Disk, Buffer, Files

- Ordering: File → Page → Record
- **Unordered Heap Files (DDL)**
  - header page
  - each page has 2 ptrs + free + data
  - cons: check many pages w/ free space
- **Unordered Heap Files (page directory)**
  - multiple header pages (ptr to page + #free)
  - header page accessed often → in cache
  - faster to find insert page → HP tells #free bytes
- Page Layout
  - **Header**: #records, free space, next/last ptr, bitmap, slot table
  - **Fixed length, packed**: rid = (pageID, location)
    add: append, delete: update rid.
  - **Fixed length, unpacked**: bitmap, (pageID, slotID)
    add: 1st empty slot, delete: clear bit
  - **Variable length**: slotted page (footer)
    ① ptr to free page, ② slot count
    ③ length + ptr to record (in reverse order)
    - rid = location in slot from right
    - Delete: set slot dir ptr to null.

B+Trees

Total capacity = $2d(2d+1)^h$

Max fanout = $2d+1$

#leaf nodes = $(2d+1)^h = F^h$

- IO cost for bit 3 ≤ bit 2 accessing same data
- Always balanced, grow at root not leaf
- Bit 2: (h+1) read leaf + 1 read data page = h+2
- Bit 2" range search: (h+1) + x read other leaves + (# matching pages) IO read data
- Bit 2: unclustered range search: same as ↑, but (# matching records) IO read data

$2N(1+\lceil \log_{B-1}\lceil N/B\rceil\rceil)$

	Heap	sorted	clustered index
scan all	BD	BD	$\frac{3}{2}$ BD
equality	0.5 BD	$(\log_2 B)D$	$(\log_F(BR/E+2))D$
range	BD	$((\log_2 B)+\#pages))D$	$(\log_F(\frac{BR}{E})+3\cdot pages)D$
insert	2D	$((\log_2 B)+3)D$	$(\log_F(\frac{BR}{E})+4)D$
delete	$(0.5B+1)D$	$((\log_2 B)+B)D$	$(\log_F(\frac{BR}{E})+4)D$

Sorting

- Pass 0: use B buffer pages, produce $\lceil \frac{N}{B} \rceil$ sorted page of B pages each.

  pass 1-n: merge B-1 runs at a time
- 2 passes at most $B(B-1)$ pages, 3: $B(B-1)(B-1)$
- # passes = $1+\lceil \log_{B-1}\lceil \frac{N}{B}\rceil\rceil$
- Total I/O = $2N(1+\lceil \log_{B-1}\lceil \frac{N}{B}\rceil\rceil$
- Parallelize: split on value range, each send

Hashing

- Partition: B-1 partitions, each size ≤ B pages
- Total I/O: ~2N (#passes)
- Parallelize: shuffle data across machines

Joins

- Naive Join: cost = [R] + |R|[S], $\overbrace{[R] \le B}$ $[R] \le (B-2)*$ fill factor
- Page Nested Loop Join: [R] + [R][S]
- Blocked Nested Loop: $[R] + \lceil [R]/(B-2) \rceil * [S]$
  B-2 b/c: 1 for write, 1 for read S
- Index Nested Loop Join: equijoin
  $[R] + |R| * (\text{cost to find match})$
- Sort Merge Join: equijoin
  output sorted on join attribute
  ① sort R, S. ② Router loop, S inner, track "mark"
  Best case: sort R + sort S + ([R]+[S])
  worst case: last term is [R]+|R|[S]
  optimization: if #runs in last merge for R+S+1 ≤ B
  ⇒ subtract 2([R]+[S]) I/Os
  Ex How big the buffer have to be to sort both R and S in two passes each?
  $B > \sqrt{\max([R],[S])}$
- Grace Hash Join: equi, $[R] \le B^2$
  - partition: B-1 partitions, 2([R]+[S]) I/O
  - Matching: B-2, [R]+[S] I/O

  ⇒ Total: 3([R]+[S])

