Bitmap Indexes.

- **Assumption:**

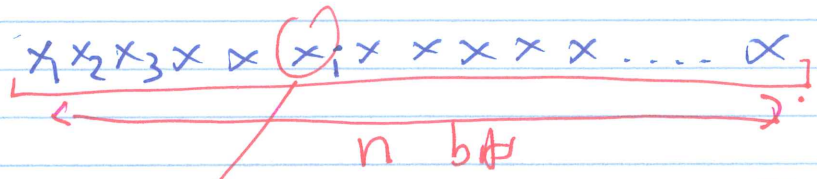
  Records in a file have a permanent number (position id) associated with each record.

  e.g.: ~~file~~ record position

- **Field F** = (domain?) A specific field in the record.

  = Current set of values stored in a field F in the record.

- **Bitmap index** for the field F =
  - a collection of bit vector of length n
    (n = # records in the file.
  - One bit vector for each possible value V that appear in the field F.
  - Bit vector for value V =



$x_i = 1$ if i^{th} record's field F = V.

Example:

File has 6 records: (A, B)

record position

A	B	
1 (30,	foo)	2 (30, bar) 3 (40, baz)
4 (50,	foo)	5 (40, bar) 6 (30, baz)

Bitmap index for field (A):

	1	2	3	4	5	6
30	1	1	0	0	0	1
40	0	0	1	0	1	0
50	0	0	0	1	0	0

Bitmap index for field (B):

	1	2	3	4	5	6
foo	1	0	0	1	0	0
bar	0	1	0	0	1	0
baz	0	0	1	0	0	1

Example: gold-jewelry data.

	Age	Salary
1:	25	60
2:	45	60
3:	50	75
4:	50	100
5:	50	120
6:	70	110
7:	85	140
8:	30	260
9:	25	400
10:	45	350
11:	50	275
12:	60	260

Bitmap index on Age:

	1	2	3	4	5	6	7	8	9	10	11	12
25:	1	0	0	0	0	0	0	0	1	0	0	0
30:	0	0	0	0	0	0	0	1	0	0	0	0
45:	0	1	0	0	0	0	0	0	0	0	1	0
50:	0	0	1	1	1	0	0	0	0	0	0	0
60:	0	0	0	0	0	0	0	0	0	0	0	1
70:	0	0	0	0	0	1	0	0	0	0	0	0
85:	0	0	0	0	0	0	1	0	0	0	0	0

Bitmap index on Salary:

	1	2	3	4	5	6	7	8	9	10	11	12
60:	1	1	0	0	0	0	0	0	0	0	0	0
75:	0	0	1	0	0	0	0	0	0	0	0	0
100:	0	0	0	1	0	0	0	0	0	0	0	0
110:	0	0	0	0	0	1	0	0	0	0	0	0
120	0	0	0	0	1	0	0	0	0	0	0	0
140:	0	0	0	0	0	0	1	0	0	0	0	0
260:	0	0	0	0	0	0	0	1	0	0	0	1
275:	0	0	0	0	0	0	0	0	0	0	1	0
350:	0	0	0	0	0	0	0	0	0	1	0	0
400:	0	0	0	0	0	0	0	0	1	0	0	0

Example using bit map index:

Find buyer (of jewelry) with:

- age = 50
- salary = 100.

	1 2 3 4 5 6 7 8 9 0 1 2
Bitmap index for age = 50 :	0 0 1 1 1 0 0 0 0 0 0 0
Bit map index for Salary = 100 :	0 0 0 0 1 0 0 0 0 0 0 0
AND	0 0 0 0 1 0 0 0 0 0 0 0

→ the 5th record satisfies all criteria.

(50, 120).

- Strength of Bitmap Indexes:

(1) Can answer partial match queries efficiently.

Eg:

Find buyers (of jewelry) with

age = 50.

Use bitmap index for

	1	2	3	4	5	6	7	8	9	10	11	12
age = 50:	0	0	1	1	1	0	0	0	0	0	0	0

↳ records 3, 4, 5

(2) Can answer range queries efficiently.

Eg: Find buyers (of jewelry) with:

$45 \leq age \leq 55$

$100 \leq salary \leq 200$

range query.

