### CS3223 AY22/23 SEM 2

github/jovyntls

Notation	Meaning					
r	relational algebra expression					
r	number of tuples in output of r					
r	number of pages in output of r					
b <sub>d</sub>	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 <sup>+</sup> -tree index (i.e., number of pointers to child nodes)					
h	height of B <sup>+</sup> -tree index (i.e., number of levels of internal nodes)					
	$h = \lceil \log_F(\lceil \frac{  R  }{b_i} \rceil) \rceil$ if format-2 index on table $R$					
В	number of available buffer pages					

Data entry formats: 1. actual data record; 2. (k, RID) - fixed length (k, •); 3. (k, RID-list) - e.g. (k, {RID11, RID12})

#### 04.1 SORTING

- clustered index → order of data entries ≈ data records
  - > 1 per relation: format 1 is always clustered

#### External Merge Sort

- sorted run → sorted data records written to a file on disk
- 1. create temporary file  $R_i$  for each B pages of R sorted
- 2. merge: use B-1 pages for input, 1 page for output
- total I/O =  $2N(\lceil \log_{B-1}(N_0) \rceil + 1)$ 
  - 2N to create  $\lceil N/B \rceil$  sorted runs of B pages each
  - merging sorted runs:  $2N \times \lceil \log_{B-1} N_0 \rceil$

#### optimisation with blocked I/O

- seguential I/O read/write in buffer blocks of b pages
- one block (b pages) for output, remaining blocks for input
  - number of runs merged per pass,  $F = \left| \frac{B}{I} \right| 1$
  - number of passes =  $\lceil \log_E(N_0) \rceil + 1$

### Sorting with B<sup>+</sup>-trees

- when sort key is a prefix of the index key of the B<sup>+</sup>-tree
- sequentially scan leaf pages of B<sup>+</sup>-tree
- · for Format-2/3, use RID to retrieve data records

### 04.2 SELECTION: $\sigma_n(R)$

- $\sigma_p(R)$  selects rows from relation R satisfying predicate p
- selectivity of an access path → number of index & data pages retrieved (more selective = fewer pages retrieved)
- covering index I for Q → if all attributes referenced in Q are part of the key of or include columns of I (index-only plan: no RID lookup)

### **Matching Predicates**

- term  $\rightarrow$  of form R.A op c or  $R.A_i$  op  $R.A_i$
- **conjunct**  $\rightarrow > 1$  terms connected by  $\vee$  (**disjunctive**: > 1)
- CNF predicate → one or more conjuncts connected by ∧



#### B<sup>+</sup>-tree matching predicates

- for index  $I = (K_1, K_2, \dots, K_n)$  and non-disjunctive CNF predicate p, I matches p if p is of the form
- $(K_1 = c_1) \wedge \cdots \wedge (K_{i-1} = c_{i-1}) \wedge (K_i \ op_i \ c_i), \ i \in [1, n]$ zero or more equality predicates
- · matching index: matching records are in contiguous pages

#### Hash index matching predicates • hash index I matches p if p is of form

 $(K_1 = c_1) \wedge (K_2 = c_2) \wedge \cdots \wedge (K_n = C_n)$ 

#### **Primary/Covered Conjuncts**

- **primary conjuncts**  $\rightarrow$  subset of conjuncts that I matches • e.g.  $p = (A \ge 18) \land (A \le 20) \land (W=65)$  for I = (A,W,H)
- **covered conjuncts**  $\rightarrow$  attribute appears in the key of Iprimary conjuncts ⊆ covered conjuncts

#### **Cost of Evaluation**

let p' = primary conjuncts of p.  $p_c$  = covered conjuncts of p

#### B<sup>+</sup>-tree index evaluation of p

- 1. navigate internal nodes to find first leaf page  $\begin{aligned} & \operatorname{cost}_{\mathsf{internal}} = \lceil \log_F(\lceil \frac{||R||}{b_{d \text{ or } i}} \rceil) \rceil & \text{ for format-1/otherwise} \\ 2. & \operatorname{scan leaf pages to access all qualifying data entries} \end{aligned}$
- cost<sub>leaf</sub> =  $\lceil \frac{||\sigma_{p'}(R)||}{b_{d \, or \, i}} \rceil$  for format-1/otherwise 3. retrieve qualified data records via RID lookups
- $cost_{RID} = ||\sigma_{p_n}(R)||$  or 0 if I is covering or format-1 • reduce cost with clustered data records (sort RIDs):  $\lceil \frac{||\sigma_{p_c}(R)||}{h} \rceil \leq \mathsf{cost}_{RID} \leq \min\{||\sigma_{p_c}(R)||, |R|\}$