Query Optimizations

- System R - n passes

  *SMJ helps ORDER BY *
  - Pass 1: find min cost for each relation, interesting order (index scan, full table scan) ↳ ORDER BY, GROUP BY, joins
  - Pass i: take in list of i-1 optimal plans, select min cost

* NO cartesian product, right deep plans

|c| = # distinct values for the column

Predicate	Selectivity	Assumption
c = v	1 / (number of distinct values of c in index)	We know |c|.
c = v	1 / 10	We don't know |c|.
c1 = c2	1 / MAX(number of distinct values of c1, number of distinct values of c2)	We know |c1| and |c2|.
c1 = c2	1 / (number of distinct values of ci)	We know |ci| but not |other column|.
c1 = c2	1 / 10	We don't know |c1| or |c2|.
c < v c > v	(v - min(c)) / (max(c) - min(c) + 1) (max(c) - v) / (max(c) - min(c) + 1)	We know max(c) and min(c). c is an integer.
c < v c > v	1 / 10	We don't know max(c) and min(c). c is an integer.
c <= v c >= v	(v - min(c)) / (max(c) - min(c) + 1) + (1 / |c|) (max(c) - v) / (max(c) - min(c) + 1) + (1 / |c|)	We know max(c) and min(c). c is an integer.
c <= v c >= v	1 / 10	We don't know max(c) and min(c). c is an integer.
c >= v	(max(c) - v) / (max(c) - min(c))	We know max(c) and min(c). c is a float.
c >= v	1 / 10	We don't know max(c) and min(c). c is a float.
c <= v	(v - min(c)) / (max(c) - min(c))	We know max(c) and min(c). c is a float.
c <= v	1 / 10	We don't know max(c) and min(c). c is a float.
p1 **AND** p2	S(p1)*S(p2)	Independent predicates
p1 **OR** p2	S(p1) + S(p2) - S(p1)*S(p2)	Independent predicates
NOT p	1 - S(p)	

wt

float

* node + sec. leaf + sec. data page *

Transactions & Concurrency

All schedules:
- Serializable:
  - View serializable:
    - Conflict Serializable:
      - Serial

ACID

① Atomicity: all or none, commit or abort.

② Consistency: consistent → consistent, otherwise abort.

③ Isolation

④ Durability: txns persist after commit

Equivalent: 2 schedules involve same txns, each txn's order is the same, leave DB with same final state.

Serial schedule: run from start to end without intervention.

Serializable: if equivalent to some serial schedule → isolation

conflicting ops: ① from diff. txns, ② on same resource, ③ 1 write op.

conflict equivalent: 2 schedules involve same ops of same txn, every pair of conflicting actions ordered the same way.

conflict serializable: conflict eqv. to some serial schedule.

*order of blind writes ⇒ serializable. Transform S by swapping consecutive non-conflicting ops of diff. txns. * no cycle in graph

dependency graph:

T1: R(A) W(A) R(B)
T2: R(A) W(A) R(B) W(B)

view equivalent: conflict eqv, except for potential order of blind writes
① same initial reads, ② dependent reads, ③ winning writes

view serializability: view eqv to a serial schedule
fewer false negatives than conflict serializability

2PL: txn cannot get new lock after releasing any lock

* V conflict serializability * **Lock point**: when a committing txn has reached the end of its acquisition phase. order of lock points gives an eqv. serial schedule.

* Do not avoid cascading aborts

→ avoid cascading aborts

Strict 2PL: all locks released together when txn completes

Lock Manager: {granted_set, mode, waiting queue} for each locked obj.

Lock upgrade: S → X, puts into front of queue.