Solution:

$45 \leq age \leq 55$ has these values:

	1	2	3	4	5	6	7	8	9	10	11	12
age = 45 :	0	1	0	0	0	0	0	0	1	0	0	
age = 50 :	0	0	1	1	1	0	0	0	0	1	0	
OR	0	1	1	1	1	0	0	0	1	1	0	

$100 \leq salary \leq 200$ has these values:

	1	2	3	4	5	6	7	8	9	10	11	12
sal = 100 :	0	0	0	1	0	0	0	0	0	0	0	
sal = 110 :	0	0	0	0	0	1	0	0	0	0	0	
sal = 120 :	0	0	0	0	1	0	0	0	0	0	0	
sal = 140 :	0	0	0	0	0	0	1	0	0	0	0	
OR	0	0	0	1	1	1	1	0	0	0	0	

$(45 \leq age \leq 55)$ AND $(100 \leq salary \leq 200)$ has as solution.

	1	2	3	4	5	6	7	8	9	10	11	12
	0	1	1	1	1	0	0	0	0	1	1	0
AND	0	0	0	1	1	1	1	0	0	0	0	0
	0	0	0	1	1	0	0	0	0	0	0	0

↳ records 4, 5

Storing Bitmap Indexes:

- Naive way to store bitmap indexes is very inefficient:

Suppose: field F (file has n records)

↓

m different values:

$v_1 = x\,x\,x\,x\,x\,\ldots$
$v_2 = x\,x\,\ldots$
$\vdots$
$v_m = x\,x\,\ldots$ (n bits)

} m·n bits total.

- Note:

$v_1 = x\,x\,x\,x\,\ldots$
$v_2 = x\,x\,x\,x\,\ldots$
$\vdots$
$v_m = x\,x\,x\,\ldots\,x$

total # 1-bits = n.

total # 0-bits = $m \cdot n - n$

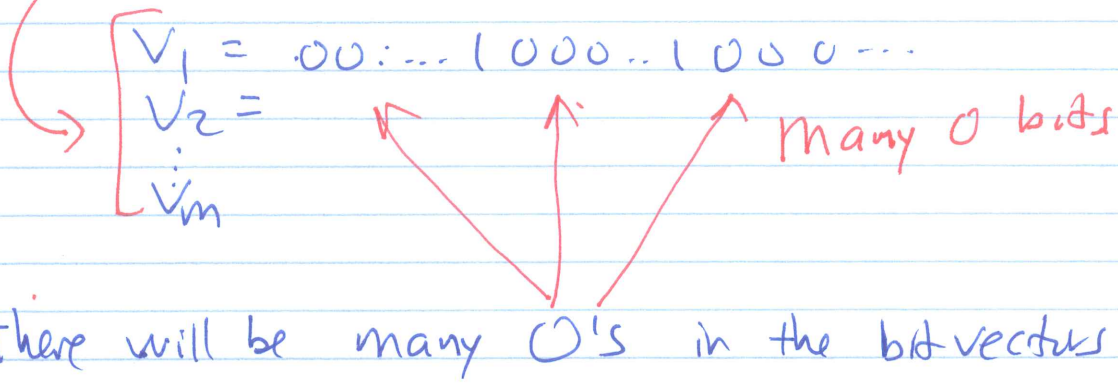
$= (m-1) \cdot n$.

(If m is large, you will have a lot of 0 bits !!!). → compression !!!

Compressed Bitmaps

- Observation: m = # possible values in a field F.

When m is large:



there will be many 0's in the bit vectors !!!