### hash index evaluation of p

- format-1: cost to retrieve data records  $\geq \lceil \frac{||\sigma_{p'}(R)||}{b_{J}} \rceil$
- format-2: cost to retrieve data entries  $\geq \lceil \frac{||\sigma_{p'}(R)||}{b_i} \rceil$
- $\text{cost to retrieve data records} = \begin{cases} 0 & \text{if I is a covering index,} \\ ||\sigma_{p'}(R)|| & \text{otherwise} \end{cases}$

### **05.1 PROJECTION** $\pi_{A_1,\ldots,A_m}(R)$

- $\pi_L(R)$  eliminates duplicates,  $\pi_L^*(R)$  preserves duplicates
- can index scan if index contains the attributes as a prefix

### Sort-based approach

#### cost analysis

- 1. extract attributes:  $|R| \operatorname{scan} + |\pi_L^*(R)|$  output temp result
- 2. sort records:  $2|\pi_L^*(R)|(\log_m(N_0)+1)$
- 3. remove duplicates:  $|\pi_L^*(R)|$  to scan records

#### optimised sort-based approach

- 1. create sorted runs with projected attributes only
- 2. merge sorted runs and remove duplicates
- if  $B > \sqrt{|\pi_I^*(R)|}$ , same I/O cost as hash-based approach
  - $N_0 = \lfloor \frac{|R|}{B} \rfloor \approx \sqrt{|\pi_L^*(R)|}$  initial sorted runs  $\log_{B-1}(N_0) \approx 1$  merge passes

### Hash-based approach

- 1. partitioning phase: hash each tuple  $t \in R$  to some  $R_i$ 
  - one buffer for input, (B-1) buffers for output
- for each t: project attributes to form t', hash h(t') to one output buffer, flush output buffer to disk when full
- 2. **duplicate elimination** from each  $\pi_{\tau}^*(R_i)$ 
  - for each  $R_i$ : initialise in-mem hash table, hash each  $t \in R_i$  to bucket  $B_i$  with  $h' \neq h$ , insert if  $t \notin B_i$
  - · write tuples in hash table to results
- I/O cost (no partition overflow):  $|R| + 2|\pi_{\tau}^{*}(R)|$ 
  - partitioning cost:  $|R| + |\pi_L^*(R)|$ • duplicate elimination cost:  $|\pi_I^*(R)|$
- · partition overflow: recursively apply partitioning
  - to avoid, B > size of hash table for  $R_i = \frac{|\pi_L^*(R)|}{B_1} \times f$ 
    - approximately  $B > \sqrt{f \times |\pi_{I}^{*}(R)|}$

### **05.2 JOIN** $R\bowtie_{\theta} S$

R = outer relation (smaller relation); S = inner relation

- ! for format-2 index. add cost of retrieving record
- tuple-based nested loop join:  $|R| + |R| \times |S|$
- page-based nested loop join:  $|R| + |R| \times |S|$
- block nested loop join:  $|R| + (\lceil \frac{|R|}{B-2} \rceil \times |S|), \ |R| \leq |S|$
- 1 page output, 1 page input, (B-2) pages to read R
- for each (B-2) pages of R: for each  $P_S$  of S: check r,s
- index nested loop join: for joining  $R.A_i = S.B_i$

$$|R| + ||R|| \times \left(\log_F\left(\lceil \frac{||S||}{b_d} \rceil\right) + \lceil \frac{||S||}{b_d ||\pi_{B_j}(S)||} \rceil\right)$$

#### sort-merge join

- sort R & S:  $2|R|(\log_m(N_R) + 1) + 2|S|(\log_m(N_S) + 1)$
- merge cost: |R| + |S| (worst case  $|R| + |R| \times |S|$ ) optimised sort-merge join
- merge sorted runs until B > N(R, i) + N(S, i); then join
- 3(|R| + |S|) = 2 + 1 (for initial sorted runs + merging)
  - if  $B > \sqrt{2|S|}$ , one pass to merge initial sorted runs

#### Grace hash ioin

for build relation R and probe relation S.

