>>

Multi-dimensional Hashing

- Hashing and *pmr*
- MA.Hashing Example
- MA.Hashing Hash Functions
- Queries with MA.Hashing
- MA.Hashing Query Algorithm
- Query Cost for MA.Hashing
- Optimising MA.Hashing Cost

∧ >>

❖ Hashing and *pmr*

For a *pmr* query like

```
select * from R where a1 = C1 and ... and an = Cn
```

- if one a_i is the hash key, query is very efficient
- if no a_i is the hash key, need to use linear scan

Can be alleviated using *multi-attribute hashing* (*mah*)

- form a composite hash value involving all attributes
- at query time, some components of composite hash are known
  (allows us to limit the number of data pages which need to be checked)

MA.hashing works in conjunction with any dynamic hashing scheme.

❖ Hashing and *pmr* (cont)

Multi-attribute hashing parameters:

- file size = $b = 2^d$ pages ⇒ use d-bit hash values
- relation has n attributes: $a_1, a_2, \ldots a_n$
- attribute a_i has hash function h_i
- attribute a_i contributes d_i bits (to the combined hash value)
- total bits $d = \Sigma_{i=1..n}\, d_i$
- a choice vector (cv) specifies for all $k \in 0..d-1$
  bit j from $h_i(a_i)$ contributes bit k in combined hash value

7	6	5	4	3	2	1	0
$b_{2,2}$	$b_{1,2}$	$b_{3,1}$	$b_{2,1}$	$b_{1,1}$	$b_{3,0}$	$b_{2,0}$	$b_{1,0}$

❖ MA.Hashing Example

Consider relation **`Deposit(branch,acctNo,name,amount)`**

Assume a small data file with 8 main data pages (plus overflows).

Hash parameters: $d=3$ $d_1=1$ $d_2=1$ $d_3=1$ $d_4=0$

Note that we ignore the **`amount`** attribute ($d_4=0$)

Assumes that nobody will want to ask queries like

```
select * from Deposit where amount=533
```

Choice vector is designed taking expected queries into account.

<< ^ >>

❖ MA.Hashing Example (cont)

Choice vector:

7	6	5	4	3	2	1	0
$b_{2,2}$	$b_{1,2}$	$b_{3,1}$	$b_{2,1}$	$b_{1,1}$	$b_{3,0}$	$b_{2,0}$	$b_{1,0}$

This choice vector tells us:

- bit 0 in hash comes from bit 0 of $hash_1(a_1)$ ($b_{1,0}$)
- bit 1 in hash comes from bit 0 of $hash_2(a_2)$ ($b_{2,0}$)
- bit 2 in hash comes from bit 0 of $hash_3(a_3)$ ($b_{3,0}$)
- bit 3 in hash comes from bit 1 of $hash_1(a_1)$ ($b_{1,1}$)
- etc. etc. etc. (up to as many bits of hashing as required, e.g. 32)

