ti (i.e. commit(Ti) < start(Tk)), we have commit(Tj) < start(Tk)  E.g. W1(x1), C1, W2(x2), C2, W4(x4), R3(y0), C4, W5(x5), C5, R6(z0). Active transactions: T3 and T6. Versions that can be deleted: x1 & x4 | |
| Snapshot Isolation Tradeoffs | Performance of SI often similar to READ COMMITTED. Unlike READ COMMITTED, SI don't suffer from lost update or unrepeatable read anomalies. But SI is vulnerable to some non-serializable executions: 1) Write Skew Anomaly. 2) Read-Only Transaction Anomaly  SI don't guarantee serializability | |
| Write Skew Anomaly &  Read-Only Transaction Anomaly | A diagram of a circle with text  Description automatically generatedR1(x0), R2(x0), R1(y0), R2(y0), W1(x1), Commit1, W2(y2), Commit2. This is a SI schedule but not a MVSS (LHS)  R1(y0), R2(x0), W1(y1), Commit1, R2(y0), W2(x2), R3(x0), R3(y1), Commit3, Commit2. This is a SI schedule but not MVSS (RHS)  Graphs shows DSG for Write skew & Read only. | |
| Serializable Snapshot Isolation (SSI) Protocol | | Stronger protocol that guarantees serializable SI schedules. Keep track of rw dependencies among concurrent Xacts  Detect formation of Tj involving 2 rw dependencies. Once detected, abort 1 of Ti, Tj, or Tk.  May result in unnecessary rollbacks due to false positives of SI anomalies  SSI schedule = schedule S that is SI and MVSS |
| Transactio-nal Dependen-cies | - ww dependency from T1 to T2: T1 writes a version of some data item x, and T2 later writes the immediate successor version of x  - wr dependency from T1 to T2: T1 writes a version of some data item x and T2 reads this version of x  - rw dependency from T1 to T2: T1 reads a version of some data item x, and T2 later creates the immediate successor version of x  xj = immediate successor of xi if 1) Ti commits b4 Tj, and (2) no transaction that commits btw Ti's and Tj's commits produces a version of x | |
| Dependency Serialization Graph (DSG) | A diagram of a diagram  Description automatically generatedConsider a schedule S consisting of a set of committed transactions {T1, …, Tk}. DSG(S) = edge-labelled directed graph (V, E)  V = transactions {T1, …, Tk}. E = transactional dependencies: Ti Tj or Ti Tj or Ti Tj  Edge types: - -> if transaction pair is concurrent. –> if transaction pair is non-concurrent  E.g. Schedule S: W1(x), W1(y), W1(z), C1, W3(x), R2(x), W2(y), C2, R3(y), C3 (diag wrong for T3 to T2, wr???) | |
| Non-MVSS SI schedules | Thrm 2: If S is a SI schedule that is not MVSS, then 1) There is at least 1 cycle in DSG(S), and  2) For each cycle in DSG(S), there exists 3 transactions, Ti, Tj and Tk s.t. - Ti & Tk are possibly the same transaction  - Ti & Tj are concurrent w an edge Ti Tj, and Tj & Tk are concurrent w and edge Tj Tk | |

|  |  |  |
| --- | --- | --- |
| Crash Recovery | | Recovery manager guarantees atomicity and durability properties of Xacts  - Undo: remove effects of aborted Xact to preserver atomicity. - Redo: re-installing effects of committed Xact for durability  3 types of failures: 1) Transaction failures: transaction aborts. App rollbacks transaction OR System rollbacks transaction (e.g. deadlock, violation of integrity constraint). 2) System crash: loss of volatile memory contents. Power failure, bug in DBMS/OS, hardware malfunction  3) Media failures: data is lost/corrupted on non-volatile storage. Disk head crash/failure during data transfer |
| Recovery Manager | | Commit(T): install T's updated pages into DB. Abort(T): restore all data that T updated to their prior values.  Restart: recover DB to a consistent state from system failure. Abort all active Xacts at time of system failure. Installs updates of all committed Xacts that were not installed in the DB before the failure  Desirable properties: Add little overhead to normal processing of Xacts. Recover quickly from a failure |