- Definition: Run

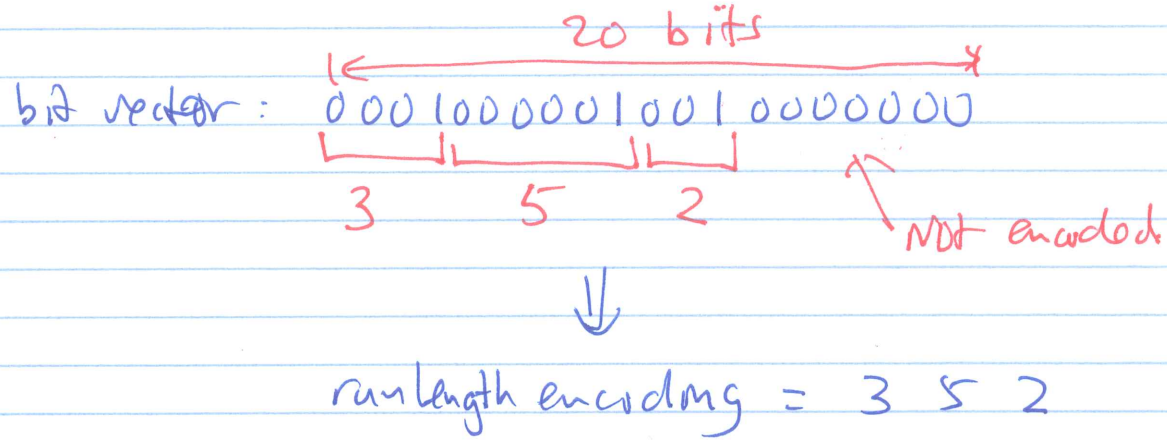
Run = a sequence of 0 bits followed by one (single) 1 bit.

eg: 1 — run length = 0
01 — run length = 1
0001 — run length = 4.

- Run length = # 0 bits in the run.

- Run length encoding = representation of a bit vector by the run lengths.

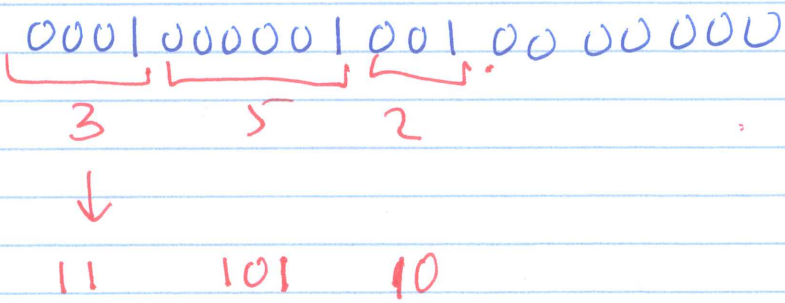
eg: $n = 20$ (20 bits in bit vector)



Fact:

- Run length encoding is done in binary.

So:



- $64,000 question:

How do we encode the run lengths???

How to represent the run length code in binary:

Fact: you cannot ~~simply~~ represent each run length as a binary number and simply concatenate the bits.

Example:

bit vector 1: 0 0 0 | 1 | 0 | 1

3 1

= 11 = 1 ⇒ code = 11 1

bit vector 2: 0 | 1 | 0 | 1 | 0 | 1 ⇒ code = 1 1 1

Ambiguous !!!

Run length encoding:

Run length = same binary number of (k) digits

encoding: 1 1 1 1 0 ← run length in binary

k bits.

Example:

run length = 13 binary = 1101

$8 + 4 + 1 = 13$.

Encoding: 1110 1101

Next 4 digits is the run length

13

Example Encoding:

$n = 20$.

bit vector: 0001 000001 001 00000000

3 5 2 Not coded

11 101 10.

1011 1010 1010

Decode

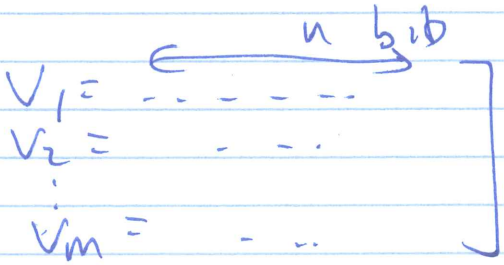
11 101 10

3 5 2.

(You need to remember $n = 20$ explicitly).

Analysis: # bits used in bitmap index.

Previously: (uncompressed).



bits = $m * n$.

Suppose $m = n$ (Every value occurs only once)

best case compression

worst case size for bit map.

uncompressed = n^2 bits.

Compressed:

V_1 = ... 1 . . .

V_2 = ... 1 . . .

avg # $\emptyset$ = $\frac{1}{2}n \approx n$.