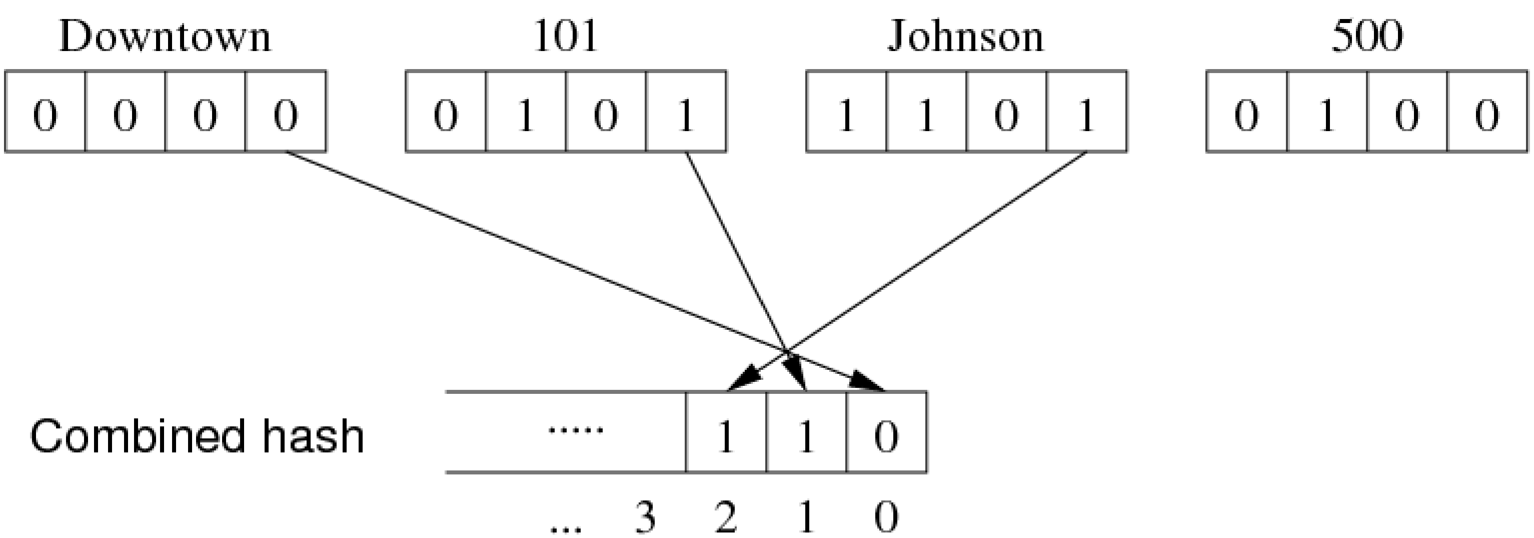
<< ^ >>

❖ MA.Hashing Example (cont)

Consider the tuple:

branch	acctNo	name	amount
Downtown	101	Johnston	512

Hash value (page address) is computed by:



<< ^ >>

❖ MA.Hashing Hash Functions

Auxiliary definitions:

```
#define MaxHashSize 32
typedef unsigned int HashVal;

// extracts i'th bit from hash value
#define bit(i,h) (((h) & (1 << (i))) >> (i))

// choice vector elems
typedef struct { int attr, int bit } CVelem;
typedef CVelem ChoiceVec[MaxHashSize];

// hash function for individual attributes
HashVal hash_any(char *val) { ... }
```

❖ MA.Hashing Hash Functions (cont)

Produce combined *d*-bit hash value for tuple *t* :

```
HashVal hash(Tuple t, ChoiceVec cv, int d)
{
    HashVal h[nAttr(t)+1];  // hash for each attr
    HashVal res = 0, oneBit;
    int     i, a, b;
    for (i = 1; i <= nAttr(t); i++)
        h[i] = hash_any(attrVal(t,i));
    for (i = 0; i < d; i++) {
        a = cv[i].attr;
        b = cv[i].bit;
        oneBit = bit(b, h[a]);
        res = res | (oneBit << i);
    }
    return res;
}
```

<< ^ >>

❖ Queries with MA.Hashing

In a partial match query:

- values of some attributes are known
- values of other attributes are unknown

E.g.

