COMPRESS (usually 2x perf boost!)

Want: fixed-len compression (unless external data), late decompression when querying, lossless

- Naive: LZO, LZ4, ... (block level)

Col-level compression below

- Run-len (better if sorted) (RLE)
  [val, start idx, length]
- Bit pack: e.g. try use i32 instead of i64 if all fit
- Mostly/patching: try bit pack, store outliers elsewhere
- Bitmap: one-hot, with each unique val in col being vocab (only good if cardinality of col small)
- Delta: store differences, with base value somewhere
  even better: sort, delta, then RLE
- Incremental: delta but common prefix / suffixes & len recorded to reduce dups (good if sorted)
  [len of prefix same as above, suffix]
- Dictionary: map distinct vals in col to shorter id
  need to preserve order for query (so id ≠ hash
  destruct: array, hash table, BTree
  (sort & store str ptrs; expensive update; fast, no range query; slower, good range query)

Buf Pool

Global alloc policy: consider all active transactions
Local .. : try make curr trans. faster

Optim

- Multi-pool: per-db, per-page type,
  Which pool (balancing): by obj-id | by hash
- Pre-fetching: sequential | tree traversal
  Either case figure it out as DBMS
- Scan sharing: queries look same place buf together
  - Cont. scan sharing: keep scanning table like stream and let queries hop on
  - Light scan aka buf pool bypass:
    scan on disk but not put in buf

Replacement Policies

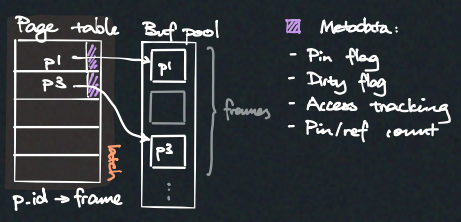
- LRU: keep list sorted/queue by last access
- MRU: most recently used
- Clock: approximate LRU. (or a counter)
  each page gets a ref bit, access sets it 1
  evict: sweep clockwise, decrement those with ref, try evict first 0.

Problems of LRU & Clock: seq flooding, ignores frequency

- LRU-K: track last k access
  evict page with oldest k^{th} access
  can cache recently evicted to bring back history
  - SQL LRU-2 approx.: young list + old list
- Localisation: evict on per-query/trans basis, less pollution
- Priority hints: transection tells buf what's important

Dirty handling

→ Background process keep writing dirty
→ When writing, try bypass OS page cache



AGGREGATION

DISTINCT

→ sort then dedup (good if need sorted distinct)
→ hash (faster)
1. Partition using h_1 into buckets on disk (B-1 of them)
2. Rehash with h_2 into hash table on mem

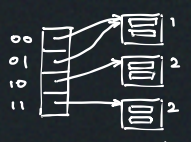
Hash Table

Scheme: collision handling Load fac: $\frac{\text{filled}}{\text{slots}}$

- Static hashing (viz. fix sized) > open addressing
  - Linear probing (mem)
    - probe from hashed loc, use tombstone when del
    - for non-unique keys: separate list | just put multiple
  - Cuckoo (mem) use h_1, h_2
    - evict: kick random one & rehash
    - if eviction cycle: double size & rebuild

  If out of space, often double size of htable
- Dynamic hashing
  - Chained Hashing (mem & disk)

  - Extendible Hashing (disk)



overflow:
local depth < global — just split
else double global depth

  - Linear hashing (disk)

$h_0(k) = k \% n$
$h_1(k) = k \% (2n)$

    - overflow ptr starting bucket 0, incr & split if overflow
    - can revert split if highest bucket empty

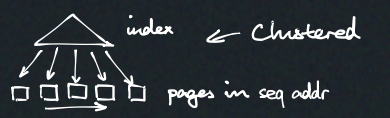
B+ Tree

- Perfectly balanced M-way search tree (M = fanout)
- Half full: leaf ≥ $\lceil \frac{M-1}{2} \rceil$ keys internal ≥ $\lceil \frac{M}{2} \rceil$ ptrs
- Keys all sorted (NULL before | after everything)
- Leaf node vals: record id | tuple data

INS
1. find leaf L
2. add to arr
   if overflow, split L ↝ L, L_2, rebalance, copy up middle key, fix parent

DEL
1. find leaf L
2. remove
   if underflow, try borrow from sibling, else merge with sibling & fix parent
   if needed pull down from root

- Dup keys: overflow node | append record id



Latching — lat. parent, lat. cld, release parent if cld safe
- safe if will not split or merge

Ins/del: if cld safe, release all parent latches
- opt: as if reading in first time, only W lat. on leaf, if leaf not safe — fallback too all W lat.

Horizontal scan: kill self if hitting node w W lat.