Deadlock: wait-for graph (cycle = deadlock)
wait-die, wound-wait

Lock granularity: (coarse) DB → Table → page → record (fine)

2PL
- S, IS → parent holds IS/IX
- X, IX, SIX → parent holds IX/SIX

Lock compatibility Matrix:

Mode	NL	IS	IX	S	SIX	X
NL	T	T	T	T	T	T
IS	T	T	T	T	T	F
IX	T	T	T	F	F	F
S	T	T	F	T	F	F
SIX	T	T	F	F	F	F
X	T	F	F	F	F	F

Phantom Problem:
concept of conflict serializability doesn't apply (only for static DB).
- lock entire table
- lock index entry if index available
- predicate lock

Recovery

Durability

Force: force dirty pgs to disk when commits

No-Force: only write to disk when evicted from buffer pool.
↳ * reduce unnecessary writes, but not Durable ⇒ REDO

Atomicity

Steal: can evict pages with uncommitted updates
↳ * max use of buffer pages, but violates Atomicity ⇒ UNDO

No-steal: cannot flush pages w/ uncommitted updates

write-ahead log: ① txn not committed until log for all changes → disk
② changes logged before make modification on disk.

UNDO Log: ⇒ Atomicity. (START, COMMIT, ABORT, <T,X,V>) (V: old val)
- If T commits, FLUSH(x) → disk before <COMMIT T>.
  Force → UNDO any change if txn crashes before COMMIT.
- If T modifies X, <T,X,V> → disk before FLUSH(X)
  Steal → UNDO any change if txn crashes before FLUSH
- UNDO all uncommitted txns start from earliest uncommit txn.
- txn complete if START + COMMIT/ABORT. only START incomplete.

REDO Log: ⇒ Durability (START, COMMIT, ABORT, <T,X,V>) (V: new val)
- If T changes X, both <T,X,V> and <COMMIT T> → disk before FLUSH
  no-steal, no-force → REDO any changes if crash before FLUSH
- REDO all committed txns start from beginning

ARIES: UNDO-REDO, steal, no-force.

Every checkpoint

Transaction Table
- xID: txn ID
- status: running, committing, aborting
- lastLSN: LSN of most recent txn

Dirty Page Table (DPT)
- pageID
- recLSN: 1st operation to dirty the page

Log Record
- LSN
- prevLSN
- xID
- type: update, commit, abort, CLR, END, BEGIN_CKPT, END_CKPT

update: [pageID, length, offset, before, after]
- flushedLSN (in RAM, most recent log written to disk)
- pageLSN: LSN of last op. to update the page
- undoNextLSN: stored in CLR, LSN of next op to undo.

Fuzzy checkpoint protocol:

① log <BEGIN CKPT>, ② flush log to disk, ③ continue normal ops, ④ when xactTable & DPT written to disk, log <END CKPT>, ⑤ flush log → disk.

ARIES Normal Ops:

- **xact start**: log<START>; update xactTable.
- **xact update**: log<UPDATE>; update prevLSN = lastLSN, pageLSN = LSN, lastLSN = LSN, recLSN = LSN if null.
- **Pg flush**: Flush log ≤ pageLSN; remove page from DPT and buffer pool.
- **Pg fetch**: create entry in DPT & buffer pool; recLSN = null.
- **xact commit**: log<COMMIT>; flush log to this entry; update xactTable to commit; log<END>; update xTable complete.
- **xact Abort**: log <ABORT>; start UNDO changes from lastLSN; log <CLR>; undoNextLSN = next to undo; follow lastLSN; change xTable to abort; flush dirty pgs to disk; log <END>; change xTable to complete.

Recovery: start from CKPT.

Analysis start from last successful CKPT, after BEGIN CKPT
- get xTable, DPT from CKPT
- scan log forward: ① END: remove xact from xTable
  ② UPDATE: if p not in DPT, add, recLSN = LSN
  ③ !END: Add xact to xTable, lastLSN = LSN, change xact status on commit/abort (CLR counts)
- * At end: for any xact in xTable in COMMITTING, write <END> log, remove from xTable. change status of running xact to aborting, write <ABORT> log.