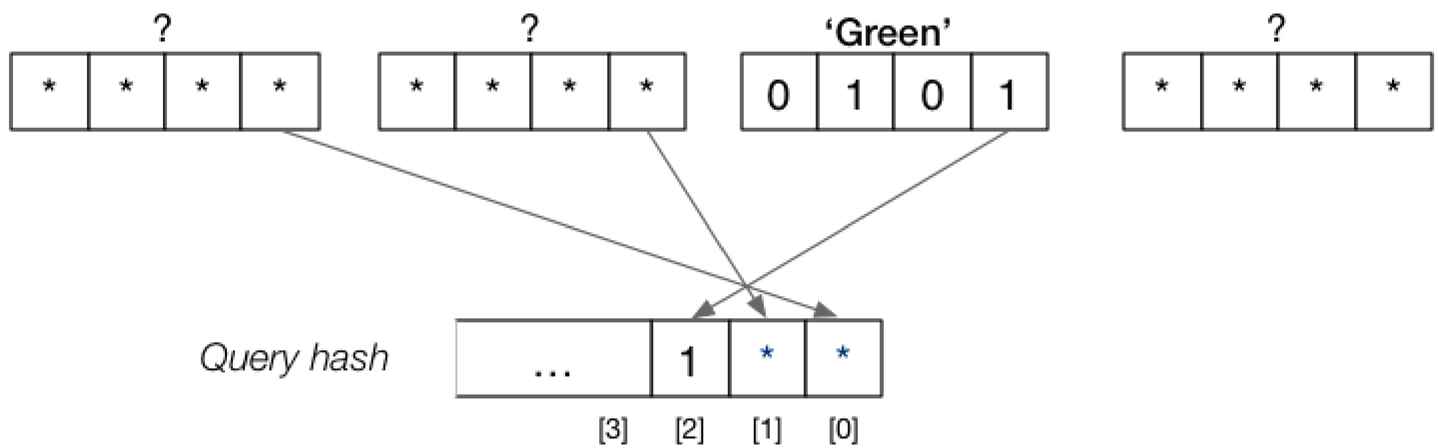
```
select amount
from   Deposit
where  branch = 'Brighton' and name = 'Green'
```

for which we use the shorthand **`(Brighton, ?, Green, ?)`**

<< ∧ >>

❖ Queries with MA.Hashing (cont)

Consider query: select amount from Deposit where name='Green'



Matching tuples must be in pages: **100**, **101**, **110**, **111**.

<< ∧ >>

❖ MA.Hashing Query Algorithm

```
// Builds the partial hash value (e.g. 10*0*1)
// Treats query like tuple with some attr values missing

ns = 0;   // # unknown bits = # stars
for each attribute i in query Q {
    if (hasValue(Q,i)) {
        set d[i] bits in composite hash
            using choice vector and hash(Q,i)
    } else {
        set d[i] *'s in composite hash
            using choice vector
        ns += d[i]
    }
}
...
```

<< ^ >>

❖ MA.Hashing Query Algorithm (cont)

```
...
// Use the partial hash to find candidate pages

r = openRelation("R",READ);
for (i = 0; i < 2^ns; i++) {
    P = composite hash
    replace *'s in P
        using i and choice vector
    Buf = getPage(fileOf(r), P);
    for each tuple T in Buf {
        if (T satisfies pmr query)
            add T to results
    }
}
```

<< ∧ >>

❖ Query Cost for MA.Hashing

Multi-attribute hashing handles a range of query types, e.g.

```
select * from R where a=1
select * from R where d=2
select * from R where b=3 and c=4
select * from R where a=5 and b=6 and c=7
```

A relation with n attributes has 2^n different query types.

Different query types have different costs (different no. of *'s)

$Cost(Q) = 2^s$ where $s = \Sigma_{i \notin Q}\, d_i$ (alternatively $Cost(Q) = \Pi_{i \notin Q}\, 2^{d_i}$)

Query distribution gives probability p_Q of asking each query type Q.

❖ Query Cost for MA.Hashing (cont)

Min query cost occurs when all attributes are used in query

$Min\ Cost_{pmr} = 1$

Max query cost occurs when no attributes are specified

$Max\ Cost_{pmr} = 2^d = b$

Average cost is given by weighted sum over all query types:

$Avg\ Cost_{pmr} = \Sigma_Q\, p_Q\, \Pi_{i \notin Q}\, 2^{d_i}$

Aim to minimise the weighted average query cost over possible query types

❖ Optimising MA.Hashing Cost

For a given application, useful to minimise $Cost_{pmr}$.

Can be achieved by choosing appropriate values for d_i (*cv*)

Heuristics:

- distribution of query types (more bits to frequently used attributes)
- size of attribute domain (≤#bits to represent all values in domain)
- discriminatory power (more bits to highly discriminating attributes)

Trade-off: making Q_j more efficient makes Q_k less efficient.

This is a combinatorial optimisation problem
(solve via standard optimisation techniques e.g. simulated annealing)

Produced: 15 Mar 2021