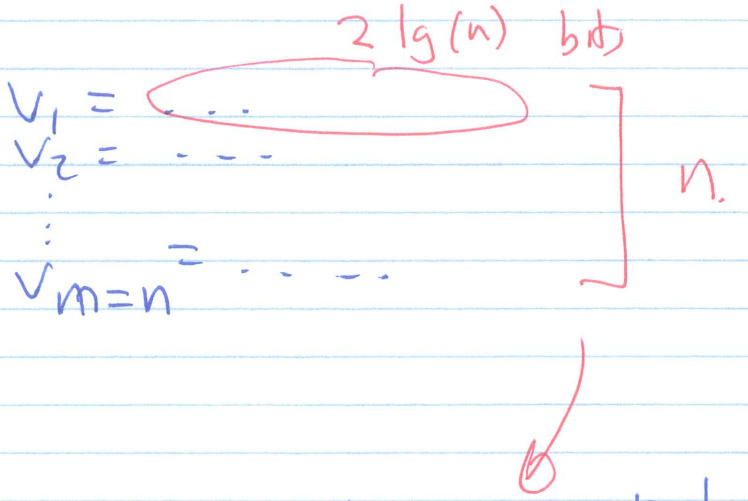
Needs $\lg(n)$ bits

to repr. ~~dist~~ number n.

Code for 1 bit vector: 111110 (last $\lg(n)$ bits)

($\lg(n)$ bits)

$= 2\lg(n)$ bits.

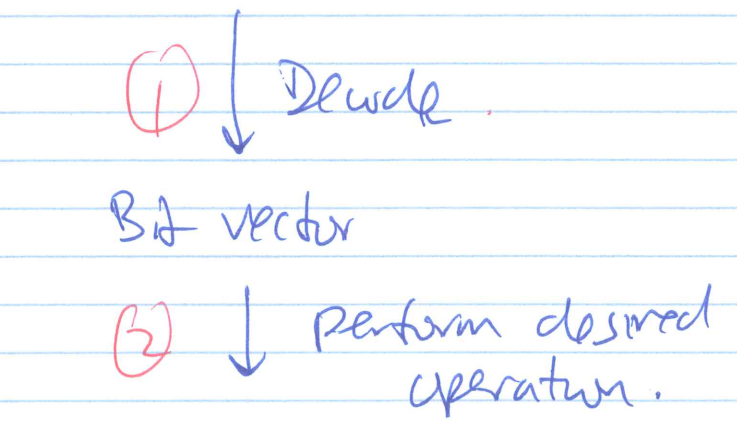


Total # bits = $2n\lg(n)$ bits

(vs. n^2 # bits) !!!

Operations on Run-length encoded Bit-vectors:

- There is overhead involved to perform operations on Run-length encoded Bit vectors:



- Trick:

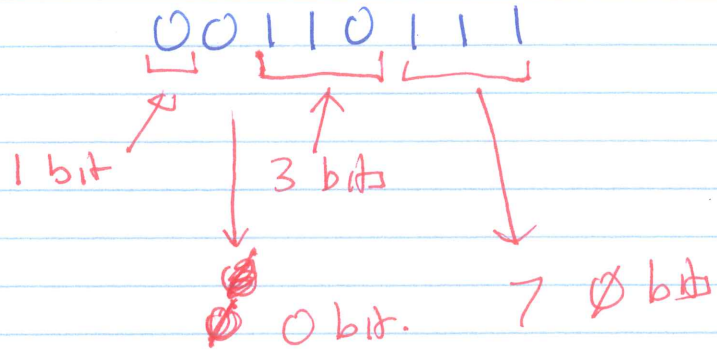
  "Just-in-time" decoding.

Example: OR on these Encoded bitvectors:

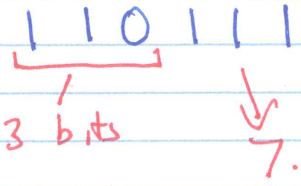
vector 1: 0 0 1 1 0 1 1 1 (n=12)
vector 2: 1 1 0 1 1 1

Naive: decode and then perform OR

decode:

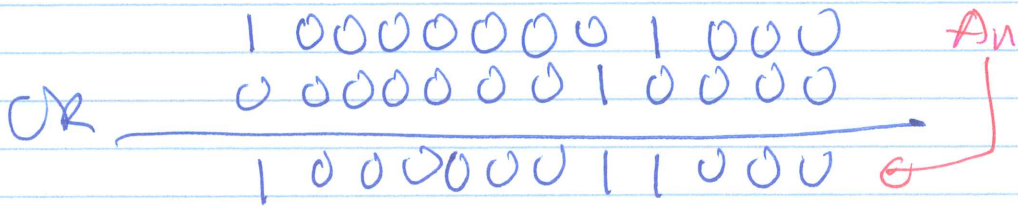


⇒ 1 0 0 0 0 0 0 0 0 1 0 0 0

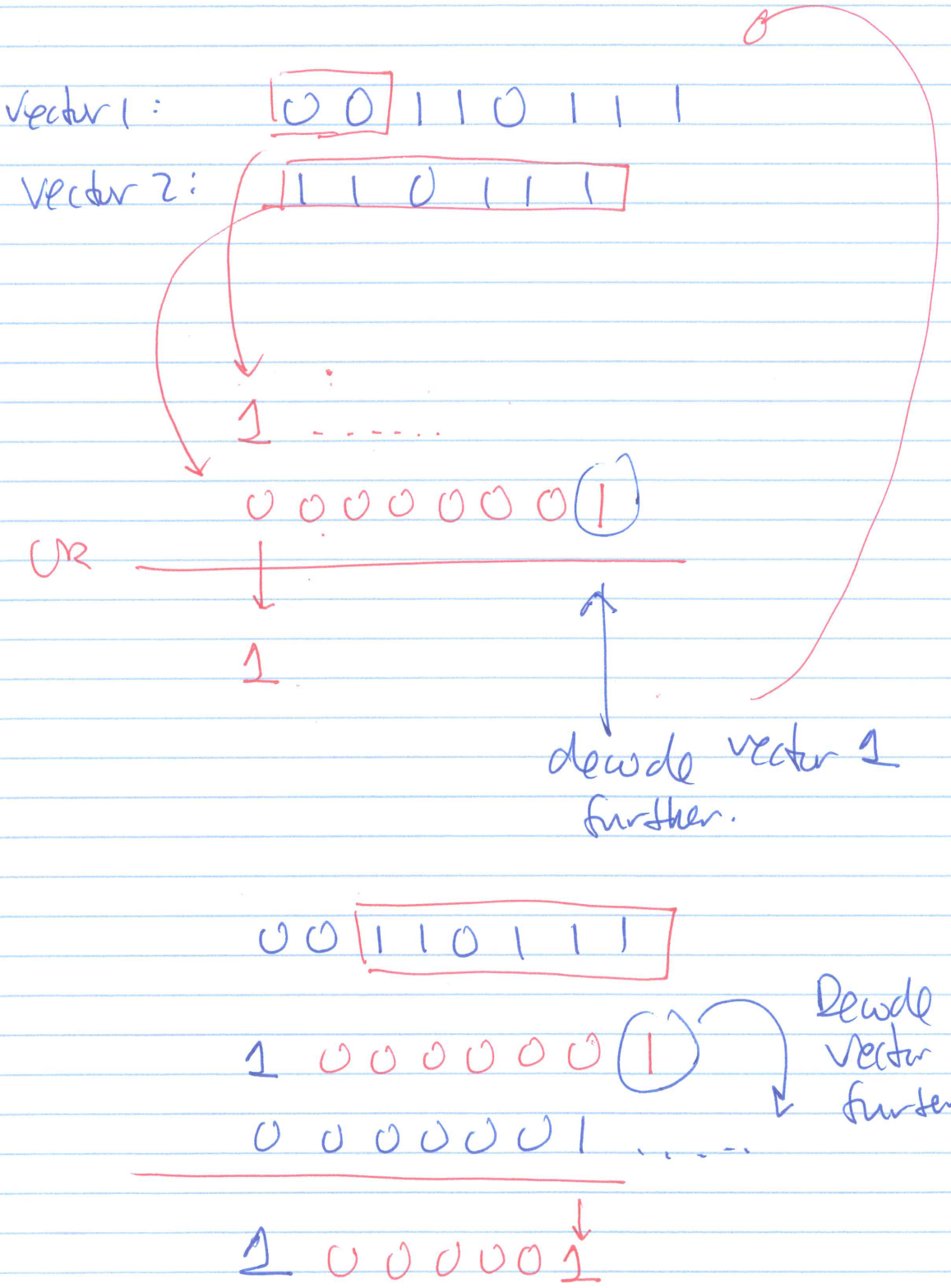


0 0 0 0 0 0 0 0 1 0 0 0 0.