REDO *Durability
- scan forward from smallest recLSN in DPT after Analysis.
- For each UPDATE/CLR, REDO, unless:
  ① pg not in PDT, ② pg in PDT, but recLSN > LSN
  ③ pageLSN ≥ LSN
- TO REDO, reapply logged actions, set pageLSN = LSN. NO loggin & forcing

UNDO * Atomicity
- scan backward until oldest log rec. of xact active at crash.

```
toUndo = {lastLSNs of all xacts in xTable}
while (!toUndo.empty()) {
    thisLR = toUndo.find_and_remove_largest_LSN()
    if thisLR.type == CLR: (do not undo)
        if thisLR.undoNextLSN != null: add undoNextLSN to toUndo.
        else: write END for thisLR.xID
    else:
        if thisLR.type == UPDATE: write CLR, undo update in DB
        if thisLR.prevLSN != null: add prevLSN to toUndo
        else: write END for thisLR.xID.
```

Inequalities:

① pageLSN ≤ flushedLSN: flushed the corresponding log records before we can flush data page to disk. Before page i can be flushed, log records for all ops that have modified page i must have → disk.

② flushedLSN ≥ lastLSN$_T$: all logs must → disk before xact T commits. If flushedLSN > last op of the xact, all logs for that xact on disk.

③ recLSN$_p$ ≤ in memory pageLSN$_p$: If a page is in DPT, must be dirty. So last update must not have made it to disk. recLSN$_p$ is the 1st op to dirty page, so must < last op to modify the page.

④ recLSNp > on disk pageLSNp: If page dirty, the op that dirtied the page (recLSN) must not have → disk yet. So must be after the op that did make it to disk for that page.

DataBase Design

ER Diagram constraints

- key constraint (at most 1) →
- participation constraint (at least 1) —
- key cons. + total participation (==1) ⟶
- Non-key partial participation (0 or more) —

weak entity partial key
- must be one-to-many relationship (1 owner) with total participation

ER to tables
- many-to-many: keys for each entity set as foreign keys ⇒ this set = superkey for relation
- key constraints: primary key = primary key of weak entity
- key + participation: foreign key on owner NOT NULL, ON DELETE NO ACTION
- weak entity: primary key = (weak PK, owner PK)

Armstrong's Axioms
- Reflexivity: if $Y \subseteq X$, then $X \to Y$
- Augmentation: if $X \to Y$, then $XZ \to YZ$
- Transivity: if $X \to Y$ and $Y \to Z$, then $X \to Z$
- Union: if $X \to Y$ and $X \to Z$, then $X \to YZ$
- Decomposition: if $X \to YZ$, then $X \to Y$ and $X \to Z$

Functional Dependency: $X \to Y$

superkey: if $X \to$ [all attributes of R]

candidate key: minimum superkey

closure: closure of a set of FDs F is $F^+ \Rightarrow$ set of all FDs implied by F

attribute closure: set of attribute A: $X \to A$ is in X^+

BCNF: if for every FD $X \to A$ that holds over R, either $A \subseteq X$ or X is a superkey $\Rightarrow$ no redundancy

BCNF Decomposition: For each FD $X \to Y$ in F^+:
if $X \to Y$ violates BCNF, decompose R into
① X^+, ② $X \cup (R - X^+)$
*Final results depend on order of decomposition.

Lossiness: BCNF always lossless
lossless iff F^+ contains ① $X \cap Y \to X$ or ② $X \cap Y \to Y$
* otherwise create extra "junk" > original #tuple

Dependency Preserving: BCNF not always dep. pres.
iff $(F_X \cup F_Y)^+ = F^+$

i. **(1 pt)** For a given schedule, it is possible for the same transaction to abort under both wait-die and wound-wait deadlock prevention policies.

- [x] True
- [ ] False

ii. **(1 pt)** Guaranteeing view serializability is sufficient for guaranteeing isolation from the ACID properties.

- [x] True
- [ ] False

view → serializability → isolation

iii. **(1 pt)** The waits-for graph is the conflict dependency graph with edges reversed.

- [ ] True
- [x] False

dependency graph x consider which lock each txn holds at each timestamp, waits-for graph does

iv. **(1 pt)** A view serializable schedule will always satisfy strict 2PL requirements.

- [ ] True
- [x] False

vii. **(3 pt)** Consider the following modification to the Selinger Optimizer: after we determine our **final** query plan, we test if materializing the output of each operator (single access or join) before inputting it to the next operator will reduce the I/O cost. For our query above, how many query plans would we have to consider after the final step of the Selinger Optimizer if we consider all possible combinations of materialization/no materialization?

