Jin Wei CS3223 Finals

Disk Access Time

- Command Processing Time (negligible)
- Seek Time: moving arms to position disk head on track
  - Average seek time: 5-6ms
- Rotational Delay: waiting for block to rotate under head
  - Depends on RPM; average rotation delay = time for $1/2$ revolution
  - x RPM => $(60'000)/x$ ms for 1 revs
- Transfer Time: time to move data to/from data surface
  - $n * \frac{\text{time for one revolution}}{\text{number of sectors per track}}$
- Access time = seek time + rotational delay + transfer time
- Response time = queueing delay + access time

B+ Tree

- Leaf nodes are doubly-linked
- Internal nodes $(p_0, k_1, p_1, \ldots, k_n, p_n)$
- Order d: non-root node $[d, 2d]$; root node $[1, 2d]$
- Maximum order: $2d$(#bytes of key) $+$ $(2d+1)$(#bytes of page address) $\leq$ #bytes of page size. Solve for d
- Minimum number of leaf nodes: $2*(d+1)^{i-1}$; Maximum: $(2d+1)^i$; i is levels of internal

Sorting

- Create $N_0 = \lceil N/B \rceil$ sorted runs, N pages
- Merging: $B - 1$ pages for input, 1 for output
- Total I/O = $2N * (\lceil \log_{B-1}(N_0) \rceil + 1)$
- Optimised Merging: $\lfloor \frac{B-b}{b} \rfloor$ for input, b for output
- Total IO = $2N * (\lceil \log_F(N_0) \rceil + 1)$, $F = \lfloor (B - b_{output})/b_{input} \rfloor$
- Sequential I/O: $\lceil N/b \rceil * (\text{\#passes}) * ((\text{seek + rotate}) + b * (\text{transfer}))$

Selection

- Full Table Scan; Index Scan; Index Intersection (combination), + RID lookup
- Goal: reduce number of index & data pages retrieved
- Covering Index I for query Q if $Q \subseteq I$; no RID lookup; index-only plan
- Term: A op B; Conjuncts: terms connected by OR; CNF: conjuncts joined by AND
- B+ Index $I = (K_1, K_2, \ldots)$ matches Predicate p if $(K_1, \ldots, K_i)$ is prefix of I **AND** only K_i can be non-equality
- Hash Index I match if **equal** for **every** attribute
- Subset that matches is **primary conjuncts**; Subset that **covered** is **covered conjuncts**
- $||r||$: #tuples, $|r|$: #pages, b_d: #data records (entire tuple), b_i: #data entries
- B+ tree cost = (height of internal nodes) + (scan leaf pages) + (RID lookup)
  - Can reduce I/O cost of lookup by sorting
- Hash cost = (retrieve data entries) + (retrieve data records)
- Plans: Full Table Scan, Index Scan, Intersections, Unions
  - Primary: can traverse tree / hash (if not: check all leaf)
  - Covered: no RID lookup
  - Intersection: (retrieve leaf entries for both) + Grace-Hash Join + (retrieve data records)

Projection

- Remove unwanted attributes, eliminate dupes
- Sort: Extract -> Sort -> Remove Dupes (linear scan)
- Optimised Sort: Create Sorted Runs with attributes L (read N pages, write $N * \frac{|L|}{|\text{\#no of attrs}|}$) -> Merge Sorted Runs + Remove Dupes
  - If $B > \sqrt{|\pi^*_L(R)|}$, $N_0 \approx \sqrt{|\pi^*_L(R)|}$, similar as Hash
- Hash: Partition into $B - 1$ partitions using hash function -> Remove Dupes in partitions -> Union partitions
  - Partition phase: 1 for input, $B - 1$ for output -> remove unwanted attributes -> hash -> flush when buffer is full
  - Dupe Elim phase: Use in-memory hash table with h'
  - Partition overflow problem -> recursively apply partition until can fit in-memory
  - Approximately $B > \sqrt{f * |\pi^*_L(R)|}$ to avoid partition overflow
  - If no partition overflow: Partition: $|R| + |\pi^*_L(R)|$, Dupe Elim: $|\pi^*_L(R)|$
- Indexes: Use index scan; If B+ & wanted attributes is prefix: already sorted, so compare adjacent

