DISK, FILES

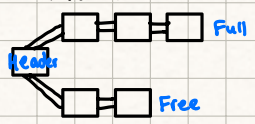
Read: Disk —Page→ RAM
Write: RAM —Page→ Disk

Disk
- SSD (Flash)
  - Cells
  - Fast random reads
  - Failure after erasure
- Platter
  - Sequential

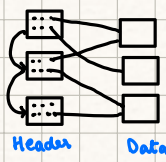
Files → Pages → Records

FILES

Unordered Heap Files — Worst Case Insert
- No order
- 2nd last of file
- 1. Linked List
  - Free page full → Move to front of full pages list



- 2. Page Directory
  - Entries in header page: < Pointer, Free Space >
  - Insertion faster



↓ Searching: full scan (N I/Os) ↑ Insertion

Sorted Files
- Pages ordered
- Records sorted by Key(s)

↑ Searching: log N I/O ↓ Insertion: log N + N I/Os (req checking if keys unique, read all data pages)

RECORDS [# page, # record id]

Fixed Length Records: Same bytes

Variable Length Records: Fixed + Variable length fields
- stores fixed first
- record header w pointers to END of VLFs

PAGES

FLR Pages
- Packed: calc pos for insertion & deletion
- Unpacked: bitmap to track open slots

VLR Pages — # records = $\lfloor (page - 8)/(record + 8) \rfloor$
- Page Footer: FSP, Slot count, < record start pointer, record length >
- Unpacked Insertion: At FSP, null < r pointer (4r), r length (4r) > set (4) periodically packed
- Packed Insertion: At FSP, new entry
- Fragmentation

SQL

Tables make up relational databases

NULL: falsey, any type

GROUPING
- summarize cols [SUM, AVG, MAX, COUNT]
- ignores NULL except COUNT(*)

check select statement w grouping?
- where → grouping → having?
- if any grouping done, only SELECT grouped/agg cols
- default ASC
- UNION removes duplicates

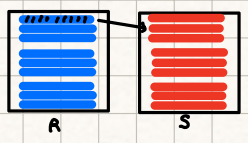
SELECT [DISTINCT] <columns> ⑤
FROM <table> ①
[INNER / LEFT / RIGHT / FULL] JOIN <table>
ON <predicate>
[WHERE <predicate> ②]
[GROUP BY <columns> ③]
[HAVING <predicate>] ④
[ORDER BY <column list>] ⑥
[LIMIT <# rows>]; ⑦
LIKE '%': 0+ chars, _ any char

Subqueries: WHERE, FROM, WITH
UNION - distinct in either
INTERSECT - common in both
ALL - all subquery meet condition
ANY - any one true

JOINS

[R][S] → # pages
|R| |S| → # records / page
exclude final writes to disk

SNLJ



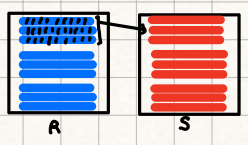
Every record → Every Record

```
for r_i in R:
  for s_j in S:
    if θ(r_i, s_j)
      yield
```

[R] + |R|[S]

Do both ones, take the min one

PNLJ



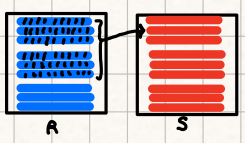

```
for each P_R in R:
  for each P_S in S:
    for each row r in P_R:
      for each row s in P_S:
        < ... >
```

[R] + [R][S]

Keep smaller relation as outer loop

SNLJ, PNLJ use 3 buffers (2 input + 1 output)

BNLJ




```
for each block of (B-2) pages C_R:
  for each P_S in S:
    for each row r in C_R:
      for each row s in P_S:
        < ... >
```

blocks

$[R] + \lceil \frac{[R]}{B-2} \rceil [S]$

(B-2) buffers for smaller one, 1 for other, 1 output

INLJ

```
for each record r_i in R:
  for each record s_j in S where θ(s_i, r_j) is true:
    <yield>
```

[R] + |R| × (cost to lookup in S)

SORT-MERGE JOIN:
1. Sort both R and S
2. Merge matching tuples

```
while (True) {
  while (r<s) { advance r }
  while (r>s) { advance s }
  mark
  while (r==s) {
    while (r==s) {
      yield <r,s>
      advance
    }
    reset s to mark
    advance r
  }
}
```

