JOIN R(M) outer S(N) inner R⋈S

- Simple nested loop $M + (m \times N)$
- Block nested $M + (\lceil \frac{M}{B-2} \rceil \times N)$
- Index nested loop $M + (m \times C)$ (C: index lookup cost)
- Sort-Merge (good if R or S sorted / has cluster index)
  - Sort R: $2 \cdot \{M,N\} \cdot (1 + \lceil \log_{B-1} \lceil \frac{\{M,N\}}{B} \rceil \rceil)$
  - Merge: $M+N$ if mostly unique, MN if all dup
- Basic Hash (opt: bloom; good if know R fits)
  1. Build htab for R w h_1
  2. Prob htab for every $s \in S$
- Grace Hash (use disk, usually faster than sort-merge)
  1. Build htab for R & S with h_1. Recursively partition with h_2.. if needed
  2. For each corresponding buckets do some loop join

If no rec part & htab on disk:

Part cost $2 \times (M+N)$

Prob cost $M+N$

	F	N-F
S		WAL faster
N-S	easy. slow	

↑ isolation level ↓ performance

Extern Merge Sort

passes $= 1 + \lceil \log_{B-1} \lceil \frac{N}{B} \rceil \rceil$

IO cost $= 2N \cdot$ # passes

Half full: leaf $\geq \lceil \frac{M-1}{2} \rceil$ keys internal $\geq \lceil \frac{M}{2} \rceil$ ptrs

Query Execution Models top-down / bot.-up

- Iterator aka Volcano, Pipeline
  - Next(), Open(), End()
- Materialisation + OLTP → Can prop down LIMIT etc.
  - Output() to get entire result
- Vectorisation + OLAP + use vectorised instrs
  - Next() returns batch

Access Method

- Seq Scan → Prefetch → Bypass buf pool → Late materialise
  - → Data Skipping: be lossy / keep zone map (page aggs)
- Index Scan → histogram → multi-index (set ops)

Del / update: Track modified records ids

Halloween Problem: updated tuple moved and read again

Scheduling

Node just calls children / put stuff in work queue ← walk dep. tree

Intra-Query Par

- Horizontal (gather, distribute, repartition – exchange ops)
- Vertical
- Bushy (do both)

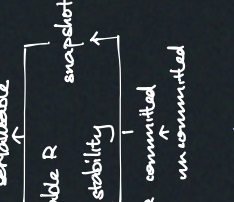
I/O Par

- RAID controller, multi disks behave like one to DBMS
- Partition – shared log, separate directory

Query Optim

logical plan → physical plan, NP-hard w.r.t. # of joins

- Rules → push predicate σ down → $\sigma_{A.id=B.id}(A \times B) \equiv \bowtie_{A.id=B.id}$
  - → push projection π down
- Cost based, with cost model on {CPU, RAM, Disk I/O, network}
  - Single-rel: find best access path for each col
  - Sys R: DP to find min cost among all left-deep join
  - Bot.-up: start from data source, reach logical query
  - Top-down: start from query, search down to find best physical



Cost Estimation

→ Histogram → Independence assumption (?) → Take samples

→ Assume uniform dist. → Inclusion assumption: join key match across rels

Txn Manage, wanted:

Atomicity	All or nothing, no partial completion
Consistency	Start consistent and end consistent
Isolation	Different txns isolated
Durability	Persist after txn commit

Dependency Graph & Serialisability

T_1	T_2	
W(A)	R(A)	W-R dirty read
R(B)	W(B)	R-W unrepeatable read
W(C)	W(C)	W-W lost update

Thm No cycle in dep graph ⇔ the schedule is conflict serialisable

$T_1 \rightarrow T_2$ ✓ $T_1 \rightleftarrows T_2$ ✗

Thm Direction in the dep graph is corresponding serial ordering

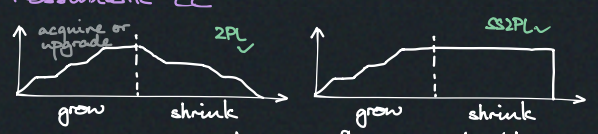
Verify conflict serialisable: use dep. graph or keep swapping ops not in conflict to get serial schedule

Def View serialisable allows blind write i.e. write without read (other constraints same as conflict serialisable)