Nested-Loop Join

- Smaller should be outer (R)
- Tuple-Based: For each outer tuple: check each inner tuple $|R| + ||R|| * |S|$
- Page-Based: For each outer page: check each inner page: compare tuples within these pages $|R| + |R| * |S|$ (3 buffer pages, 2 input, 1 output)
- Block-Based: read in $B - 2$ sequential pages of R, read in page of S one-by-one
  - $|R| + (\lceil \frac{|R|}{B-2} \rceil * |S|)$
- Index-Based: for each tuple in R: search S's index
  - Assuming uniform distribution: $|R| + ||R|| * J$, J = tuple search cost
- Minimum for *any* join: cost of $|R| + |S|$ with $|R| + 1 + 1$ buffer pages, store entire |R| in memory

Sort-Merge Join

- Sort both R and S, then join
- Each tuple in R partition merges with all tuples in matching S-partition
- Advance pointer pointing to smaller tuple; rewind S-pointer as necessary
- I/O cost = (Cost to sort R) + (Cost to sort S) + Merging cost; (merging cost = |R| + |S| if no rewind, |R| + ||R|| * |S| if rewind everytime)
- Optimised (partial sorting): if $B > N(R, i) + N(S, j)$, stop sorting, $N(R,i) = $ total number of sorted runs of R after pass i
  - I/O cost if $B > \sqrt{2|S|}$, $3 * (|R| + |S|)$ => 2 for creating initial sorted runs (one pass is sufficient), 1 for merge
  - else $3 * (|R| + |S|) + c * |R| + d * |S|$, where c and d is number of merge passes for R, S

Grace Hash Join (no Hybrid Hash Join)

- Split R and S into k partitions each, join these k partitions together in probing phase
  - read R_i to build hash table (build relation) - pick smaller for build (must fit in-memory)
  - read S_i to probe hash table (probe relation)
- partitioning phase: 1 input buffer, k hash buffers, once full, flush into page on disk
- probing phase: 1 input buffer, 1 output buffer, 1 hash table; use different h'(.) and build a hash table for each partition, then, probe with S, if match, add to output buffer
  - build R_1, probe S_1, build $R_2, \ldots$
  - once output buffer is full, flush (don't flush between partitions)
- set $k = B - 1$ to minimise partition sizes
- assuming uniform hashing distribution:
  - size of each partition R_i $\frac{|R|}{B-1}$
  - size of hash table for R_i $\frac{f*|R|}{B-1}$, f is fudge factor
  - during probing phase, $B > \frac{f*|R|}{B-1} + 2$, one each for input & output
  - approximately, $B > \sqrt{f * |R|}$
- Partition Overflow Problem: hash table doesn't fit in memory, recursively partition overflowed partitions
- I/O cost = 3(|R| + |S|) if no partition overflow
- I/O cost = (c * 2 + 1)(|R| + |S|) where c is number of partitioning phases

Join Conditions

- Multiple Equality-Join conditions $(R.A = S.A) and (R.B = S.B)$
  - Index Nested Loop Join: use index on all attrs; or only on primary conjuncts, then data lookup uncovered conjuncts
  - Sort-Merge Join: sort on combinations
  - Other algorithms are unchanged
- Inequality-Join conditions $(R.A < S.A)$
  - Index Nested Loop Join: requires B+ tree index
  - Sort-Merge Join: N/A (becomes nested loop join)
  - Hash-Based Join: N/A (becomes nested loop join)
  - Other algorithms are unchanged

Operations

- Aggregation: scan table while maintaining running information
- Group-by aggregation:
  - sort on grouping attributes, scan sorted relation to compute aggregate
  - build hash table on grouping attributes, maintain (group-value, running-information)
- Index Optimisation: if have covering index, use it; avoids need for sorting

Query Evaluation

- Materialised (temporary table) Evaluation - waits for everything to be done
  - operator is evaluated only when its operands are completely evaluated or materialised
  - intermediate results are materialised to disk
  - may reduce number of rows
- Pipelined Evaluation - requires more memory
  - output produced by operator is passed directly to parent (interleaves execution of operators)
  - operator O is *blocking operator* if it requires full input before it can continue (e.g. external merge sort, sort-merge join, grace-hash join)
  - Iterator Interface: top-down, demand-driven (parent calls `getNext()` from child)