- 1. **partition** R and S into k partitions each, k = B 1
  - $\pi_A(R_i) \cap \pi_B(S_i) = \emptyset \quad \forall R_i, S_i, i \neq i$ •  $R = R_1 \cup R_2 \cup \cdots \cup R_k$ ,  $t \in R_i \iff h(t.A) = i$
- 2. **probing phase**: hash  $r \in R_i$  with h'(r,A) to table T;  $\forall s \in S_i, r \in \text{bucket } h'(s.B)$ : output (r, s) if match
  - $R \bowtie_{R} A = S R S = (R_1 \bowtie S_1) \cup \cdots \cup (R_k \bowtie S_k)$
- partition overflow if R<sub>i</sub> cannot fit in memory: recurse
- I/O cost: 3(|R| + |S|) (no partition overflow)
- $B>\frac{f\times |R|}{B-1}+2$  (input & output buffer)  $\approx B>\sqrt{f\times |R|}$  during probing, B> size of each partition +2

### adapting join algorithms

- multiple equality-join conditions:  $(R.A=S.A) \land (R.B=S.B)$ 
  - · index nested loop join: use index on some/all join attribs
  - sort-merge join: sort on combination of attributes
- inequality-join conditions: (R.A < S.A)
  - index nested loop join: requires B<sup>+</sup>-tree index
- · not applicable: sort-merge join, hash-based joins
- set operations

materialised evaluation

- intersection:  $R(A,B) \cap S(A,B) = \pi_{R.A,R.B}(R \bowtie_p S)$
- cross product:  $R \times S = R \bowtie_{true} S$
- union/difference: duplicate elimination/slightly modified

### 06. QUERY EVALUATION

- · aggregation: maintain running information while table scan
- · index scan if there is a covering index for the guery
- group-by: sort/hash to group by attributes then aggregate · if group-by attributes are a B+tree prefix, just aggregate
- · evaluates bottom-up; materialise intermediate results to disk
- x incurs I/O ✓ simple implementation ✓ less memory pipelined evaluation (top-down, demand-driven)
- interleaved execution of operators pass output directly to parent operator - can switch execution to where it is needed
- blocking operator: can't produce output until all input tuples received (grace hash & sort-merge join, external mergesort) hybrid: pipelined evaluation with partial materialisation

### • materialise if repeatedly scanned (e.g. nested loop join)

query: > 1 logical plans: implemented by > 1 physical plans query plan trees

- **linear**  $\rightarrow \geq$  1 operand per join oper-
- ation is a base relation (else **bushy**)

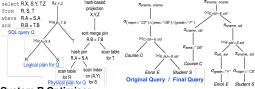
   **left-deep** → every right join operand is a base relation



#### query optimisation

query plans

- binary operators (⋈, ×) are commutative & associative
  - · push selection and projection to operands first
- DP query plan enumeration: use all optimal sub-plans to build overall plan (single-relation  $\Rightarrow$  two-relation  $\Rightarrow \dots$ )



#### System R Optimiser

- enumerate only left-deep query plans; avoid cross-product query plans; consider early selections and projections
- DP + sort order  $o_i$  of query plan output:  $optPlan(S_i, o_i)$

#### cost estimation

- · estimation assumptions
- 1. uniformity of distribn of attr values
- 2. independence for distribut of values in different attrs
- 3. inclusion for  $R \bowtie_{R,A=S,B} S$ , if

$$||\pi_A(S)|| \le ||\pi_B(S)||$$
, then  $\pi_A(R) \subseteq \pi_B(S)$   
 $\Rightarrow$  every  $R$  tuple joins with some  $S$  tuple

- size estimation for query  $q = \sigma_p(e), \quad p = t_1 \wedge \cdots \wedge t_n$ 
  - selectivity factor → fraction of tuples satisfying term
    - aka reduction factor,  $rf(t_i) = \frac{||\sigma_t(e)||}{||e||}$
  - $||q|| \approx ||e|| \times \prod_{i=1}^{n} rf(t_i)$
  - · join selectivity:  $rf(R.A = S.B) \approx \frac{1}{\max\{||\pi_A(R)||, ||\pi_B(S)||\}}$
- · histogram estimation
  - equiwidth → ≈equal number of values per bucket
  - equidepth → ≈equal number of tuples per bucket
  - with MCV: keep a k/v pair of value/#tuples

