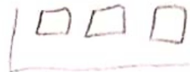
Data Systems : Index   Lecture 7

> Why do we need indices? — What is an index? — Why have an index.



• Data / file / Relation — stored on a non-volatile storage — Access

- search, query, quick access.
- unique mapping — ordering on data.

| what makes an index an index?
> Notion of defining index — what are its pre-requisites.

• Relations

- unique way to access a row
- pointer to a row of data.

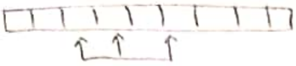
> Say I have only queries that need to access all rows ⇒ Usefulness of an index.

n bit vector

2^n values

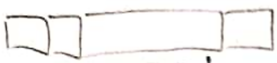
> interested in only a few bits. — access only based on these bits

→ which are the bits I am interested in?



q(, ,) : what kind of an index? what about masking?

10^{10} rows, N queries / min.



one column is an n-bit vector

Index

Π Relation ⟷ : Column Semantics

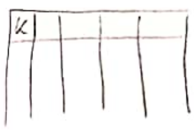
Query

↲ Access column values as they are : query & column semantics must match.

"threesome game"
Index + Query + Relation
Column Semantics

↳

Primary Key : K
ordered OR unordered



Attribute : key or non-key
↓
duplicate entries are possible

Primary Index : On K, file ordered on K, index on K

Disk Block.

In a disk block, rows are ordered on key K.

• Only need pointers to blocks

Points to blocks

block anchor

b_i

b_{i+1}

l : length of record

B : block size

bfr ≡ how many records can fit in a particular block

n_R : number of records

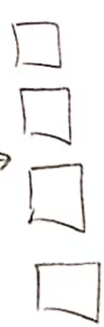
— Can calculate # required blocks

n_{PI} : # records in primary index blocks

$(\lg_2 n_{PI}) + 1$

Index Entry : (Key, block_ptr)

n blocks →



• n such entries

> All on Primary Index

File Ordered on Attribute A (non-key)..

(A = value) → many possible rows. — Index on A.

A's values: $V_1, V_2, \ldots, V_n$

▷ How many blocks to select: $\sigma_{(A=V_i)}(R)$? fixed block size.

n_c blocks

$lg_2(n_c)$ → points to block [|] with first $A = V_i$.

◦ How many rows have $A = V_i$?

n_i rows: bfr number of rows in each block: $\left\lceil \frac{n_i}{bfr} \right\rceil$

$lg_2(n_c) + \left\lceil \frac{n_i}{bfr} \right\rceil$ → the max value

→ Need actual number $|\{A : A = V_i\}|$

• Cluster Index: Rows put together in Attribute = Value manner one after another and then they are grouped.

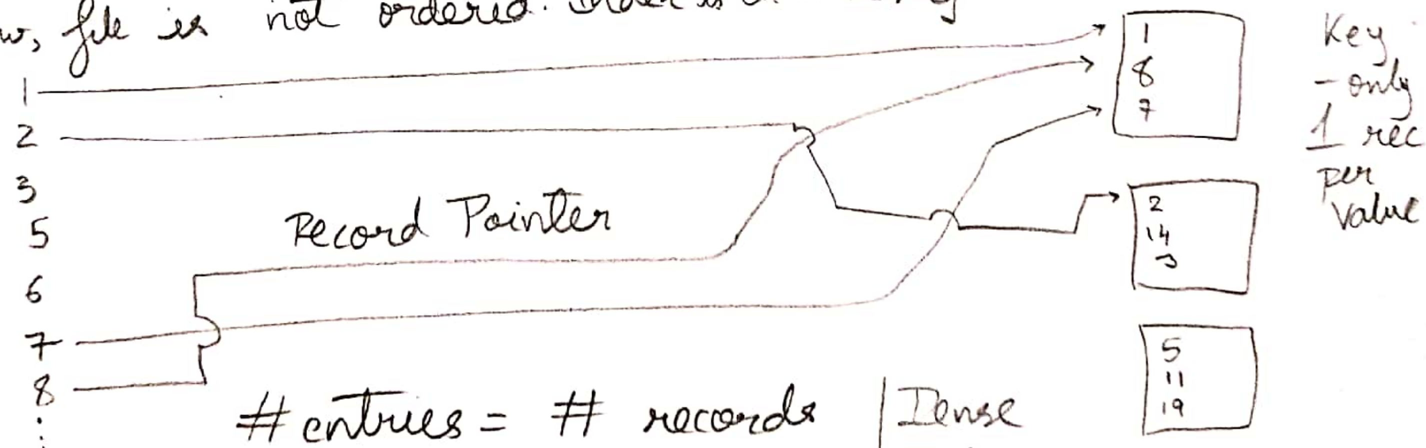
→ Two Types — See Slides.

• How are deletions handled? $Z \in dom(A)$

— Point to NULL. When a record comes, insert it there. Tells if there are records of that particular value or not. Exists or not — just check for NULL pointer. How Index & query cooperate?: power & semantics of indexing. Can also store counts in the index.

— Primary & Clustering Index are on ordered files.