Don't need to materialize sorted relation in final merge pass of sorting both, stream to SMJ

→ SAVES 2([R]+[S])

Need: # runs in last merge of R + # runs in last merge of R ≤ (B-1)

Optimised: Total - 2([S] + [R])

RELATIONAL ALGEBRA

Projection	π	[select] takes columns
Selection	σ	[where] filters rows
Union	∪	combine diff relations, remove duplicates
Set Diff	−	rows in table 1 not in table 2
Group By	Λ	[GROUP BY]
Intersection	∩	rows in both tables
Cross Product	×	1 tuple for every possible pair
Alias	ρ	[AS] renames attributes, for join
Join	⋈	default natural join

} MUST have same attributes

∧ (AND) ∨ (OR)

can't have duplicates (tuples)

After π, other cols DON'T exist

$S_1 - (S_1 - S_2) = S_1 \cap S_2 = S_1 \bowtie S_2$

B+ TREES

- Order d, always balanced
- Must have sorted $d \le x \le 2d$ entries
- Inner Nodes have max $(2d+1)$ pointers
- Only leaf nodes have records

Max values: $2df(2d+1)^h$

INSERTION

1. Find leaf node, add <K, R> in order
2. Overflow ($L > 2d$)
   - → L_1 (d) L_2 (d+1)
   - → If L leaf, copy to parent
     If L inner, PUSH to parent
   - → Adjust pointers
   - → Recurse

DELETION

- Just delete from leaf, NEVER from inner

STORAGE

1. Alt 1 (By Val): Leaf contains data <K, R> ↓ can't support multiple indices, sorted
2. Alt 2 (By Ref): Contains pointers <K, # Page # Record> ↑ multiple indices
3. Alt 3 (By List of Ref): List of pointers <K, [···]> ↑ multiple records, same leaf node entry

CLUSTERING

Unclustered: ~1/O per record
Clustered: ~1/O per page

COUNTING I/Os

1. Read root → leaf
2. Read data page (unvs clustered)
3. Write data page
4. Write index page

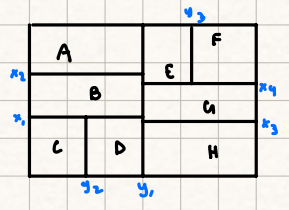
BULK LOADING

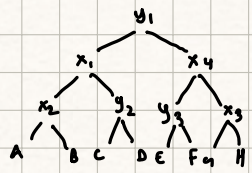
1. Sort data
2. Fill leaf to f
3. Add pointer to parent
4. Parent Overflow:
   L_1 (d) L_2 (d+1)
   Move L_2 first entry to parent
5. Adjust pointers

SELECT → best case: doesn't exist, so only root → leaf
Check if pages overflow
Can be built on any type of column
full scan = # data pages
Index Search: $h+1 = \log_F(\#\text{leaves}) + 1$
Bulk loading makes use of temporal locality

SPATIAL INDICES

Brute force search = # calc = # points
K-d tree search
K-d tree that organizes points in a K-dim space
Balanced binary tree that splits on dimensions





EQUALITY SEARCH: $\log(BR)+2$
RANGE SEARCH: $\log(BR)+1+2^*\#\text{pages}$
DELETION: $\log BR+4$
INSERTION: $\log BR+4$ → search, read leaf, write data, write index

B: # Blocks
R: # Records / block

Searching:
- → Use BST to find block it's in
- → Backtrack if other neighbour block is closer

BUFFER MANAGEMENT

- Buffer frame holds 1 data page perfectly
- Metadata Table: Frame ID (memory addr), Page ID (pg on frame), Dirty Bit (modified), Pin Count (# requestors)
- LRU: evict oldest used pg that is UNPINNED
- CLOCK: LRU approx, uses ref bit ↑ time
  1. Skip pinned until unpinned + ref bit 0 found
  2. If ref=0, evict, write, read new page + set ref bit to 1
  3. Move clock hand to next frame
  4. Hits: Set ref bit to 1, DON'T move hand
- MRU: evict most recently used ↑ sequential scans

Requestor sets dirty page
Page could have multiple pins
MRU > LRU sequential, = if P<F

SORTING

B buffer pages, (B-1) input, 1 output
Stop when 1 run

I/Os: $2N \cdot \left(1 + \log_{B-1}\frac{N}{B}\right)$ (# passes)