Query Plans

- Query has many *equivalent* logical query plans, which has many physical query plans.
- Want to avoid **BAD** plans, not pick the *best*
  - Ideally minimise size of intermediate results
- join-plan notation
  - nested-loop: left is outer, right is inner
  - sort-merge: left is outer, right is inner
  - hash-join: left is probe, right is build

Relational Algebra Rules

1. Commutativity
   1. $R \times S \equiv S \times R$
   2. $R \bowtie S \equiv S \bowtie R$
2. Associativity
   1. $(R \times S) \times T \equiv R \times (S \times T)$
   2. $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$
3. Idempotence
   1. $\pi_{L'}(\pi_L(R)) \equiv \pi_{L'}$ if $L' \subseteq L \subseteq attrs(R)$
   2. $\sigma_{p_1}(\sigma_{p_2}(R)) \equiv \sigma_{p_1 \wedge p_2}$
   3. $\pi_L(\sigma_p(R)) \equiv \pi_L(\sigma_p(\pi_{L \cup attrs(p)}(R)))$
4. Commutating Selection w/ Binary Ops - pushes operations down to leaf node
   1. $\sigma_p(R \times S) \equiv \sigma_p(R) \times S$ if $attrs(p) \subseteq attrs(R)$
   2. $\sigma_p(R \bowtie_{p'} S) \equiv \sigma_p(R) \bowtie_{p'} S$ if $attrs(p) \subseteq attrs(R)$
   3. $\sigma_p(R \cup S) \equiv \sigma_p(R) \cup S$
5. Commutating Projection w/ Binary Ops
   - let $L = L_R \cup L_s$, where $L_R \subseteq attrs(R)$ and $L_S \subseteq attrs(S)$
   1. $\pi_L(R \times S) \equiv \pi_{L_R}(R) \times \pi_{L_S}(S)$
   2. $\pi_L(R \bowtie_p S) \equiv \pi_{L_R}(R) \bowtie_p \pi_{L_S}(S)$ if $attrs(p) \cap attrs(R) \subseteq L_R$ and L_S
   3. $\pi_L(R \cup S) \equiv \pi_L(R) \cup \pi_L(S)$

Query Optimisation

1. Search space
2. Plan enumeration - how to enumerate search space
3. Cost model

Query Plan Trees

- **Linear** if at least one operand for *each join* is base relation; otherwise it's **bushy**
- **Left-deep** if *each right join* is base relation
- **Right-deep** if *each left join* is base relation
- Use DP: compute optimal cost $optPlan(S_i)$ for each subset of relations being joined
  - i.e. for $R \bowtie S \bowtie T$, consider $optPlan(\{R, S, T\}) = \min\{optPlan(R) + optPlan(\{S, T\}), \ldots\}$
  - Enhanced DP: might be worth using sub-optimal if produces sorted order, $optPlan(S_i, o_i)$, where o_i captures attrs sorted (or null)

Cost Estimation

- Uniformity Assumption: uniform distribution
- Independence Assumption: independent distribution in different attrs
- Inclusion Assumption: assumes all $r \in R$ maps to some $s \in S$, if $||\pi_A(R)|| \leq ||\pi_B(S)||$

Size Estimation

For $q = \sigma_p(e), p = t_1 \wedge \ldots t_n, e = R_1 \times \ldots R_n$

- $||q|| \approx ||e|| \times \Pi_{i=1}^n(rf(t_i))$, reduction / selectivity factor $rf(t_i) = \frac{||\sigma_{t_i}(e)||}{||e||}$
- $rf(A = c) \approx \frac{1}{||\pi_A(R)||}$ by uniform
- $rf(R.A = S.B) \approx \frac{1}{max\{||\pi_A(R)||, ||\pi_B(S)||\}}$ by inclusion

Estimation w/ Histogram

- Equiwidth: each bucket has (almost) equal number of **values**
- Equidepth (* better): each bucket has (almost) equal number of **tuples**; sub-ranges might overlap (can however, e.g. 1-6)
- MCV: separately keep track of top-k

Transaction Properties

- Atomicity: all or nothing (by recovery manager)
- Consistency: if each Xact is consistent, and DB starts consistent, ends consistent
- Isolation: execution of Xacts are isolated (by concurrency control manager)
- Durability: once commit, persist (by recovery manager)