→ Now, file is not ordered. Index is on the key.



entries = # records | Dense Index
~~# blocks~~

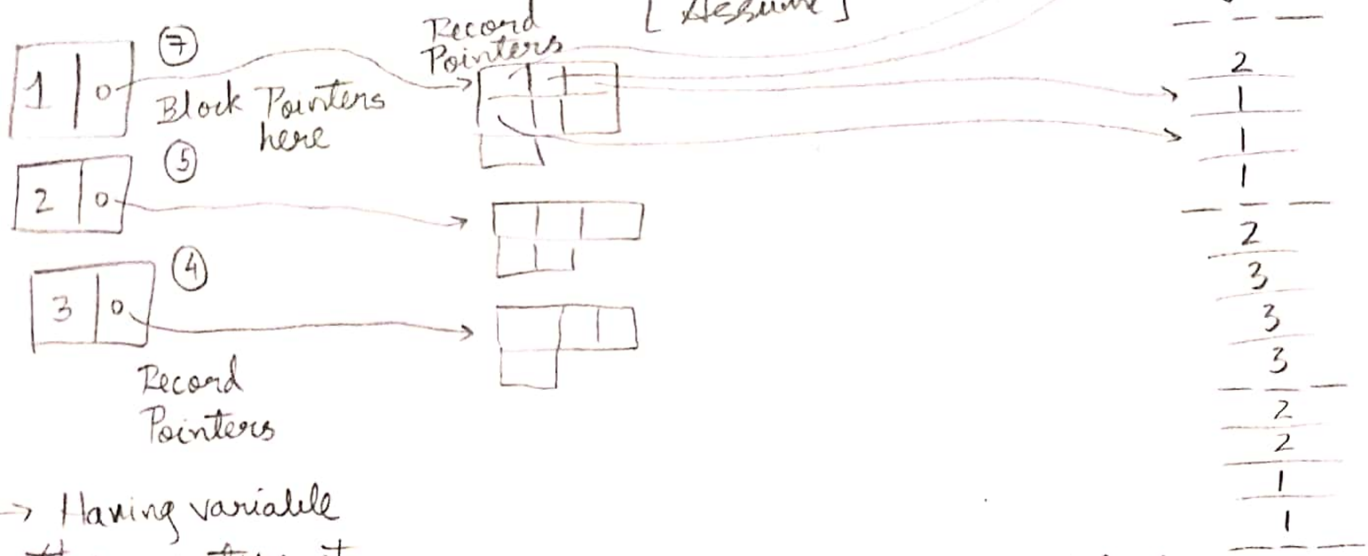
$lg_2(n_{SI}) + 1$

↓

blocks in secondary index.

→ File is unordered − Index is on non-key.

V_i : 1, 2, 3 : One value (index entry per block) [Assume]



→ Having variable # of pointers at this stage is clumsy.

||

n_A blocks in stage 1

$\sigma_{A=v}(R)$: How many block accesses do we need?

Indirection blocks. Contain only record pointers

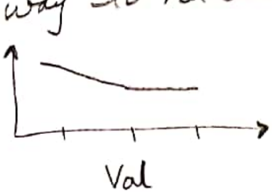
[Might need to access each block.]

$$\lg_2(n_A) + \left\{ \begin{array}{c} \#\text{indirection blocks} \\ \text{for } V_i \end{array} \right\} + \#\text{records}$$

#indirection blocks

Suppose: $\sigma_{A > Val}(R)$ − is this a good way to handle such queries?

$A \geq 8$, $A \leq 2$, $A > 3$: cases.



optimize based on dataset knowledge.

− if number of values per A is very small, use a level 1 non dense indexing scheme.

ATM : Fast cash vs withdrawal − what kind of indexing will they use? Which one would be faster? SBI : 350×10^6+ bank accounts.

− Maybe a defined hardware o/p for fast cash − as balance update will typically cost the same. − custom generated SQL query.

− what about indexing? | : Canned Queries · Pre-compiled queries

− PI on Account Number

− Clustering Index or Dense Index − will be too difficult.

∘ Hash Table on Account Number.

Relational Database : Key (Unique Values) → Value

4 kinds of indices. – File: Ordered, Unordered.

File	on key | index on key	on A | index on A
Ordered	P.I, m index blocks, $\lg_2(m) + 1$	C.I, $v_1\ v_2\ \dots\ v_n$, m blocks, $\lg_2(m) + \left\lceil \frac{n_{A=v_i}}{bfr} \right\rceil$
Unordered	$\square\ v_i \rightarrow \square\ K = v_i$, $\lg_2(m) + 1$, S.I on Key	$\boxed{v_1} \rightarrow n_{A=v_i}$ ptrs, $v_1, v_2, \dots, v_i$, $n_{A=v_i}$ rows, V_K; $A = v_i$ is not pre-known: $\lg_2(m) + \left\lceil \frac{n_{A=v_i}}{\text{bfr of rec. ptr}} \right\rceil + n_{A=v_i}$, S.I on non key

Length of record or row.
- Actual constant len
- Average length of record

Record : rows of reln.
$R(K, A_1, \dots, A_n)$
$r_l\ (K_{byte} + A_{1\,byte} + \dots + A_{n\,byte})$

$$bfr = \left\lfloor \frac{B}{r_l} \right\rfloor \text{ rows / block.}$$

B_find	R	PI	CI	SI on Key	SI on A
	r_l	K_{bytes} + block ptr	length of A_{bytes} + block ptr	len of key + record ptr	

Index block
len of A + block ptr
↓
Indirection block
length of record pointer

≡ Will give different bfr (s) based on type of index.