### 07. TRANSACTION MANAGEMENT

- to ensure ACID properties of transactions →
- 1. atomicity either all or none of the actions happen
- 2. consistency if each txn is consistent, and the DB starts consistent, then the DB ends up consistent
- 3. isolation execution of one txn is isolated from other txn
- 4. durability if txn commits, its effects persist
- view equivalent → same reads-from and final write view serialisable → view equiv to some serial schedule
- conflict → at least 1 write + different txns + same object conflict equivalent → all pairs of conflicting actions are
- ordered in the same way conflict serialisable → conflict equivalent to a serial sched
- 1. acyclic conflict serialisability graph (node: committed txn, edge: precedes and conflicts with any action)
- 2. conflict serialisable ⇒ view serialisable
- 3. view serialisable + no blind writes ⇒ conflict serialisable
- blind write → did not read before write

#### anomalies arise due to conflicting actions

- · dirty read due to WR conflicts
- unrepeatable read due to RW conflict  $(R_1, W_2, R_1)$
- · lost update due to WW conflict
- phantom read re-executing a query on a search condition gives different results (prevent by predicate/index locking)

#### recovery

- cascading abort  $\rightarrow$  if  $T_1$  reads from  $T_2$ ,  $T_1$  must abort when  $T_2$  aborts (for correctness)
- ullet recoverable ullet if T reads from T', then T commits after T' guarantees that committed txns will not be aborted
- cascadeless  $\rightarrow$  whenever  $T_i$  reads from  $T_j$ ,  $Commit_j$  must precede this action
- all values read are produced by a committed transaction
- **before-images**: log before action & restore (must be strict)
- strict  $\to$  for every  $W_i(O)$  in S,O is not read/written by another txn until  $T_i$  either aborts or commits
- strict schedule  $\Rightarrow$  cascadeless  $\Rightarrow$  recoverable

### **08. CONCURRENCY CONTROL**

## Lock-based Concurrency Control

### 2PL (Two Phase Locking)

- may release locks any time
- once a txn releases a lock, it cannot request any more locks
   growing/shrinking phase: before/after releasing 1<sup>st</sup> lock
- · prevents all anomalies, including phantom read

#### Strict 2PL

- Strict 2PL → txn must hold locks until it commits/aborts
- 2PL ⇒ conflict serialisable
- strict 2PL ⇒ strict & conflict serialisable

### **Lock Management**

#### deadlocks

- · deadlock detection: waits-for graph (WFG)
  - nodes represent active txns
  - edge  $T_i \to T_i$  if  $T_i$  is waiting for  $T_i$  to release a lock
  - WFG has a cycle ⇒ deadlock
    - abort one transaction and its edges from WFG
- deadlock prevention: older = higher priority
  - wait-die policy → lower-priority aborts instead of waiting
     less aggressive; younger txns may keep aborting
  - wound-wait policy → (preemptive) higher- aborts lower-
  - wound-wait policy → (preemptive) nigner- aborts lower preemptive can abort another txn

· restarted txn uses original timestamp to avoid starvation

#### lock conversion

- increases concurrency; only in the growing phase
- lock upgrade,  $UG_i(A)$ : allowed if no other txn is holding a shared lock on A and  $T_i$  has not yet released any lock
  - ensures serialisable schedule
- lock downgrade,  $DG_i(A)$ : allowed if  $T_i$  has not modified A and has not released any lock

#### **ANSI SQL Isolation Levels**

	Dirty	Unrepeatable	Phantom
Isolation Level	Read	Read	Read
READ UNCOMMITTED	possible	possible	possible
READ COMMITTED	not possible	possible	possible
REPEATABLE READ	not possible	not possible	possible
SERIALIZABLE	not possible	not possible	not possible

	Isolation	Write	Read	Predicate
Degree	level	Locks	Locks	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 → can be released before commit/abort
- long-duration lock → held until txn commits/aborts

#### **Locking Granularity**

- (coarsest/most granular) database > relation > page > tuple
- $\frac{\text{multi-granular}}{\text{granular}}$  lock  $\rightarrow$  can request different granularity
- if T holds lock mode M on data granule D, then T implicitly holds M on data granules finer than D

#### I-lock (intention)

- before acquiring any S-/X-lock on G, must acquire I-locks on granules coarser than G in a top-down manner
- can be shared with other I-locks
- ullet × limited concurrency: S-lock is incompatible with I-lock