N=1000, B=11
Pass 0: 1000/11 = 91 runs, 11 each
Pass 1: 91/10 = 10 runs, 110 each
Pass 2: 10/10 = 1 run, 1100 each
(stop here)

pages to sort in p passes: $B(B-1)^{p-1} \ge N$
I/Os does NOT depend on # B pages

HASHING

Hash into (B-1) partitions, recurse until ≤ B, 1 for streaming in
Group By + Duplicate Values
Hash func need to be distinct + 1 for conquer phase
Data skew ↓
Hash w/o recursive part: B(B-1)
all unique values will be in 1 partition ALWAYS

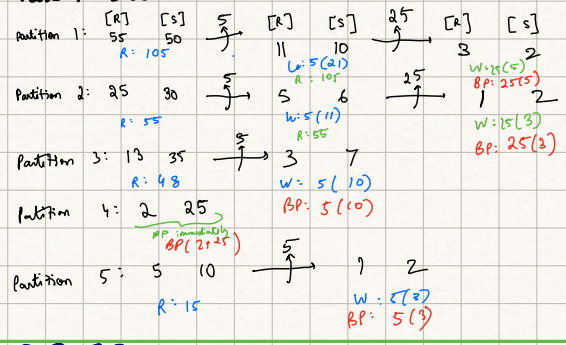
B_{10} buffers, p_3 passes ⇒ $(B-1)^{p-1}B$ = # max pages
$(9)(9)\,10 = 810$
(9)(9): part i, 10: conquer

JOINS PT 2

GRACE HASH JOIN:

1. Partition Data
   - → Partition R,S into (B-1) partitions
   - → Recurse until either ≤ (B-2)
2. Build and Probe
   - → Build in memory hash table for smaller one

Pass 1: 500



COSTS

	Heap	Sorted	Clustered
Scan All	B	B	$\frac{1}{\text{fill factor}} \cdot B$
Equality	B/2	$\log_2 B$	$\log_F\left(\frac{BR}{E}\right)+2$
Range	B	$\log_2 B$ + pages	$\log_F\left(\frac{BR}{E}\right)+3\cdot\#\text{pages}$
Insert	2B	$\log_2 B + B$	$\log_F\left(\frac{BR}{E}\right)+4$
Delete	$\frac{B}{2}+1$	$\log_2 B + B$	$\log_F\left(\frac{BR}{E}\right)+4$

× D time per scan

B: # Data Blocks
R: # Records / Block
F: Fanout
E: # Entries per leaf

QUERY OPTIMIZATION

Streaming operator: Work to produce each tuple

Blocking operator: Consume entire input THEN output generated

Selectivity Estimation

Pred	Selectivity
$C = V$	$1/\|C\|$, $1/10$
$C_1 = C_2$	$\frac{1}{\max(C_1, C_2)}$
$C \leq V$ INT	$\frac{V - \min_C}{\max_C - \min_C + 1} + \frac{1}{\|C\|}$
$C \geq V$ INT	$\frac{\max_C - V}{\max_C - \min_C + 1} + \frac{1}{\|C\|}$
$C \leq V$ FLOAT	$\frac{V - \min_C}{\max_C - \min_C}$
$C \geq V$ FLOAT	$\frac{\max_C - V}{\max_C - \min_C}$
P_1 AND P_2	$S(P_1) \times S(P_2)$
P_1 OR P_2	$S(P_1) + S(P_2) - S(P_1)S(P_2)$
NOT P	$1 - S(P)$

Join Selectivity

1. $\frac{|A| \cdot |B|}{\max(A, B)}$ (join sel)
2. $|A| \cdot |B|$ (no. of joined tuples)
3. $\frac{\text{tuples}}{\text{records per page}}$ (no. of pages)

Heuristic: Left deep, push down, NO cross join

System R:

Pass 1: Full scan / Index scan

Alt 1: # h + sel × leaves

Alt 2/3 clustered: # h + sel × leaves + sel × data

Alt 2/3 unclustered: # h + sel × leaves + sel × rec

Pass 2 ... N: Join (i-1) + 1 tables

Advance lowest + interesting orders → ORDER, GROUP, JOINS

↳ SMJ produces interesting order

↳ SNLJ, INLJ preserve left

B=5 [P] = 50, [T] = 100, P_R = 30, T_R = 40

mat both: (50+30) + (100+40) + 30 + $\lceil \frac{30}{3} \rceil$ 40

no mat: 50 + $\lceil \frac{30}{3} \rceil$ 100

TRANSACTIONS

Atomicity: all or none

Consistency: start & end consistent

Isolation: isolated from other Xact

Durability: commit should persist

1. Inconsistent Read (WR): reads only a part of the update
2. Lost (WW): multiple Xact write to same resource
3. Dirty Read (WR): reads update never committed
4. Unrepeatable Read (RW): same record read as different value

Conflict Serializability

↳ If dependency graph is acyclic

↳ Conflict: ≥ 1 write, dif Xact, same resource

↳ $T_i \longrightarrow T_j$: T_i came earlier, conflict

VIEW

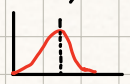
CONFLICT

Serial Equivalent: Same Xact, order, state

View Equivalent: Same intial read, dependent read, winning writes

2PL: Ensures conflict serializable

1. S before X 2. No acquiring after releasing



* If added to queue Xact is blocked

Strict 2PL: Prevents cascading aborts

1. Release all locks together

Lock Management: Hash table { Resource: (Granted, Mode, Queue }

Deadlock Avoidance (T_i waiting for T_j)

1. Wait Die: T_i higher priority → T_j waits; T_i lower priority → T_i aborts
2. Wound Wait: T_i higher priority → T_j aborts; T_i lower priority → T_i waits

Waits-For Graph

$T_i \longrightarrow T_j$: T_i waiting for T_j

Lock Granularity

Compatibility

out \	NL	IS	IX	S	SIX	X
NL	✓	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	✓	x
IX	✓	✓	✓	x	x	x
S	✓	✓	x	✓	x	x
SIX	✓	✓	x	x	x	x
X	✓	x	x	x	x	x

Parent

out \	NL	IS	IX	S	SIX	X
NL	✓	x	x	x	x	x
IS	✓	✓	x	✓	x	x
IX	✓	✓	✓	✓	✓	✓
S	✓	x	x	✓	x	x
SIX	✓	x	✓	x	x	✓
X	✓	x	x	x	x	x

RECOVERY

WAL: Log records to disk

1. BEFORE page flush
2. on commit ALWAYS

Page flush if: $pageLSN \leq flushedLSN$

Xact commit if: $lastLSN \leq flushedLSN$

In DPT, $recLSN \leq$ in mem LSN, $recLSN >$ on disk LSN

ARIES

1. Analysis: Recreate DPT, Xact Table
   - Start at begin checkpoint
   - Not End: Add to Xact, $lastLSN$ = LSN
   - Update/CLR Undo: Add to DPT (only if not in DPT), recLSN = LSN
   - Commit/Abort: Change status in Xact
   - End: Remove from Xact
   - At end, any committing txn: put END record, remove from Xact
   - running txn: put ABORT record, change to aborting
2. REDO (durability): Reconstruct state
   - Start at smallest recLSN
   - REDO ALL Update/CLR.
   - DON'T if
     1. Page not in DPT
     2. recLSN > LSN
     3. pageLSN (disk) ≥ LSN
3. UNDO (atomicity):
   - Start at end, move up
   - UNDOes all updates for Running/Aborting Xacts
   - ↳ Write CLR UNDO Tn: LSN prevLSN, undoNEXTLSN
   - ↳ Once all undone, write end record
   - ↳ Remove Xact

DISTRIBUTED TRANSACTIONS

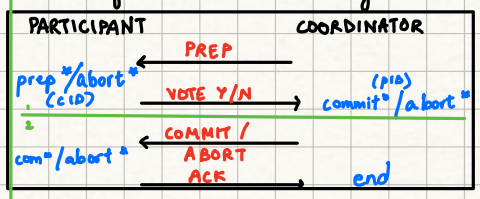
Distributed Locking

↳ Every node has own lock table (independent)

↳ UNION waits-for graph for distributed deadlock

2PC

↳ 2PC guarantees serializability



↳ Log + flush THEN send

↳ Presumed Abort: DON'T flush abort

↳ Commit ONLY happens when co logs commit

2PC Recovery

1. Participant Recovering
   - ↳ No prepare: abort
   - ↳ Prepare: ASK coord
   - ↳ Abort I: abort + vote NO abort
   - ↳ Abort II: abort + send ack abort
   - ↳ Commit: send ack
2. Coordinator Recovering
   - ↳ No commit: abort
   - ↳ End: Nothing
   - ↳ Abort: rerun phase 2 abort

