>>

Signature-based Indexing

∧ >>

❖ Indexing with Signatures

Signature-based indexing:

- designed for *pmr* queries (conjunction of equalities)
- does not try to achieve better than *O(n)* performance
- attempts to provide an "efficient" linear scan

Each tuple is associated with a signature

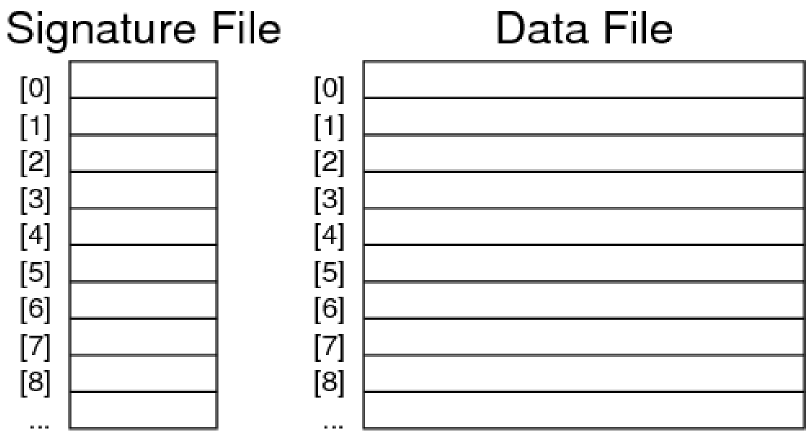
- a compact (lossy) descriptor for the tuple
- formed by combining information from multiple attributes
- stored in a signature file, parallel to data file

Instead of scanning/testing tuples, do pre-filtering via signatures.

<< ^ >>

❖ Indexing with Signatures (cont)

File organisation for signature indexing (two files)



One signature slot per tuple slot; unused signature slots are zeroed.

Signatures do not determine record placement ⇒ can use with other indexing.

<< ^ >>

❖ Signatures

A signature "summarises" the data from one tuple

A tuple consists of n attribute values $A_1 .. A_n$

A codeword $cw(A_i)$ is

- a bit-string, m bits long, where k bits are set to 1 ($k \ll m$)
- derived from the value of a single attribute A_i

A tuple descriptor (signature) is built

- by combining $cw(A_i)$, $i=1..n$
- aim to have roughly half of the bits set to 1

Two strategies for building signatures: overlay, concatenate

<< ^ >>

❖ Generating Codewords

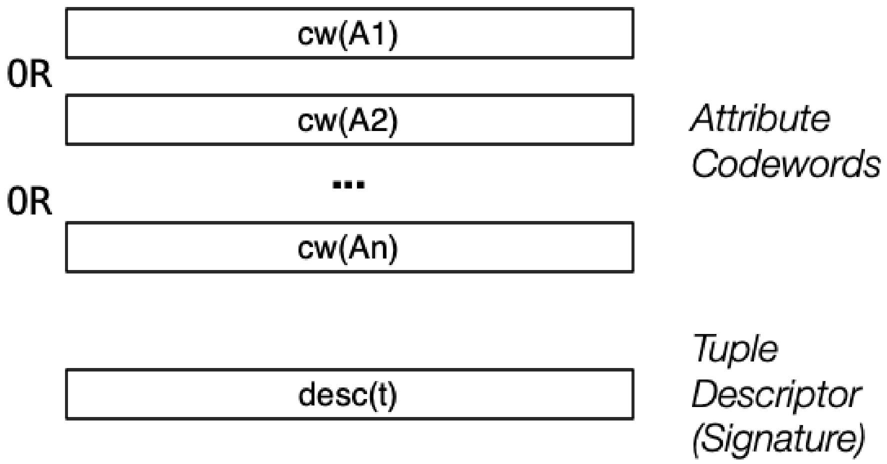
Generating a *k-in-m* codeword for attribute A_i

```
bits codeword(char *attr_value, int m, int k)
{
   int  nbits = 0;    // count of set bits
   bits cword = 0;    // assuming m <= 32 bits
   srandom(hash(attr_value));
   while (nbits < k) {
      int i = random() % m;
      if (((1 << i) & cword) == 0) {
         cword |= (1 << i);
         nbits++;
      }
   }
   return cword;   // m-bits with k 1-bits and m-k 0-bits
}
```

❖ Superimposed Codewords (SIMC)

In a superimposed codewords (simc) indexing scheme

- tuple descriptor formed by *overlaying* attribute codewords (bitwise-OR)



<< ^ >>

❖ Superimposed Codewords (SIMC) (cont)