#### IS- and IX-lock (intention shared/exclusive)

- · acquire locks top-down, release locks bottom-up
  - to obtain S or IS lock: must hold IS or IX lock on parent
  - to obtain X or IX lock: must hold IX lock on parent

Lock compatability matrix					Lock compatability matrix					
Lock	Lock Held					Lock	Lock Held			
Requested	-	IS	IX	S	Х	Requested	-		S	Х
IS	V	V	√	V	×		<u> </u>		- V	×
IX	V	V	V	×	×	'	V.	V	^.	^
S	1/	1/	×	1/	×	5	\ √	×	√	×
Х	1/	×	×	×	×	Х		×	×	×

# 09. MULTIVERSION CONCURRENCY CONTROL (MVCC)

- · maintain multiple versions of each object
  - $W_i(O)$  creates new version.  $R_i(O)$  reads some version
- $\checkmark$  read-only txns not blocked by update txns  $\checkmark$  update txns not blocked by read-only txns  $\checkmark$  read-only txns never aborted
- $\frac{\text{multi-version}}{\text{multi-version}}$  schedule  $\rightarrow$  read can return any version
- $\frac{\text{mono-version}}{\text{mono-version}}$   $\rightarrow$  always reads most recent version
- multi-version view equivalent ,  $S \equiv_{mv} S' \to$  same set of read-from relationships;  $R_i(x_j) \in S \iff R_i(x_j) \in S'$
- final write doesn't matter (concept in monoversion only)
- multi-version view serialisable (MVSS) → exists a serial mono-version schedule that is multi-version view equivalent
  - mono-version view serialisable ⇒ MVSS
  - VSS  $\subseteq$  MVSS; VSS  $\Rightarrow$  MVSS; MVSS  $\Rightarrow$  VSS

### **Snapshot Isolation**

- each  ${\rm txn}\ T$  sees a snapshot of the DB comprising updates by transactions that committed before T starts
- $\frac{\text{concurrent}}{\text{concurrent}}$  txns  $\rightarrow$  overlap, defined by start(T)/commit(T)
- $\operatorname{protocol}$ :  $O_i$  is more recent if  $\operatorname{commit}(T_i)$  is later
- $W_i(O)$  creates version i of O
- $R_i(O)$  reads either its latest  $W_i(O)$  or the latest version of O created by a txn that committed before  $\mathrm{start}(T_i)$

- concurrent update property: if multiple concurrent txn update the same object, only 1 commits (ensure serialisable)

  - FUW (first updater wins): acquire X-lock to update
    - T proceeds iff all concurrent T' (previously holding the X-lock) aborts and O has not been updated by any concurrent txn. / else abort T
- · garbage collection: if not read by any (active/future) txn
  - delete  $O_i$  if there exists a newer  $O_j$  (commit $(T_i)$  < commit $(T_j)$ ) such that for every active  $\operatorname{txn} T_k$  that started after commit $(T_i)$ , we have  $\operatorname{commit}(T_i)$  <  $\operatorname{start}(T_k)$
- performance: 
  √ similar to READ\_COMMITTED but without lost update or unrepeatable read anomalies
  - × 

    serialisability (some non-serialisable executions)
    - write skew anomaly:  $R_1(x_0), R_2(y_0), W_1(y_1), W_2(x_2)$
    - read-only txn anomaly:  $T_3 \xrightarrow{rw} T_2 \xrightarrow{rw} T_1 \xrightarrow{wr} T_3$
  - × does not guarantee serialisability

#### Serialisable Snapshot Isolation (SSI)

- ensures MVSS
- detect  $T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k$  and abort one of  $T_i, T_j, T_k$
- keeps track of rw dependencies; possible false positives

### transactional dependencies: www, wr, rw

- **immediate successor**  $\rightarrow$  no W(x) commits betw commits
- dependency serialisation graph, DSG
  - · nodes: (committed) transactions
  - edges: transactional dependencies, e.g.  $T_i \xrightarrow{wr} T_j$ 
    - --→/→ for concurrent/non-concurrent
- if S is a SI schedule that is not MVSS, then
  - there is at least one cycle in DSG(S)
  - for each cycle in DSG(S),  $\exists T_i, T_i, T_k$  such that
    - $T_i \xrightarrow{rw} T_i \xrightarrow{rw} T_k$  exists
    - $T_i$  and  $T_k$  may be same txn (eg. write-skew anomaly)