Then perform (OR) operation:

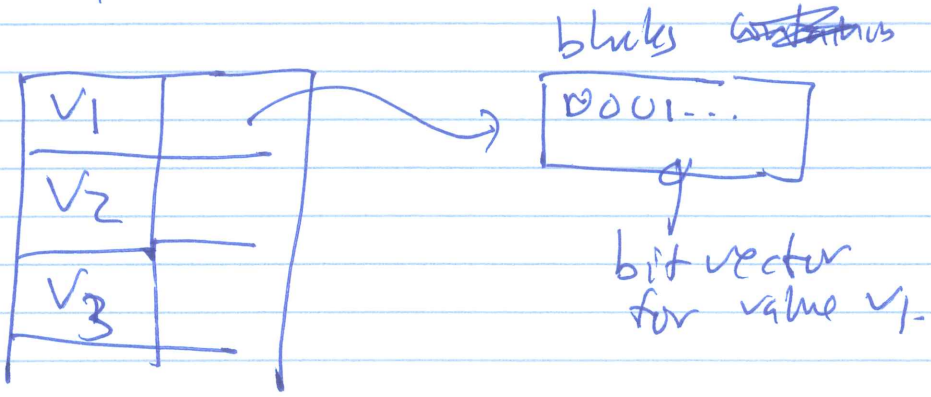


Just-in Time decoding:



Managing Bitmap Indexes.

- Storing Bitmap Indexes:
  - Use an index file on the values in the field.
  - schematically:

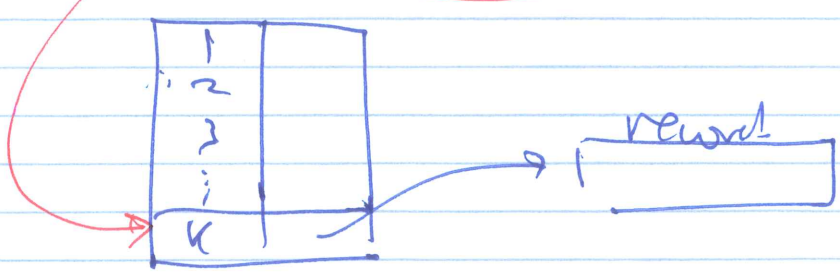


- Retrieving records using bit map index.

bit vector = ... 1 ...

position k.

- Use secondary index to index the records on their position.



- Maintaining bit vectors when records are: inserted and deleted.

Problem: record numbers must remain FIXED once it is assigned to a record!

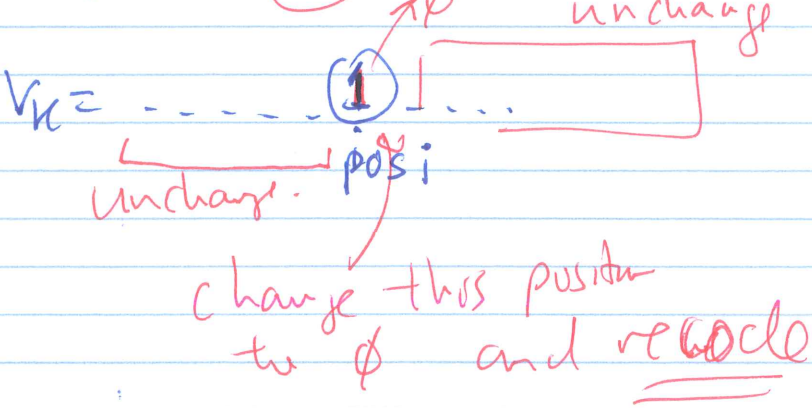
- Delete a record i

record i : 

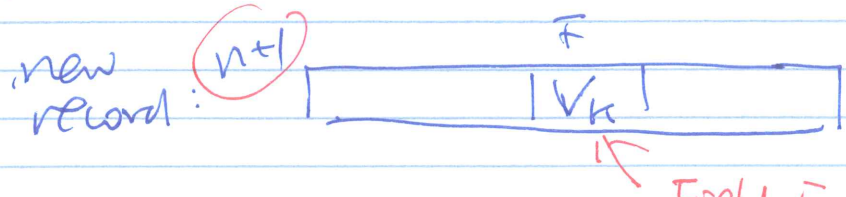
Field F has value V_K.