A SIMC tuple descriptor *desc(t)* is

- a bit-string, m bits long, where $j \le nk$ bits are set to 1
- $desc(t) = cw(A_1)$ **OR** $cw(A_2)$ **OR** ... **OR** $cw(A_n)$

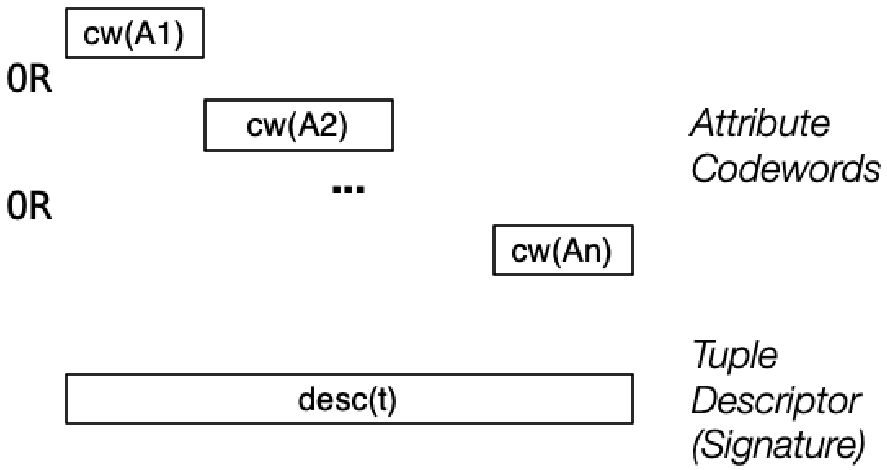
Method (assuming all n attributes are used in descriptor):

```
Bits desc = 0
for (i = 1; i <= n; i++) {
   bits cw = codeword(A[i],m,k)
   desc = desc | cw
}
```

❖ Concatenated Codewords (CATC)

In a concatenated codewords (catc) indexing schema

- tuple descriptor formed by *concatenating* attribute codewords



<< ∧ >>

❖ Concatenated Codewords (CATC) (cont)

A CATC tuple descriptor *desc(t)* is

- a bit-string, *m* bits long, where *j = nk* bits are set to 1
- $desc(t) = cw(A_1) + cw(A_2) + ... + cw(A_n)$ (+ is concatenation)

Each codeword is *p = m/n* bits long, with *k = p/2* bits set to 1

Method (assuming all *n* attributes are used in descriptor):

```
Bits desc = 0 ;  int p = m/n
for (i = 1; i <= n; i++) {
   bits cw = codeword(A[i],p,k)
   desc = desc | (cw << p*(n-i))
}
```

❖ Queries using Signatures

To answer query q with a signature-based index

- first generate a query descriptor *desc(q)*
- then scan the signature file using the query descriptor
- if sig_i matches *desc(q)*, then tuple i may be a match

desc(q) is formed from codewords of known attributes.

Effectively, any unknown attribute A_i has $cw(A_i) = 0$

E.g. for SIMC **`(a,?,c,?,e)`** = *cw*(**`a`**) **`OR`** 0 **`OR`** *cw*(**`c`**) **`OR`** 0 **`OR`** *cw*(**`e`**)

❖ Queries using Signatures (cont)

Once we have a query descriptor, we search the signature file:

```
pagesToCheck = {}
// scan r descriptors
for each descriptor D[i] in signature file {
    if (matches(D[i],desc(q))) {
        pid = pageOf(tupleID(i))
        pagesToCheck = pagesToCheck ∪ pid
    }
}
// scan bq + δ data pages
for each pid in pagesToCheck {
    Buf = getPage(dataFile,pid)
    check tuples in Buf for answers
}
```

<< ^ >>

❖ False Matches

Both SIMC and CATC can produce false matches

- `matches(D[i],desc(q))` is true, but `Tup[i]` is not a solution for `q`

Why does this happen?

- signatures are based on hashing, and it is possible that

  `hash(key1) == hash(key2)` even though `key1 != key2`

- for SIMC, overlaying could also produce "unfortunate" bit-combinations

To mitigate this, need to choose "good" m and k

❖ SIMC vs CATC

Both build *m*-bit wide signatures, with ~1/2 bits set to 1

Both have codewords with ~*m/2n* bits set to 1

CATC: codewords are *m/n* = *p*-bits wide

- shorter codewords → more hash collisions

SIMC: codewords are also *m*-bits wide

- longer codewords ⇒ less hash collisions, but also has overlay collisions

CATC has option of different length codeword p_i for each A_i ($\sum p_i = m$)

Produced: 20 Mar 2021