S_1, S_2 view ≡ if

- T_i reads initial val in S_1 ⇔ T_i reads initial val in S_2
- T_i reads T_j's write in S_1 ⇔ T_i reads T_j's write in S_2
- T_i does last write in S_1 ⇔ T_i does last write in S_2

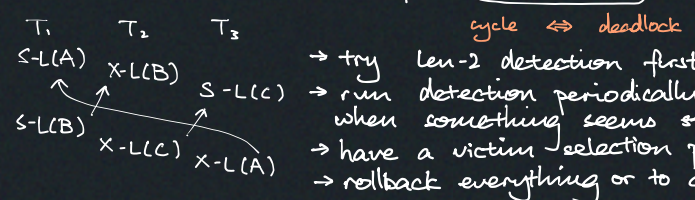
Pessimistic CC



- 2PL + guarantees conflict serialisable
  - suffers cascading abort
  - can have dirty read and deadlocks
- Strong Strict 2PL: only unlock when txn done
  - writes not read/overwritten by another txn until commit
  - + conflict serialisable + no cascading abort

Deadlock Handling

- Detection with wait for graph



→ try len-2 detection first
→ run detection periodically or when something seems stuck
→ have a victim selection policy
→ rollback everything or to clept

- Prevention: kill a txn if two try to grab same lock, based on some priority (say higher for older txns)
  (when dead txn restarts, reuse orig timestamp)

requesting txn: if higher priority than holding txn then ↓ else ↓

wound the holding txn → Wound-wait | Wait-die

T_1	T_2
BEGIN	BEGIN
X(A) →	X(A)

wound-wait: T_2 aborts

wait-die: T_1 wait

Distributed Joins

select * from R join S on R.id = S.id, 2 nodes

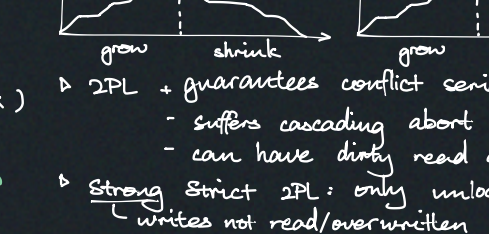
1. If S replicated: do local join and concat result
2. If S, R partitioned by same ranges: again do local join
3. If S, R partitioned differently and R part. by join key
   broadcast to case 1 to have S copied on both
4. Else – neither S, R part. by join key
   split up networking to go to case 2
   like hash join, just with hashed partitions on nodes

→ Semi-join: project join key and send only that

Dist Query Exec

Push query to data Send query to where the data is

Pull data to query Bring needed data to processor running the query



MVCC (optimistic)

Write: make new version

Read: find the correct version

Benefits: W doesn't block R, R doesn't block W (when W-W, kill second writer)] they can find the right version and time travel

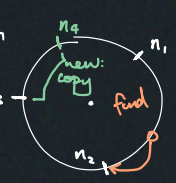
Snapshot: like repo at commit, with copy of Txn Table

Storing versions

- Just append to heap and chain things up
- Time travel table (extern ver. chain)
- Delta storage

GC:

- Tuple level: → chasing w by proc → each thread identifies, on old-to-new chain only
- Txn level: each txn keeps write set



Logging & Recovery

Undo – revert aborted or unfinished txns

Redo – reapply committed txns

Buf Pool Policy

STEAL: uncommitted txn can overwrite most recently committed value in volatile storage

FORCE: changes must write to non-volatile before txn commits (and return success to user)

Isolation Levels

- serializable – no phantoms, no dirty reads, all reads repeatable
  - get all locks first, index lock, and strong strict 2PL
- repeatable read – allow phantoms
  - same as above but without index lock
- read committed – allow phantoms and unrepeatable reads
  - same as above but release S lock immediately
- read uncommitted – allow all three
  - same as read committed but no S lock
- (snapshot isolation) – allows write skew ↙ cuts through timeline
  - all read consistent with a snapshot at txn's start
  - commit only if writes don't conflict with other txns' writes since snapshot taken
- (cursor stability) – prevents lost update

Def

Txn recoverable if it commits only after things that wrote to what it reads have committed

Dirty read: read uncommitted write

Unrepeatable read: read and get sth different next time

Phantom read: redo scan, different rows

Write skew:

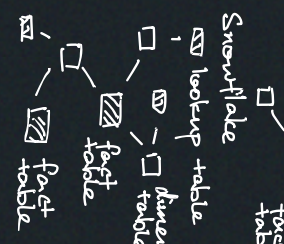
CC

Distributed Archi

Failure types: 2.