Transaction

- T_j reads O from T_i in a schedule S if last write action on O before $R_j(O)$ is $W_i(O)$
- T_j reads from T_i if T_j read some object from T_i
- T_i performs final write on O in a schedule S if last write action on O in S is $W_i(O)$
  - determines final state

Schedules

- VE if same read-froms & same final-writes
- VSS if VE to some serial schedule
  - VSG - (T_j, T_i) if T_i reads-from T_j, or T_i does final-write
  - VSG cyclic => *not* VSS
  - VSG acyclic & (serial schedule from topo-sort is VE to S) => VSS
- Conflicting Action if
  - at least one of them is **write action**
  - and actions are from different transactions
- Anomalies from Interleaving
  - dirty read problem (WR)
    - $W_1(x), R_2(x)$
  - unrepeatable read problem (RW) => same row, different value
    - $R_1(x), W_2(x), C_2, R_1(x)$
  - lost update problem (WW)
    - $R_1(x), R_2(x), W_1(x), W_2(x)$
- CE: every pair of conflicting actions are ordered in the same way
- CSS: CE to some serial schedule (CSS => VSS) ("serialisable" = CSS)
  - CSS $\iff$ CSG is acyclic
- blind write: Xact no read before it writes
  - VSS & no blind writes => CSS
- cascading abort: if T_i reads from T_j and T_j aborts, T_i must abort too for correctness
- Recoverable Schedule (essential): for every Xact T that commits in S, T must commit after T' if T reads from T
- Cascadeless Schedule: can only read from committed Xacts
- Strict Schedule (can use before-image): for every $W_i(O)$, O is not read or written by another Xact until T_i abort / commit
- Strict $\subseteq$ Cascadeless $\subseteq$ Recoverable

Transaction Scheduler

for each input action (read, write, commit, abort):

1. output action to scheduler (perform the action)
2. postpone the action by blocking Xact
3. reject and abort Xact

Lock-based Concurrency Control

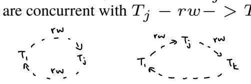
- if lock request not granted, Xact is blocked, Xact is added to O's *request queue*
- 2PL => CSS: once release a lock, no more request
- Strict 2PL => strict & CSS: Xact must hold onto lock until commit / abort
- Wait-For-Graph: $T_i \rightarrow T_j$ if T_i waiting for T_j (must remove edge)
- Timeout mechanism: when Xact start, start timer, if timeout, assume deadlock
- Deadlock Prevention - older Xact has higher priority (not restarted on kill)
  - suppose T_i requests a lock held by T_j (Higher-Lower)
  - wait-die: T_i wait for T_j, T_i suicide => may starve
  - wound-wait: kill T_j, T_i wait for T_j
  - if T_j dies, T_i still waits
- Lock upgrade: similar to acquiring X
- Lock downgrade: has not modified O, has not released any lock
- Phantom Read Problem: re-executes query for a search condition but obtains *different rows* due to another recently committed transaction
  - can't lock row if don't exist => perform *predicate locking* instead, but use **index locking** in practice for efficiency
- Isolation Level (Dirty Read, Unrepeatable Read, Phantom Read, Write, Read, Predicate)
- Read Uncommitted: Y, Y, Y, L, N, N
- * Read Committed: N, Y, Y, L, S, N
- Repeatable Read: N, N, Y, L, L, N
- Serialisable: N, N, N, L, L, Y
- Granularity: DB, Relation, Page, Tuple
  - higher lock => lower is locked
  - intention lock: must have I-lock on all ancestors

- acquire top-down
- to obtain S or IS, must have IS or IX on parent
- to obtain X or IX, must have IX on parent
- release bottom-up