→ structure of index
→ bfr — know number of index entries
↓
infer how many blocks do I need?

· Index : Ordered File. $n_I > 1$ blocks → Index

Number of levels: $\lg_{bfr_a}(n_i)$
block accesses = # levels + 1

bfr_a

till we have index with 1 block ←

Index on index

n_I = number of blocks in the lowest level index

File

Create multilevel index on any kind of index PI, CI, SI ...
— Acts like a search tree — faster access: but packed, filled blocks
:— insertion may involve rearrangement. — costly. Deletions are also painful.
— Streamline the notion of multilevel index — well regulated manner

Node of B blocks : | ≡ 1st well designed index. — how block should be indexed.
— how should it be organised

entries a_1 — ☐
a_2 — ☐

a_p — ☐

in a block
value | pointer

organize as. ↳ B tree.

$< P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, P_3, \ldots$
$\ldots <K_{q-1}, Pr_{q-1}>, P_q >$

block Pointers — Key — Record Pointers

- q – block ptrs
- q – 1 – record ptrs
- q – 1 – keys.

$K_1 < K_2 < \ldots < K_{q-1}$

$X < K_1 \rightarrow P_1$
$K_{i-1} < X < K_i \rightarrow P_i$
$X > K_{q-1} \rightarrow P_q$
$key = K_j \rightarrow Pr_j$

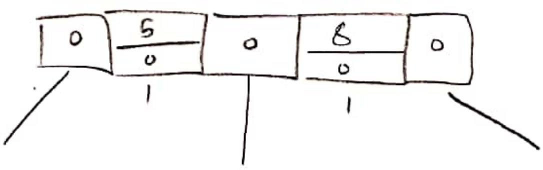
block pointed key

B bytes to play with.

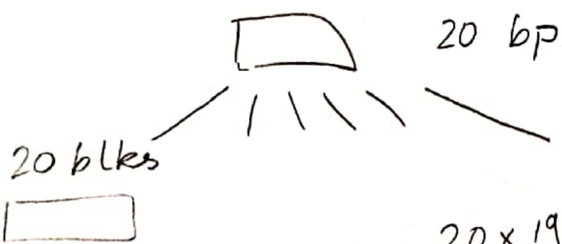
Max Number of Keys & ptrs in B bytes.

$$\boxed{q \times l_{block\text{-}ptr} + (q-1) \times (l_{rec\text{-}ptr} + l_{key}) \le B}$$

: Case for q = 3. | 3 block ptrs, 2 record keys, 2 record ptrs.
— All blocks have same structure



— q = 20



20 bp. 19 rp level 1

20 blks

20 × 20 bp level 2

20 × 19 rp

20 bp
19 rp

bp	rp	level
20	19	1
420	399	2
8420	7999	3
24420	23199	4

Number of block Accesses is variable. — luck.

< 22000 records $\rightarrow$ 4 levels is enough.

— beauty of this designer index

: keep 2/3 rd full.

$q = 2/3 \times 20 \approx 13.$

$q = 13$ blk ptr
12 record ptr

$\rightarrow$ to have space for new records

bp	rp	level
13	12	1
$13 + 13\times 13$	$12 + 13\times 12$	2
$13 + 13\times 13 + 13\times 13\times 13$	$12 + 13\times 12 + 13\times 13\times 12$	3

B+ tree : leaf node & non-leaf Node design structure

1. Non leaf: $< P_1, K_1, P_2, \;\; , P_{q-1}, K_{q-1}, P_q >$
   All are Block ptrs. $\hookrightarrow$ A non leaf or leaf node.

- $P_1 \rightarrow \; X \le K_1$
- $P_q \rightarrow \; X > K_{q-1}$
- $P_i \rightarrow \; K_{i-1} < P_i \le K_i$

2. Leaf : $< <K_1, Pr_1>, <K_2, Pr_2>, \;\;\; <K_{q-1}, Pr_{q-1}>, \rightarrow$

to the sibling $\uparrow$ 1 block ptr $\downarrow$ fast traversal


K_1


K_2

- To build :
  non-leaf
  $\hookrightarrow$ leaf

Block size B : $q \times$ blk_ptrs + len_key $\times (q-1) \le B$

$\llcorner$ Non leaf

leaf : $q \times$ (len-key + len-recptr) + len-blkptr $\le B$

Levels: 1 30

2

3

compact,

flatter, more fan-out

\# levels + 1 : block accesses

See tutorial for Algorithms — Slides for summarised points.

Numericals : In GA3