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: $2\{M,N\}(1 + \lceil \log_{B-1} \lceil \frac{\{M,N\}}{B} \rceil \rceil)$
  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 $M+N$

DB & SQL

- DBMS (DB Management Sys)
  - Relation — unordered
  - Rel/table with n attributes/cols — n-ary relation
  - Tuple is set of attri values (aka domain) in rel
  - Disk: block addressable, often 4kB/block, want seq. access
- Rel algebra (on sets)
  - SELECT $\sigma_{predicate}(R)$
  - PROJECT $\pi_{R.a+1,\ R.c}(R)$
  - R×S CROSS JOIN
  - UNION ALL allows dups, UNION dedups
  - EXCEPT R-S viz. difference
  - NATURAL JOIN — common attris
    JOIN S USING (a, b)
    JOIN S ON R.a = R.b AND R.b = S.b

- Rel model
  + structure, integrity, manipulation, durable
- SQL (on bags viz. unordered!, allows dups)
  - DML (Data Manipulation Lang)
    - Declarative — specifies <u>what</u> to find (rel calc.)
    - Procedural — write the <u>steps</u> to find (rel alg.)
  - DDL (Data Definition Lang)
    └ schema, indices, views, ...
  - DCL (Data Control Lang)
    └ security, access, ...
  - integrity, ref constraints, transactions

Modern SQL

⚠ WHERE filters for FROM, HAVING filters after GROUP BY
... group by e.cid having avg(s.gpa) > 3.9;
⚠ non-agg values after SELECT must have been grouped

% - any substring _ - any single char || - concat

```
select T1 into T2 ...; — saves in new table
insert into T2 (select ... from T1); — insert to existing
select ... limit 20 offset 10; — get 20 but skip first 10
select ... fetch first 10 rows with ties;
```

Window: can use ROW_NUMBER(), RANK() (↓ pre sorting, ↓ post sorting)
select... over (partition by... order by...)

Nesting in | any (has row) | all | exists | not exists
select ... from T1 where a in (select a from T2);
Lateral
select * from course as c
  lateral (select count(*) as cnt from enrolled
           where enrolled.cid = c.cid) as t1,
  lateral (... c.cid) as t2;

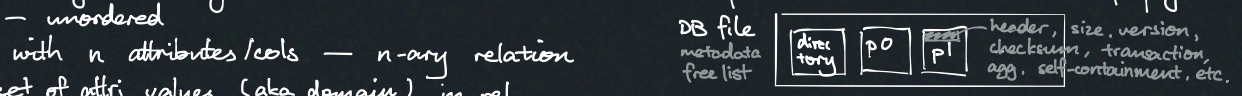
with Tmp(c1, c2) as (select ...) select c1+c2 from Tmp;
with Tmp1(...) as (...), Tmp2(...) as (...) ...

Extern Merge Sort

passes = $1 + \lceil \log_{B-1} \lceil \frac{N}{B} \rceil \rceil$
IO cost = $2N \cdot$ # passes
first run len = B, other ones len = B - 1
→ Double buffering: needs 2x buf size
→ Clustered B-Tree can be faster if exists, not unclustered

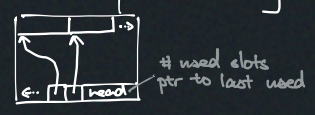
File Storage

▸ Heap file — unordered collection of pages



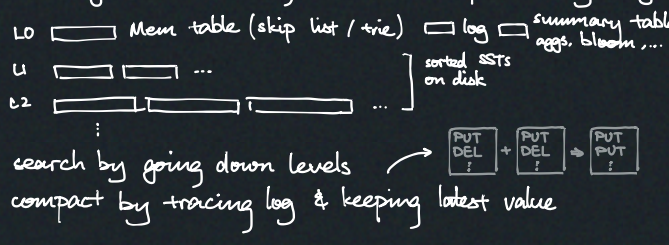
Access: p-id ↦ address

▸ Page structure
→ fixed length, can do # tuples + array
→ Slotted page

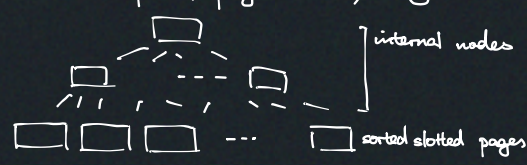


(tuple oriented problems: fragmentation, useless IO, rand. IO)

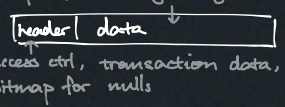
→ Log structured, fast write slow read, com amplify write
LSFS (log struc file sys) LSM Tree (log struct merge)
flush logs to Sorted String Tables (SSTs), periodic log merge



→ Index organised
BTree kV pairs, pages at leaf, O(lgn) search ins del



▸ Tuple layout

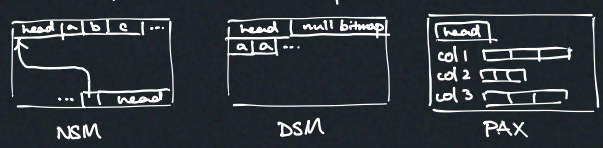


Tuple identifier, usually p_id + slot_id / offset
Denormalise: pre-join tuples from multiple tables

Storage Model

▸ N-ary (NSM)
- space locality, col compression, useless IO
+ fast ins/del of rows

▸ Decomposition (DSM)
+ fast row access, OLAP, col compression
- select *, OLAP, using many cols ⇒ buf pool stress
└ tuple id: fixed-len offset (pad var len) | embedded IDs
└ var-len data: key into extern dict
└ use per-col null bitmap



▸ PAX (hybrid)

Workloads

OLTP — __ Transaction Processing
OLAP — __ Analytical ...
HTAP — hybrid

SORTING

▸ Fit → in mem quicksort
▸ Top-n heap sort (keep in-mem priority queue)
▸ Else: below