| Interaction of Recovery & Buffer Managers | | |  |  |  | | --- | --- | --- | |  | Force | No-force | | Steal | undo & no redo | undo & redo | | No-steal | no undo & no redo | no undo & redo |   Steal policy: allows dirty pages updated by Xact T to be replaced (written to disk) from buffer pool before T commits. No-steal policy = no undo  Force policy: requires all dirty pages updated by Xact T to be written to disk when T commits. Force policy = no redo |
| Log-based DB Recovery | | Log = history of actions executed by DBMS. Contains a log record for each write, commit & abort  Log is stored as a sequential file of records in stable storage (storage that can survive crashes & media failures – implemented by maintaining multiple copies of info, possibly at diff locations, on non-volatile storage devices).  Each log record has a unique identifier called Log Sequence Number (LSN). Earlier log records = smaller LSN |
| ARIES Recovery Algo (Algo for Recovery and Isolation Exploiting Semantics) | | Designed to work w a steal, no-force approach. Assumes strict 2PL for concurrency control  Recovery-related Structures: - Log file. - Transaction table (TT): 1 entry for each active Xact. Each entry contains XactID, **lastLSN** = LSN of most recent log record for this Xact, Xact status (C or U). C = committed, U = uncommitted  - Dirty page table (DPT): 1 entry for each dirty page in buffer pool. Each entry contains pageID = page ID of dirty page, **recLSN** = LSN of earliest log record for an update that caused page to be dirty |
| Implementing Abort. Undo all updated by Xact to DB pages  Write-ahead logging (WAL) protocol: don't flush an uncommitted update to the DB until the log record containing its before-image has been flushed to the log  - Each DB page contains the LSN of the most recent log record = **pageLSN**, that describes an update to this page  - Before flushing a DB page P to disk, ensure all log records up to the log record corresponding to P's **pageLSN** have been flushed to disk  To undo all updates of Xact: for each log record of Xact in reverse order, restore log record's before-image  To efficiently retrieve Xact's log records in reverse order: - Xact Table (TT) = maintains 1 entry for each active Xact  - Use **lastLSN** to get most recent log record for Xact; other log records for Xact are retrieved via the **prevLSN** of each retrieved log record  Logging changes during Undo: changes made to DB while undoing a Xact are also logged to ensure that an action is not repeated in event of repeated undos |
| Implementing Commit. Need to ensure all updates of Xact must be in stable storage (DB or log) before Xact is committed  Force-at-commit protocol: don't commit a Xact until the after-images of all its updated records are in stable storage (DB or log)  - Write a commit log record for Xact. - Flush all log records for Xact to disk  Xact is considered to have committed if its commit log record has been written to stable storage |
| Implementing Restart. Recovery from system crashes consists of 3 phrases:  1) Analysis phase: identifies dirtied buffer pool pages & active Xacts at time of crash  2) Redo phase: redo actions to restore DB state to what it was at time of crash. 3) Undo phase: undo actions of Xacts that didn't commit  Repeating History during Redo: During restart following a crash, first restore system to the state before crash, and then undo the actions of Xacts that are active at the time of crash |
| Normal Transaction Processing. 1) Updating Xact Table (transID, **lastLSN**, status)  - when first log record is created for Xact T, create a new entry for T w status = U  - When a new log record ris created for Xact T, update **lastLSN** for T’s entry to be r’s LSN  - If Xact Tcommits, update status for T's entry to be C. - When an end log record is generated for Xact T, remove T's entry  2) Updating DPT (pageID, **recLSN**). - When a page P in buffer pool is updated & DPT has no entry for P, create a new table entry for P w **recLSN** = LSN of log records corresponding to update. - When a dirtied page P in buffer pool is flushed to disk, remove entry for P |