5 operators (3 single access, 2 join) $\Rightarrow 2^5 = 32$ combinations

viii. **(1 pt)** True or False? Pushing down on-the-fly selections in single access scans will always reduce the final I/O cost of the query plan outputted by the Selinger Optimizer.

- [ ] True
- [x] False

push down right relation of BNLJ x decrease IO

`[R] = 60, [S] = 40`

Lakshya wants to try using GHJ. Like before, for each pass of GHJ, Lakshya has fewer buffer pages than the previous pass. Consider the following figure for the number of buffer pages available for each pass and answer the questions below. Assume that our pass 1 hash function hashes $\frac{1}{2}$ of the pages to the first bucket, $\frac{1}{4}$ pages to the second bucket, and perfectly hashes the remaining pages uniformly across the remaining buckets,

GHJ	Partitioning Pass 1	Partitioning Pass 2	Build and Probe
	20	?	12

i. **(3 pt)** What is the total I/O cost of pass 1?

209

Table `R`: 60 (read) + 30 (bucket 1) + 15 (bucket 2) + 1 * 17 (buckets 3 to 19 combined) = 60 + 62 = 122 I/Os
Table `S`: 40 (read) + 20 (bucket 1) + 10 (bucket 2) + 1 * 17 (buckets 3 to 19) = 40 + 47 = 87 I/Os
Table `R` + Table `S` = 122 + 87 = 209 I/Os
Partial: 109 (did not include first read)

ii. **(3 pt)** Assuming we use a uniform hash function in pass 2, what is the minimum number of buffer pages that must be available in pass 2 in order to ensure that this is the **last partitioning phase**?

3

To fit everything in the build and probe we only need to ensure that all partitions have one table of size 10 pages or less (B - 2 = 10) – this enables us to build an in-memory hash table. As we use a uniform hash function in pass 2, it is sufficient to ensure that the largest bucket can be hashed into subpartitions – we just need to hash `S` into a small enough partition to build an in-memory table on it and then probe with `R`.

i. **(2 pt)** How many IOs will Sort Merge Join of X (25 pages) and Y (30 pages) with 10 buffer pages take? You should not assume that these tables are already sorted.

Sort(X) = 2 * [X] * (1 + ⌈log_{B-1}⌈[X]/B⌉⌉) = 2 * 25 * 2 = 100
Sort(Y) = 2 * [Y] * 2 = 120
∴ IO = 100 + 120 + (30+25) − 2(30+25) = 165

Sort X: 3 runs
Sort Y: 3 runs
3+3 = 6 runs < 10 ⇒ opt

v. **(2 pt)** Soon after, Jerry gets access to a large supercomputer!

How many IOs will Sort Merge Join of X (25 pages) and Y (30 pages) with 100 buffer pages take? You should not assume that these tables are already sorted.

buffer large enough, no external sort. only read into memory
25 + 30 = 55

viii. **(2 pt)** After executing the rest of the Grace Hash Join (including recursive partitioning, if necessary) how many IOs in total will Grace Hash Join take?

200 + 180 + 190 = 570

(200 + 180) partition

naive hash join: (9+10) * 10 = 190

iii. **(2 pt)** `SELECT R.a, S.b, S.c  FROM R INNER JOIN S on R.a = S.a WHERE R.b <= 10`

25000

sel(R.b <= 10) = 1/2
sel(R.a = S.a) = 1/10
⇒ sel = 1/2 · 1/10 = 1/20

#tuples = 1/20 * (50 * 100) * (100 * 100) = 2500000
#pages = 2500000 / 100 = 25000

iv. **(3 pt)** Jerry is trying to estimate the I/O cost for R SMJ T. He thinks it's 300 I/Os (the cost of sorting R) + 150 I/Os (the cost of sorting T) + 75 I/Os (cost of merging) = 525 I/Os.

However, Jennifer disagrees and thinks we can compute R SMJ T in fewer I/Os.

List 2 optimizations we can make that will lower the I/O cost of R SMJ T.