MVCC - maintain multiple ver. of each object

- read-only are never blocked / aborted
- MVE if same read-from
- MVSS if MVE to some serial monoversion schedule
  - monoversion: each read action returns the most recently created object version
  - VSS ⊆ MVSS (not other way round)
- SI: Xact T takes snapshot of *committed* state of DB at start of T
  - can't read from concurrent Xacts
  - Concurrent if overlap start & commits
  - O_i is more recent than O_j if T_i commit after T_j
  - Concurrent Update Property: if multiple concurrency Xact update *same object*, only one can commit (if not, may not be serialisable)
- First Committer Win (FCW): check at point of commit
- First Updater Win (FUW) - locks only used for checking (**NOT** lock-based)
  - to update O: request X-lock on O; when commit / abort, release locks
  - if not held by anyone:
    - if O has been updated by concurrent Xact: abort
    - else: grant lock
  - else: wait for T' to abort / commit
    - if T' commit: abort
    - else: use (if not held by anyone) case
- Garbage Collection: delete version O_i if exists a newer version O_j st for every active Xact T_k that started after commit of T_i, T_j commits before T_k starts (aka all active Xact can refer to O_j)
- SI performs similarly to Read Committed, but different anomalies; does not guarantee serialisablity too (violates MVSS, but not detected)
  - Write Skew Anomaly
    - Both Xact read from initial value
  - Read-Only Xact Anomaly
    - A Read-Only Xact reads values that shouldn't be possible
- SSI: keep track of rw dependencies among concurrent Xact
  - T_i -rw-> T_j -rw-> T_k: abort one of them (has false positives)
  - ww from $T_1 \rightarrow T_2$ if T_1 writes to O, then T_2 writes *immediate successor* of O
    - T_1 commit before T_j and no Xact that commits between them writes to O
  - wr from $T_1 \rightarrow T_2$ if T_1 writes to O, then T_2 reads this ver. of O
  - rw from $T_1 \rightarrow T_2$ if T_1 reads a ver. of O, then T_2 writes *immediate successor* of O
- Dependency Serialisation Graph (dashed if concurrent, solid if not)
  - if S is SI that is not MVSS, then (1) at least one cycle in DSG, (2) for each cycle, exists T_i, T_j, T_k st
    - T_i and T_k might be same Xact
    - T_i and T_j are concurrent with $T_i - rw- > T_j$
    - AND T_j and T_k are concurrent with $T_j - rw- > T_k$

Recovery Manager



- Undo: remove effects of *aborted Xact* to preserve atomicity
- Redo: re-installing effects of *committed Xact* to preserve durability
- Failure
  1. transaction failure: transaction aborts
     - application rollbacks transaction (voluntary)
     - DBMS 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

Buffer Pool

- Can evict dirty uncommitted pages? (yes => steal, no => no-steal)
- Must all dirty pages be flushed before Xact commits? (yes => force => no redo, no => no-force)
- in practice: use steal, no-force (need undo & need redo)
- no steal => no undo needed (not practical because not enough buffer pages leads to blocking)
- force => no redo needed (hurts performance of commit because random I/O)

Log-Based DB Recovery

- Log (trail / journey): history of actions executed by DBMS - stored as sequential file of records in *stable storage* - uniquely identified by LSN
- Algorithm for Recovery and Isolation Exploiting Semantics (ARIES) - designed for steal, no-force approach, assumes strict 2PL
- Xact Table (TT): one entry per *active Xact*, contains: XactID, lastLSN (most recent for Xact), status (C or U) because kept until End Log Record
- Dirty Page Table (DPT): one entry per *dirty page*, contains: pageID, recLSN (earliest for update that caused dirty)

Normal Processing

Updating TT (Xact ID, lastLSN, status):

- first log record for Xact T: create new entry in TT with status = U
- for new log record: update lastLSN
- when commit: update status = C
- when end log record: remove from TT

Updating DPT (pageID, recLSN):

- when page P is updated (and not in DPT): create new entry with recLSN = LSN (don't update this)
- when flushed: remove from DPT

Log Records

- default: LSN, type, XactID, prevLSN (for same Xact, first points to NULL)
- update log record (ULR): pageID, byte offset (within page), length (in bytes of update), before-image (for undo), after-image (for redo)
- compensation log record (CLR) - made when ULR is undone: pageID, undoNextLSN (prevLSN in ULR), action taken to undo
- commit log record
- abort log record - created when aborting Xact: undo is initiated for this Xact
- end log record - created after book-keeping after commit / abort is done
- (simple) checkpoint log record: stores Xact table
- (fuzzy) begin_checkpoint log record: time of snapshot of DPT & TT
- (fuzzy) end_checkpoint log record: stores DPT & TT snapshots

* only ULR and CLR are redoable log records

Implementing Abort

- Write-ahead logging (WAL) protocol: do not flush uncommitted update until log record is flushed
- need to log changes needed for undo
- to enforce, each DB page contains pageLSN (most recent log record), before flushing page P, ensure all log records up to pageLSN is flushed