| Types of Log Records | Common Info in log records: - type (update, commit, abort, …), - identifier of Xact, - **prevLSN** = LSN of previous log record for the same Xact  1) Update log record (ULR): After updating a page P, create an update log record r. Update **pageLSN** of P = LSN of r. Extra fields: - pageID, - offset = byte offset within page indicating beginning of updated portion. - length = num of bytes for updated portion of data page.  - before-image = value of changed bytes before update. - after-image = value of changed bytes after update  2) Compensation log record (CLR): When update described by an ULR is undone, create a CLR.  Add fields: - pageID, - **undoNextLSN** = LSN of next log record to be undone (i.e. prevLSN in ULR). - action taken to undo update  3) Commit log record: When a Xact is to be committed, create a commit log record r. All log records (up to and including r) are force-written to stable storage. Xact is considered committed once r has been written to stable storage  4) Abort log record: When a Xact is to be aborted, create an abort log record. Undo is initiated for this Xact  5) End log record: Once additional follow-up processing initiated by a aborted/committed Xact has completed, create an end log record  - During DB recovery, presence of an E log record for a Xact indicates that the Xact committed successfully and that its changes can be safely applied to the DB  6) Checkpoint log record  ULR & CLRs are classified as redoable log records | |
| Analysis phase | 1) Determines the point in the log to start the Redo phase. 2) Determines the superset of buffer pages that were dirty at time of crash  3) Identifies active Xacts at time of crash  Initializes DPT and TT to be empty. Scan the log in forward direction to process each log record r (for Xact T):  - If r is an end log record: { Remove T from TT }  Else { Add an entry in TT for T if T not in TT. Update **lastLSN** of entry to be r's LSN. Update status of entry to C if r is a commit log record }  - If (r is a redoable log record for page P) & (P is not in DPT): { Create an entry for P in DPT w pageID of entry = P's pageID and **recLSN** of  entry = r's LSN }  At end of analysis phase: TT = list of all active Xacts (w status = U) at time of crash. DPT = superset of dirty pages at time of crash | |
| Redo phase | | **RedoLSN** = smallest **recLSN** among all dirty pages in DPT. Let r be log record w LSN = **RedoLSN**. Scan the log in forward dirn starting from r.  If (r is an ULR) or (r is a CLR): { Fetch page P that is associated w r.  If (r's LSN > P's **pageLSN**) then { reapply logged action in r to P. updates P's **pageLSN** = r's LSN } }  At end of Redo Phase, create end log records for Xacts w status = C in TT & remove their entries from TT. System is restored to state at time of crash. |
| Undo phase | | Abort active Xacts at time of crash (loser Xacts). Abort loser Xacts by undoing their actions in reverse order  Initialize L = set of **lastLSNs** (w status = U) from TT. Repeat until L becomes empty: {  Delete largest **lastLSN** from L. Let r = log record corresponding to this **lastLSN**.  If r is an update log record for Xact T on page P, then { - create a CLR r2 for T: r2's **undoNextLSN** = r's **prevLSN**. - update T's entry in TT: **lastLSN** = r2's LSN. - undo the logged action on page P. - update P's **pageLSN** = r2's LSN. - *Update-L-and-TT* (r's **prevLSN**) }  Else if r is a CLR for Xact T, then { *Update-L-and-TT*(r's **undoNextLSN**) }  Else if r is an abort log record for Xact T, then { *Update-L-and-TT*(r's **prevLSN**) } }  def *Update-L-and-TT*(lsn): { if lsn is not null: {add lsn to L}. Else { create an end log record for T & remove T's entry from TT } } |