Solutions that listed any 2 of the following optimizations received full credit:
A. Access R using index scan on R.b so that we do not have to run external sorting on R
B. Apply selection for R.b <= 10 on the fly
C. Apply the SMJ optimization

v. **(1 pt)** Will R SMJ T produce an interesting order?

- [ ] Yes
- [x] No

The output of R SMJ T will be sorted on the b column of R and T, but orderings involving R.b or T.b will be useful in any downstream joins or in the GROUP BY/ORDER BY clauses.

ii. **(2 pt)** You have a special database that commits all active transactions prematurely and flushes their updates as soon as it detects any deadlock. The database uses strict 2PL.

- [x] Atomicity
- [x] Consistency
- [x] Isolation
- [ ] Durability
- [ ] None

Atomicity is defined as "all actions in the transaction should happen or none should." In this example, some actions may commit while the rest won't (if a deadlock occurs mid-schedule) and this violates atomicity.

The DB could be an inconsistent state at the time of deadlock. For example, in a bank transaction, if A wants to send $100 to B and the deadlock occurs after A has $100 deducted from their bank account but before B has received it, the transaction would prematurely commit and consistency would be violated.

Further, If a transaction prematurely commits, it exposes some of its intermediate operations to other transactions, which means other transactions could read those changes, so they're no longer isolated.

ii. **(2 pt)** We know from lecture that a schedule is conflict serializable if and only if its dependency graph is acyclic. Your friend suggests the following process to identify the conflict equivalent serial schedule for a schedule with an **acyclic** dependency graph: find a node with only outgoing edges, then perform depth-first search on it. He states that this will always return one such conflict equivalent serial schedule. Is your friend right? If yes, explain why. If not, propose another way of finding a conflict equivalent serial schedule. Answer in 1-2 sentences.

Depth-first search seems like a good approach until you realize that there may be multiple nodes with exclusively outgoing edges. In this case, these nodes will never be included in the serial schedule. Therefore, an approach that works similarly but includes all nodes is performing a topological sort on the nodes.

iii. **(2 pt)** Under UNDO logging, what will be the value of X after recovery? Write down the answer below. No explanation needed.

9

Recall that each "write" entry in the undo log stores the previous value that was overwritten. X was written by T1 and T2, but since T2 did not commit X will hold the last value written by T1. And line 5 tells us that T1 wrote 9 to X.

i. **(1 pt)** Assume the **only** queries executed on the database are Queries A, B, and C. Queries are executed within a transaction, and each transaction consists of a **single** query. (i.e. Query A would be executed within its own transaction.)

Select **all** of the following that guarantee conflict serializability.

- [x] 2PL
- [x] Strict 2PL
- [x] No locking
- [ ] None of the above

2PL - Yes Strict 2PL - Yes No locks - Yes, It is a read only workload so you don't need locks as there are never any conflicting operations.

ii. **(1 pt)** Now, consider adding **Query D** shown below. Assume the **only** queries executed on the database are Queries A, B, C, and D. Again, queries are executed within a transaction, and each transaction consists of a **single** query. (i.e. Query A would be executed within its own transaction.)

Query D

```
INSERT INTO Polls
VALUES (20, 5, 186, 4, 3);
```

Select **all** of the following that guarantee conflict serializability.

- [x] 2PL
- [x] Strict 2PL
- [ ] No locking
- [ ] None of the above

2PL - Yes Strict 2PL - Yes

There are now writes so we need locks. Any type of 2PL will guarantee conflict serializability.

iii. **(1 pt)** Assume the **only** queries executed on the database are Queries A, B, C, and D. Again, queries are executed within a transaction. However, there may be **multiple** queries (including repeats) within the same transaction. (i.e. Query A and B would be executed within the same transaction. Query A can also be executed twice within the same transaction.)

Select **all** of the following that guarantee conflict serializability.

- [ ] 2PL
- [ ] Strict 2PL
- [ ] No locking
- [x] None of the above

This is the definition of a Phantom Read, since we may have the same query executed multiple times within the same transaction and each time there could be different tuples in the result. 2PL cannot protect against this.