Implementing Commit

- Force-at-commit protocol: do not commit until after-images of all updated records are in stable storage
- to enforce, write commit log record for Xact, flush all log records (not data)
- Xact is committed $\iff$ its commit log record is written to stable storage

Implementing Restart (order matters)

- Analysis phase: determines point in log to start Redo phase, identifies superset of dirtied buffer pool pages & active Xacts at time of crash
- Redo phase: redo actions to restore DB state
- Undo phase: undo actions of uncommitted Xacts

Analysis Phase

- init DPT and TT to be empty
- sequentially scan logs

```
if r is end log record:
  remove T from TT
else:
  add T to TT if not in TT
  set lastLSN = r's LSN
  status = C if commit log record
if (r is redoable) & (its P not in DPT):
  add P to DPT(pageID = P, recLSN = r)
```

Redo Phase

- opt cond (defn already flushed) = (P is not in DPT) or (P's recLSN in DPT > r's LSN)

```
redoLSN = smallest recLSN in DPT
let r = log record w/ redoLSN
start scan from r:
if (r is ULR | CLR) & (not opt cond):
  fetch page P for r
  if P's pageLSN < r's LSN:
    haven't redo, so redo action
    set P's pageLSN = r's LSN
  else: because <= P's pageLSN is OK
    set P's recLSN in DPT =
      P's pageLSN+1
```

at end: create end log records for Xacts with status = C, & remove from TT

Undo Phase: abort loser Xacts

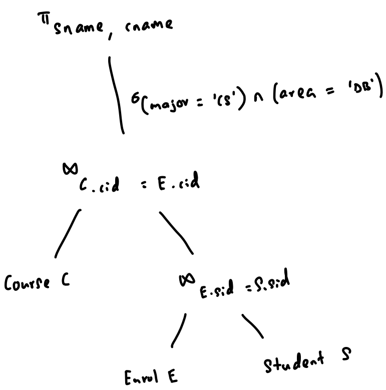
```
init L = lastLSNs (status = U) from TT
repeat until L is empty
  delete largest lastLSN from L
  let r be log record for ^
  if r is ULR:
    create CLR r2
    r2 undoNextLSN = r's prevLSN
    r2 prevLSN = r's LSN
    undo action
    update P's pageLSN = r2's LSN
    UpdateLAndTT(r's prevLSN)
  if r is CLR:
    UpdateLAndTT(r's undoNextLSN)
  if r is abort log record:
    UpdateLAndTT(r's prevLSN)

def UpdateLAndTT(lsn):
  if lsn is not null: add lsn to L
  else: # reached first log => done
    create end log record for T
    remove T from TT
```

Simple Checkpointing

- periodically perform checkpointing: suspend normal processing, wait until all current processing is done, flush all dirty pages in buffer (to sync log record & DB), write checkpoint log record containing TT, resume
- during Analysis Phase, start from begin_ LR, init with TT, DPT in end_

Fuzzy Checkpointing (no suspension)

- snapshot DPT & TT, write begin_ log record
- write end_ log record (very slow to write)
- write special *master record* containing LSN of begin_ to *known location* (for fast retrieval) in stable storage
- during Analysis Phase, start from begin_, init with TT and DPT in end_



Lock Requested \ Held	-	I	S	X
I	✓	✓	×	×
S	✓	×	✓	×
X	✓	×	×	×

Lock Requested \ Held	-	IS	IX	S	X
IS	✓	✓	✓	✓	×
IX	✓	✓	✓	×	×
S	✓	✓	×	✓	×
X	✓	×	×	×	×

LOG

LSN | prevLSN | Xact ID | type | page ID | length | offset | before image | after image

DPT

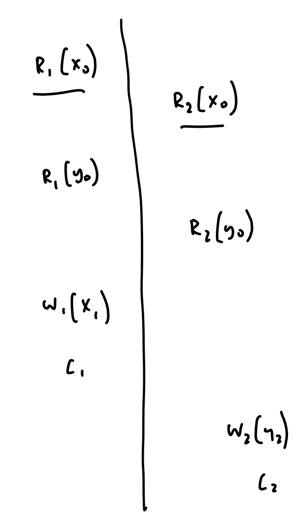
recovery ↓

page ID | recLSN

TT

Xact ID | lastLSN | status

Write-Skew Anomaly



Read-Only Anomaly