| Checkpoint-ing | | Perform checkpoint operations periodically to speed up restart recovery. Checkpointing synchronizes state of log w DB state  Simple Checkpointing: 1) Stop accepting any new update, commit & abort operations. 2) Wait till all active update, commit & abort ops have finished. 3) Flush all dirty pages in buffer. 4) Write a checkpoint log record containing Xact table. 5) Resume accepting new update, commit & abort ops  During restart recovery, Analysis Phase begins w the latest checkpoint log record (CPLR):  - initialize TT w CPLR's Xact table. - Initialize DPT to be empty |
| Fuzzy Checkpoint-ing in ARIES | | 1) Let DPT' be the dirty page table & TT' be the Xact table. 2) Write a begin\_checkpoint log record (BCPLR)  3) Write a end\_checkpoint log record (ECPLR) containing DPT' & TT'.  4) Write a special master record containing the LSN of the begin\_checkpoint log record to a known place on stable storage  During restart recovery, Analysis Phase starts w BCPLR identified by the master record: - Assume no log records btw BCPLR & ECPLR  - Initialize TT w BCPLR's Xact table. - Initialize DPT w BCPLR's dirty page table |
| ARIES Analysis Phase w Fuzzy Checkpoint | | Retrieve theBCPLR identified by the master record. Retrieve theECPLR corresponding to BCPLR  - For simplicity, assume no log records between BCPLR & ECPLR (i.e., ECPLR is the next log record after BCPLR)  - Initialize DPT & TT using ECPLR’s contents. - Scan the log in forward dirn (starting from ECPLR) to process each log record r (for Xact T):  - If r is an end log record then remove Tfrom TT  - Else { Add an entry in TT for Tif Tnot in TT. Update **lastLSN** of entry to r’s LSN. Update status of entry to C if ris a commit log record }  - If (ris a redoable log record for page P) & (P not in DPT), then { Create an entry for P in DPT with pageID of entry = P’s pageID and  **recLSN** of entry = r’s LSN |
| ARIES Redo Phase w Fuzzy Checkpoint | | **RedoLSN** = smallest recLSN among all dirty pages in DPT. Let rbe the log record with LSN = **RedoLSN** .  Scan the log in forward dirn starting from r.If (ris a redoable log record) and (condition C is false) then { - fetch page Passociated with r  - If (P’s **pageLSN** < r’s LSN) then { Reapply logged action in rto P. Update P’s **pageLSN** = r’s LSN }  - Else { Update P’s entry in DPT: **recLSN** = P’s **pageLSN** + 1 } } # **recLSN** ≤ r's LSN ≤ P's **pageLSN**.  At the end of Redo Phase, Create end log records for Xacts with status = C in TT & remove their entries from TT.  Condition C: (*P* is not in DPT) or (*P*’s **recLSN** in DPT > r’s LSN). If Condition C = TRUE: update of r already applied to P. r can be ignored |
| ARIES Undo Phase w Fuzzy Checkpoint | | Abort active Xacts at time of crash (loser Xacts). Abort loser Xacts by undoing their actions in reverse order  Initialize L = set of **lastLSNs** (w status = U) from TT. Repeat until L becomes empty: {  Delete largest **lastLSN** from L. Let r = log record corresponding to this **lastLSN**.  If r is an update log record for Xact T on page P, then { - create a CLR r2 for T: r2's **undoNextLSN** = r's **prevLSN**. - update T's entry in TT: **lastLSN** = r2's LSN. - create a DPT entry for P (with **recLSN** = r2's LSN) if P not in DPT. - undo the logged action on page P. - update P's **pageLSN** = r2's LSN. - *Update-L-and-TT* (r's **prevLSN**) }  Else if r is a CLR for Xact T, then { *Update-L-and-TT*(r's **undoNextLSN**) }  Else if r is an abort log record for Xact T, then { *Update-L-and-TT*(r's **prevLSN**) } }  def *Update-L-and-TT*(lsn): { if lsn is not null: {add lsn to L}. Else { create an end log record for T & remove T's entry from TT } } |