1 Hardware

1.1 Disk Access time

- Disk access time: seek + rotational + transfer
- seek time: move arms to correct track. 5-6ms fixed
- rotational delay: rotate track to correct block. avg 1/2 round

1.2 DB terminology

- Buffer pool: cache. split into frames, each frame same size as blocks.
- Frames: store pin count (no. clients using) + dirty flag(modified but not in disk)
- Disk blocks: aka pages. unit of retrieval, consist of contiguous sectors.
- Clock replacement: if pin count == 0: if referenced bit off evict else turn referenced bit off. Does not reset after an op.
- notation: $|A|$ number of pages to store A, $||A||$ number of records in A.

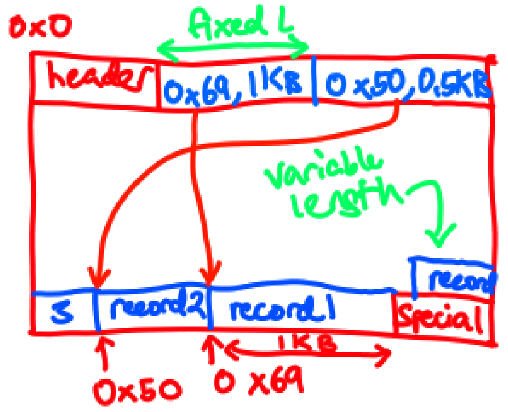
1.3 File implementation

Each record has an id called RID (page id + slot number). 1 page many slots.

1. Heap file, unordered. page header store 2 linked list of pages. 1 linked list free space, 1 used.
2. Sorted file, ordered by search key
3. Hashed file, locate block by hash function.

Record organization in pages

1. fixed, packed: store records in contiguous slots. header store number of records
2. fixed, unpacked: use bit array to indicate if slot if full, can choose any slots.
3. variable length



2 Indexes

Data structures to speed up info retrieval. search key = sequence of attributes. unique index = search key is candidate key.

2.1 B+ tree

formats:

1. key, actual record
2. key, record id
3. key, array of record ids

notes:

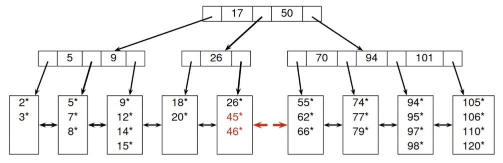
- internal node, ¡ lhs, ≥ rhs
- non leaf nodes are not values: 5 → [4][7], 5 does not exist as record
- order `d` b+ tree: nodes contain at least `d` nodes, at most `2d` nodes
- No dupes. irl usually take some unique attrib also.
- clustered index: sort order of index is same as table. e.g. sort on weight, weight 1,2,3 on same data page. (at most 1 clustered index since actual data must sort 1 way.)
- dense index: index record (key + pointer) exists for every search key value. (therefore format 2,3 sure dense)

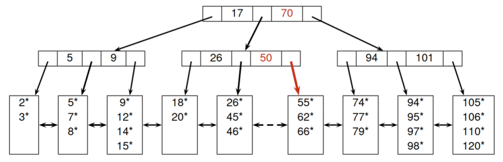
2.1.1 Insert

1. if page not full, insert.
2. check right neighbor (must same parent) then left
3. if not full, redistribute (largest record go over, update internal node)
4. split page. order `d` table, `d` records in original page, `d+1` in new page. add entry in internal node.
5. if internal node full, repeat steps above for internal. internal node split, promote middle node 1 level.

2.1.2 Delete

same as insert. check right then left. redistribute if neighbor size ¿ `d`. Merge if cannot (delete internal node between).





2.1.3 Bulk loading

1. sort records, pack into pages of `2d` records
2. keep inserting, add internal node record each time. split if need be.

2.2 Hashed based

static hashing: `N` buckets, hash is simple modulo. linked list of overflow data pages.

2.2.1 Linear hashing

`N`: number of buckets. `next`: next bucket to split. N_{level}: number of pages at level. starts at 0. bucket index starts at 0. Only split when a page overflows. If overflow page not full do not split.

1. Check if bucket `j` to insert to is full. Insert if not.
2. `next` is not `j`, split bucket `next`. `next++`. insert page into overflow page.
3. if `next` is `j`, split bucket. `next++`. If bucket to insert into is still full, insert into overflow page.
4. if `next` is N_{level} during split, set `next` to 0, `level++`. $N_{\text{level}} = 2^{\text{level}} - 1$.