### 10. CRASH RECOVERY

- · recovery manager guarantees atomicity and durability
  - · undo: preserve atomicity (remove effects of aborts)
  - · redo: durability (re-install effects of commits)
- steal policy → can write dirty page to disk before commit
- force policy  $\rightarrow$  must write all dirty pages to disk at commit

	Force	No-torce	no sicai. may ran out or
Steal	undo & no redo	undo & redo	buffer pages
No-steal	no undo & no redo	no undo & redo	force: incur random I/O

### **ARIES Recovery Algorithm**

• steal; no-force; assumes strict 2PL for concurrency control

#### data structures

- log file sequential file of records in stable storage
- transaction table (TT) 1 entry for each active txn
- (txn ID, last LSN, C/U status)
- dirty page table (DPT) 1 entry per dirty page in buffer pool
   (pageID, recLSN) = earliest log record that dirtied page
- · log records: (type, txn ID, prevLSN, other info)
  - update (! redoable): pageID, before-image, after-image
- compensation (CLR): (! redoable) pageID, undoNextLSN (ULR's prevLSN), action to undo
- when update described by ULR is undone
- ullet commit: force-write all records  $\leq r$  to stable storage

- · flush all log records for transaction to disk
- · abort: create when txn is to be aborted
- end: create when all processing for T is completed
- checkpoint: speed up recovery (scan from checkpoint)

#### implementing abort

- - each DB page contains pageLSN (LSN of latest update)
- before flushing page P, ensure all log records ≤ P.pageLSN have been flushed to disk

#### implementing commit

- force-at-commit protocol → do not commit txn until the after-images of all its updated records are in stable storage
  - commit LR; txn is considered committed if its commit log record has been written to stable storage

#### implementing restart

- 1. analysis phase TT (active txns) & DPT (superset of dirty)
- 1.1. initialise TT & DPT (retrieve ECPLR from BCPLR)
- 1.2. for each r in log file in forward direction/chronological
  - $\bullet \ \, \text{if } \textit{end LR}, \, \text{remove } T \text{ from TT; continue}\\$
  - if redoable LR for P and P not in DPT:
  - create P's entry in DPT with recLSN=r.LSN
  - add or update entry for T in TT: lastLSN = r.LSN
     if commit LR: update status=C
- 2. redo phase restore DB to state at time of crash
- 2.1. start from redoLSN = smallest recLSN in DPT
- 2.2. scan in forward direction for all redoable LR
  - i. if *not redoable* or NOT optimisation cond: continue ii. if P.pageLSN < r.LSN: (r has not been installed)
    - reapply logged action in r to P
    - update P.pageLSN = r.LSN
  - iii. (optimisation) else: recLSN ≤ r.LSN ≤ P.pageLSN
    - update P in DPT: recLSN=P.pageLSN+1
- 2.3. create *end LR* for all status=C in TT: remove entry
- optimisation cond: (P ∉ DPT) or (DPT P.recLSN > r.LSN)
- update of r has already been applied to P
- 3. **undo phase** abort **loser** txns (active at crash) in *reverse*
- 3.1. initialise L = set of lastLSN (status=U) from TT
   update-L-and-TT(LSN) := if LSN is not null, add to
- L; else create end LR for T and remove T from TT
- 3.2. while  $L \neq \emptyset$ :
  - i. r = largest lastLSN in L; delete r from L
  - ii. if r is update LR for T on P:create CLR r<sub>2</sub> with r<sub>2</sub>.undoNextLSN=r.prevLSN
  - update TT: T.lastLSN= $r_2$ .LSN
  - undo logged action and update P.pageLSN= $r_2$ .LSN
  - update-L-and-TT(r.prevLSN)
  - iii. else if r is CLR: update-L-and-TT(r.undoNextLSN) iv. else r is abort LR: update-L-and-TT(r.prevLSN)

#### normal transaction processing

- TT: create new or update existing entry for T (lastLSN)
  - when T commits, update status=C
- ullet when end log record is generated, remove T's entry
- P is updated: update P.pageLSN = r.LSN
- P is updated & not in DPT: create entry (recLSN=r.LSN)
- when P is flushed to disk: remove P's **DPT** entry