- Software failure
- Hardware failure — bugs, etc.
- power, cannot charge, etc.

Schemas



Dist OLTP2

Propagation Scheme

↑ Sync (strong consistency)
↳ Flushed to replica before commit succeed
send change log, other node recover the log

→ Async (Eventual consistency)
↳ Return commit after sending out log
↳ (need recovery if other node crash)

Failure types: 3. Storage failure

- Disk fail
- Controller problem

Timestamp Ordering CC (Optimistic)

Txn gets unique TS, atomic monotonic, need serialisability in this order

- Basic: tuples get R & W TSs, conflict serialisable, may not be recoverable
  - Read — can't read sth written in the future
    - If written by future txn: abort reading thread, restart with new TS
    - Else update R-TS, make local copy for repeated reads
  - Write — can't write correctly if read or written in future
    - If bad write: abort and restart self
    - Else update W-TS, make local copy
  - Optimisation with Thomas Write Rule, view serialisable
    - Allow but ignore blind write: if $TS(T_i) < W\text{-}TS(A)$
  - Issues: overhead, long txns can get starved
- Optimistic CC (OCC), good if txns touch few things, no deadlocks, easy to abort, waste work if validation fail
  1. Read Phase — all reads & local writes, work in local workspace, track all reads & writes
  2. Validation — any conflict with other txns? (restart if failed) ← declare timestamp at this point
  3. Write Phase — apply local changes to DB

Cases

1. T_1: R … V W; T_2: R … V W (starts after T_1 finishes) — R→W, W→R, W→W — Ok
2. T_1 finishes W before T_2 starts W — R→W, W→R, W→W — Only ok if $WriteSet(T_1) \cap ReadSet(T_2) = \emptyset$ (these set checks only cheap if OLTP)
3. T_1 finishes R before T_2 finishes R — R→W, W→R, W→W — Only ok if $WriteSet(T_1) \cap ReadSet(T_2) = \emptyset$, $WriteSet(T_1) \cap WriteSet(T_2) = \emptyset$

Validation Scope:

- Backward — Check against those that committed
- Forward — Check against those that could commit in future

Write phase:

- Serial commit — one txn write at a time
- Parallel commit — use latches to write but ensure ordering

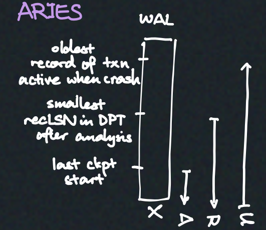
Checkpoint

- Naive: pause starting new txns, wait for active txns to finish, flush all pages
- Slightly better: pause write txns, flush modified pages, save ATT and DPT at start of ckpt
- Fuzzy (↑ ARIES): put in CHECKPOINT-BEGIN ← at this point in time; memcopy ATT and DPT; let execution continue while preparing ckpt; when done, put in CHECKPOINT-END + ATT + DPT (start recovery at CHECKPOINT-BEGIN)

ATT txn modes: running, committing, undo candidate

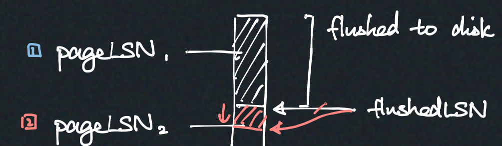
Failure types: 1. Txn failure — Logical failure — constraint violation, etc. — Internal state — deadlock, etc.

ARIES



log disk LSNs · Disk pageLSNs · Page, pageLSN · Page, pageLSN · DRAM flushedLSN

Buf pool mgr can be modular

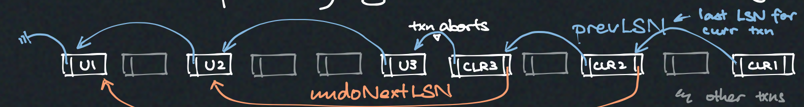


① can directly evict this page
② when evict, need to first flush log

flushedLSN	mem	last logged LSN on disk
pageLSN	$page_x$	newest update to $page_x$
recLSN	DPT (Dirty Page Table)	oldest update to $page_x$ since last flush
lastLSN	ATT (Active Txn Table)	latest record of txn T_i
MasterRecord	disk	LSN of last ckpt