To cause max splits, insert numbers such that after split, 1 bucket just nice full. Deletion, if last bucket empty, delete and decrement next if next ¿ 0, else decrement level.

2.2.2 Extendible hashing

like linear hashing, except keep directory of pointers to buckets.

1. Insert `01`. no overflow ok.
2. overflow, double directory size. `001`, `101` point to separate buckets
3. other keys e.g. `000`, `100` done need new bucket, point to original `00` bucket.

note: last bucket of split must contain bucket size + 1 items. possible that 4 pointer to same bucket split into 2 buckets with 2 pointers each.

3 sort select

3.1 external sorting

- External sorting of `x` data pages using `m` buffer/memory pages (x » `m`)

  1. Create sorted runs, each run size `m`. page cost: $2 * x$.
  2. Merge runs `m - 1` ways. 1 output page rest input.
  3. Selection sort, write to disk when output full. page cost $= 2 * x * depth$.

- optimization, blocked I/O
  - system sometimes allow reading of `b` contiguous pages in 1 i/o operation (same seek + rotational, longer transfer)
  - io time: instead of $x * pageRWtime$ use $blocks * blockRWtime$ each pass. initial pass take block size as buffer size.

3.2 B+ tree sort select

- selectivity og access path: no. of index/data pages needed to get record. smallest usually = traverse down smallest value, iterate right.
- covering index for `Q`: index contains all info required, no need fetch with RID etc.
- `where ... and`: look up RID and filter or intersect with another b+ tree index.

3.2.1 conjuncts

- Conjunctive Normal Form: express in ands. conjuncts = `A or B`.
- index `K(k1, ... kn)` matches predicate `p` if:
  - b+ tree: `p` is of the form `(k1 = c1) && ...(Ki = ci) && (Kj op cj)`. At most 1 non `==` op, must be last attrib. does not have to be all of index.
  - hash index: like b+ but no non `==` + all attributes used (kb = cn).
- `where ... and`: look up RID and filter or intersect with another b+ tree index.
- primary conjuncts: subset of conjuncts in `p` that index **matches**.
- covered conjuncts: conjuncts in `p` that index covers. conjuncts may not be primary (see matches def.)

3.2.2 cost

- b+ tree: pages accessed = leaf depth + no leaf pages to scan. if format 2/3, + no. of entries(need lookup each)
- hash: scan leaf pages(num entries to get / num entries per leaf page)
- if index is covering, no need RID lookup even if format 2/3.

best query plan: consider table scan, index scan, index intersection.

4 projection

Pick subset of attributes then dedupe. project with * mean no dedupe.

4.0.1 sort based

1. extract all columns needed. cost = `R` read num pages original + `T` write num pages intermediate table.
2. sort all column `2 * T * depth`. depth inclusive initial sorted.
3. scan and dedupe, read `T` write omitted (return).
4. optimization: create initial sorted runs at step 1, dedupe during merge. no dedupe cost, merge depth exclude initial.
5. if exists index (preferably covering/-clustered) can use also try index scan. if unclustered, for each tuple need RID lookup so very expensive. clustered can scan by pages for same sort key.
6. if index only partial, can use values with same sort key as partition, project within partition.

5 joins

Outer join Inner, R join S. can swap R and S.

5.0.1 iteration

1. For each record in outer (cost = $|R|$)
2. scan each record in inner. cost = $||R|| * |S|$ (can optimize for each page $|R|$)
3. block nested, 1 output 1 `S`, rest `R`. cost = $|R|/(B-2) * |S|$

5.0.2 Sort merge join

1. sort inner + outer
2. join. may need rewind if dupes: R = 10,10,20 S = 10, 10, 15. cost = $|R| + |S|$
3. if $|buffer|$ ¿ # initial runs `R` & `S`, after initial sorted run can join already.

5.0.3 Hash join

1. hash inner, outer into `k` partitions each
2. say inner.A = 1 in partition 1 and outer.A = 1 in partition 3, join partition 1 + 3.
3. 1 output 1 input, rest use as hash table for inner. probe outer into input, write to output. output full write to disk.
4. cost: $3 * (numPagesInner + numPagesOuter)$.
5. $k = B - 1$, $B \approx \sqrt[2]{f * numPagesInner}$, `f` is fudge factor.