PARALLEL QUERY PROCESSING

Architectures

1. Shared memory: CPUs share mem + disk
2. Shared disk: CPUs share disk
3. Shared nothing

Partitioning Schemes

↳ Sharding: 1 data page on 1 machine

↳ Replication: 1 data page on many machines

1. Range Partitioning: ↑ Key lookup, range
   - Each machine gets range of values
2. Hash Partitioning: ↑ Key lookup
   - Each machine gets hashed values
3. Round Robin: ↑ parallelisation ↓ all req activation
   - Each machine gets same # pages

Parallelism

↳ Inter-query: ↑ throughput

Each machine dif queries

↳ Intra-query

One query across machines

↳ Intra-operator

One operator fast as possible

↳ Inter-operator

Operators in parallel

1. Pipeline (pass to parent)
2. Bushy Tree (dif branches ||)

Network Cost

Data sent over network $\frac{pK(n-1)}{n}$

Parallel Sorting: Range partition + local sort

$\#Passes = 1 + \left(1 + \log_{B-1} \frac{N}{mB}\right)$

Broadcast Join: $(n-1) \cdot [P]$, P smaller table.

Parallel Hashing: Hash partition + local hash

Parallel SMJ: Range partition BOTH + local SMJ

$\#Passes: 1 + 1 + \#passes_R + \#passes_S + 1 + 1$

$I/Os: [R] + [S] + \left(1 + \log_{B-1} \frac{[R]}{B}\right) + \left(1 + \log_{B-1} \frac{[S]}{B}\right) + [R] + [S]$

pipeline breakers (wait for all data)

Parallel GHJ: Hash partition BOTH + local GHJ

Symmetric Hash Join:

1. Build 2 hash tables (* Partitions should fit in memory)
2. R/S record arrives, probe S/R

Hierarchical Aggregation:

↳ COUNT → m: count c: sum

↳ AVG → m: sum + count c: add + divide

1. Steal / No-steal (Atomicity)

Can modified, uncommitted Xact be flushed?

↳ Steal: Yes (UNDO)

↳ No-steal: No (bad performance)

2. Force / No-force (Durability)

Are modified pages forced to disk on commit?

↳ Force: Yes

↳ No-Force: No (REDO)

	No Steal	Steal
No Force	REDO	UNDO REDO
Force	∅	UNDO

Undo Logging: Steal / Force

Dirty pages to disk B4 commit log

Flush pages B4 commit

<T, A, Prev>

If T com/ab → completed else write to disk

Redo Logging: NF / NS

commit B4 flush: redo if crash after commit but not flush

<T, A, next val>

ignore uncom txn com: write new v

NOSQL

↳ Online Transaction Processing (OLTP):
- → high no. of transactions en large no. of users
- → workload: frontend (social net, online stores)
- → queries: simple lookups, updates ↑

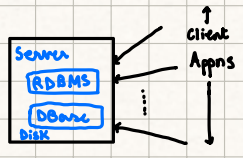
↳ Online Analytical Processing (OLAP):
- → read only workloads
- → large amount of data
- → large no. of joins + aggregations

	OLTP	OLAP
main read	small no. of rec/ query	aggregate over large no. of rec
main write	rand. access, low latency	bulk import or event stream
used	end user/customer via web app	internal analysis decision support
data	current state	history
size	GB – TB	TB - PB

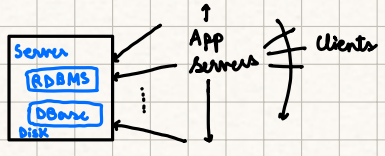
Architectures:

1. One-Tier Architecture

2. Two-Tier Architecture



3. Three-Tier Architecture



Partitioning / Sharding: Efficient for write-heavy workloads
Replication: Scale, resilient, enterprise down time
CAP Theorem:

↳ Consistency: 2 clients simultaneous get same view
↳ Availability: Every request → Not a faulty response
↳ Partition Tolerance: Continue to operate despite drop/delay

Impossible for > 2/3 to be satisfied
↑ Consistency ⇒ Error ↑ Availability ⇒ Stale