Commit: write log, ensure flushed, log END-TXN

Abort: write compensating log record (CLR) for each change, log END-TXN



Analysis: reconstruct ATT & DPT right before crash

- remove ended txns from ATT
- add record to ATT with Undo unless Committed for the txn
- add written pages to DPT

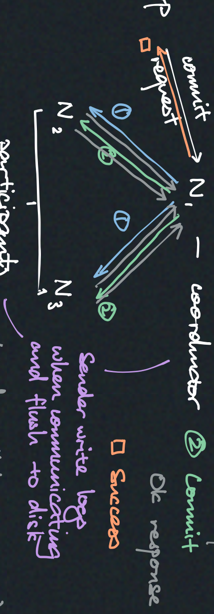
Redo: reapply logged changes to right before crash; when done, set committed txns to C and remove from ATT

Undo: in rev order undo changes of crashed/aborted txns

2 Phase Commit (atomic commit protocol)

- Consistency — all end up in same state
- Stability — no change after decision
- Liveness — no hang

* Success case



* Failure: if any participant says abort, coordinator decides abort, coordinator tells everyone abort
* Recovery:
  - In in prepare state, tell coordinator
  - Not prepared, just say abort
  - If coordinator and is committing, tell participants commit
* Optimisations:
  - → If last query, say it's last and ask if prepared
  - → Coordinator say commit success early, before phase 2

Distributed Partitioning

Physical (usually for shared nothing)

- Naive — by table
- Vertical — part. by column
- Horizontal — by key hash, ranges, round robin, …

Logical (usually for shared disk)

Middleware: routes queries to right part.

Dist OLTP3

Prop timing

- → Continuous log streaming
- → On commit

Active vs passive

- → Active - Passive: One send one receive
- → Active - Active
  - Run query on all replica, if all agree then return
  - Ship queries btwn nodes

Phantom Problem

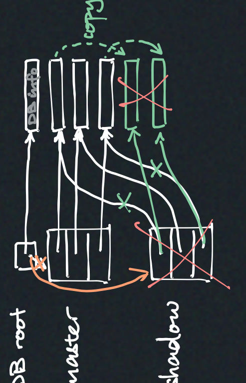
when not locking new DB objects eg. no lock on new tup

- → Rerun scan at commit to check for inconsistency
- → Predicate locking
- → Index locking eg. → Virtual lock on non-existent key; → Gap lock for non-exist ranges; → key range lock → Hierarchical lock

Shadow Paging (N-S, F)

Like file table after fork

Keep master page table & per-txn shadow page table



when write

when commit — switch to the shadow table

when abort — throw away shadow copy

never need any redo

Journal file

Before modifying a page, make and log a copy

Write Ahead Log (WAL)

Force log only, use log to recover, log to log page

sequential action list for each txn:

- ⟨T_1, begin⟩
- ⟨T_1, A, 1, 8⟩ A=1 → A=8
- ⟨T_1, B, 5, 9⟩ B=5 → B=9
- ⟨T_1, commit⟩] at this point, flush log to disk

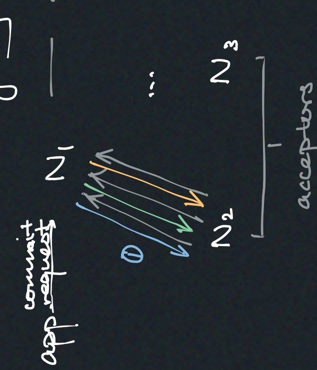
Txn complete only when all logs flushed, flush log before pages

Optimisation: Group commits and batch flush

Logging schemes:

- Physical — byte changes like git diff
- Logical — "update all records satisfying …"
- Physiological — physical across pages, logical within page

Paxos — use majority vote



① Ok if I be proposer?

② Propose

③ Commit if majority

Proposer race — handled by proposer number ↲ monotonically increasing

- → Multi-Paxos: Select leader to lead for a while
- → Raft: Only node with most up-to-date log becomes leader

Consistency Issues (CAP / PACELC)

Want:

- Consistency
- Always available
- Network partition tolerant

CAP Thm: you can only get 2 of those

But for certain data struct, like those CRDT stuff, we can get all 3.

Dist OLTP1

K-safety: k replica must be avail. for each DB obj

Replica config (in propagate / in write)

- → Primary - Replica
  - leader selection if primary down
- → Multi - Primary