(1) Replace record i with a tombstone (retire the record number!)

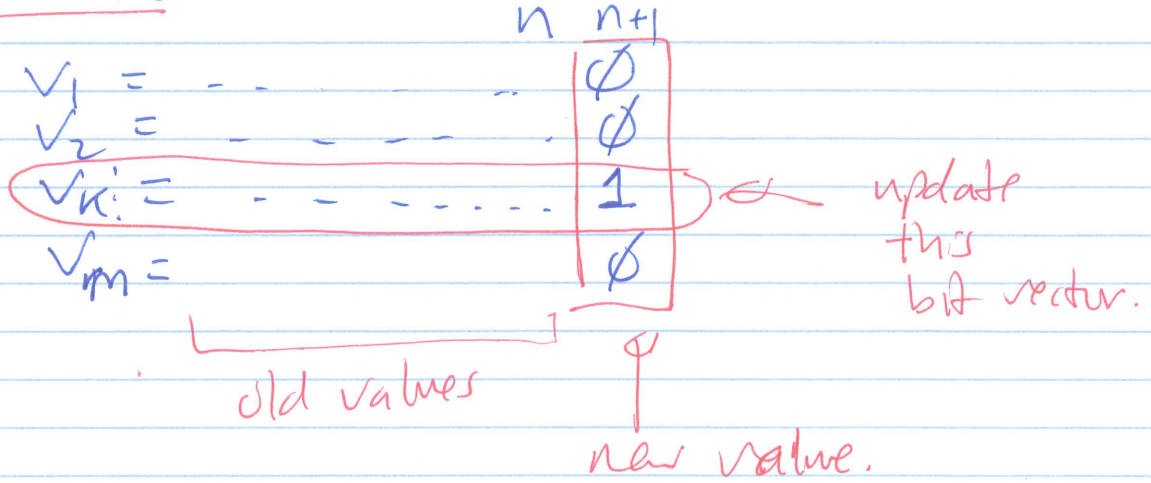
(2) Bit vector for V_K:



- Insert a new record



Old values:



- We only need to re-code the Run-length code for

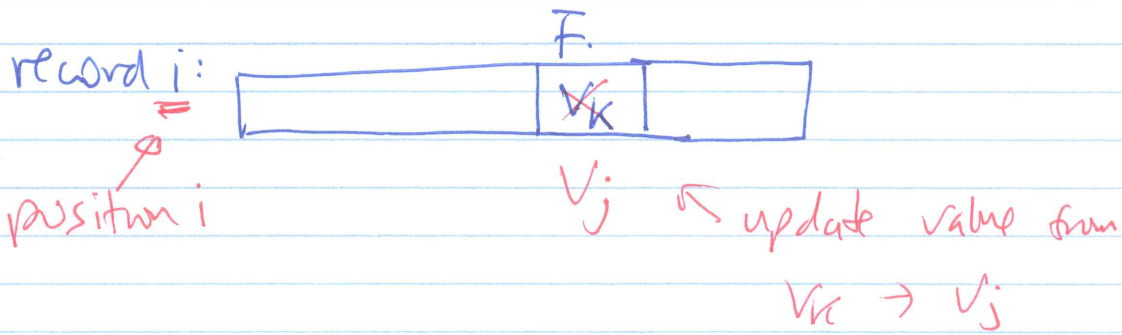
$$V_k = \ldots\ldots 1$$

(The other run-length codes remain unchanged!!!).

- If V_k is a new value, add:

$$V_k = 000 \ldots 01.$$

Update a value:



(1) Update 2 bit vectors:

$V_K =$ ..... 1 ..... → 0

pos i

$V_j =$ ..... 0 ..... → 1

pos i