course 6 Selection on Multiple Attributes

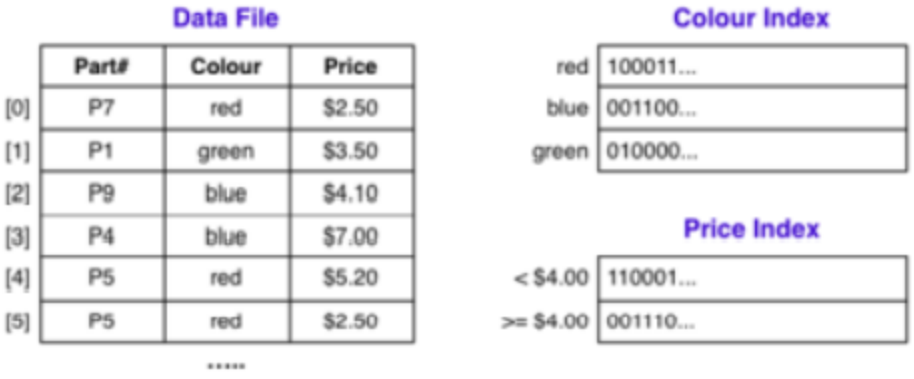
1. Muti-dimensional(Nd) Selection

1.1 operations for Nd select

- pmr = patial-match retrieval
- space = tuple-space queries

1.2 Bitmap Indexes

bitmap indexes is similar to storage management in OS (which also use bitmap to indicate whether an block is used or not), and a bitmap index assist computation of result sets in pmr.



2. N-dimensional Hashing

2.1 Hashing and pmr

we know that hahsing only effective if hash key was used in query, so we devise multi-attribute hashing that combinning hashes from attributes (by using bits-methods)

multi-attribute hashing do a work that :

- many attributes → hashing → hash_val linked to page address

2.2 Muti-attribute (MA) example

if there are b tuples in our table, we can use d bit to represent each tuple(b = 2^d)

and for each attributes in table, if will be used in query as condition, will have a value generated by h function, which is shown below.

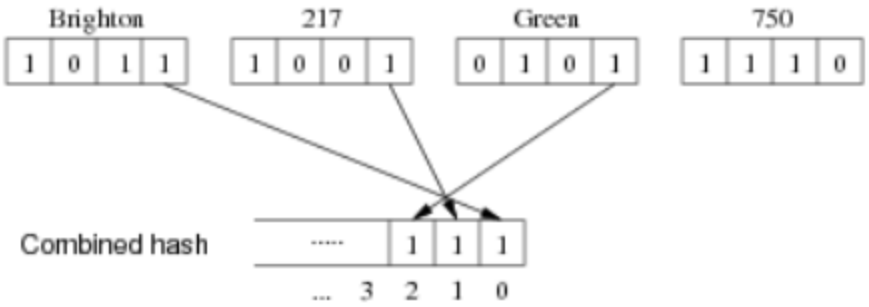
branch	h(B)	acctNo	h(Ac)	name	h(N)	amount

Brighton	1011	217	1001	Green	0101	750
Downtown	0000	101	0101	Johnson	1101	512
Mianus	1101	215	1011	Smith	0001	700
Perryridge	0100	102	0110	Hayes	0010	400
Redwood	0110	222	1110	Lindsay	1000	695
Round Hill	1111	305	0001	Turner	0110	350
Clearview	1110	117	0101	Throggs	0110	295

and for each tuples, the procedure fo combined hash is shown below:

branch	acctNo	name	amount
Brighton	217	Green	750

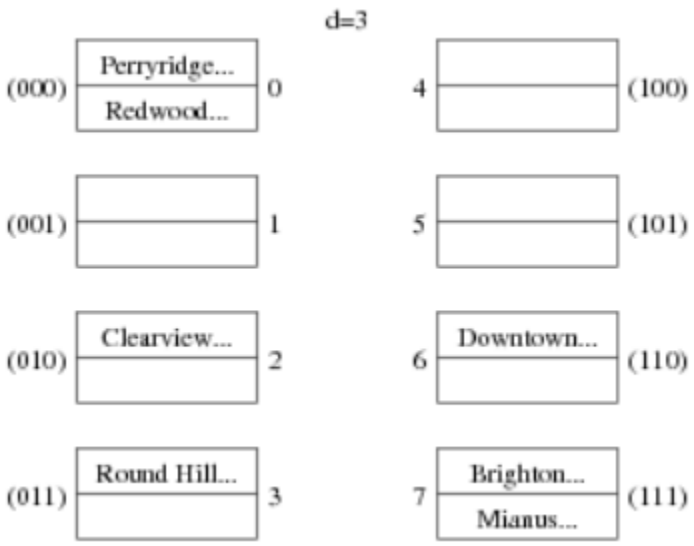
Hash value (page address) is computed by:



and finally we will get a table like this:

tuple	Hash
(Brighton,217,Green,750)	111
(Downtown,101,Johnshon,512)	110
(Mianus,215,Smith,700)	111
(Perryridge,102,Hayes,400)	000
(Redwood,222,Lindsay,695)	000
(Round Hill,305,Turner,350)	011
(Clearview,117,Throggs,295)	010

and the layout of table in page is shown below:



2.3 queries with MA.Hashing

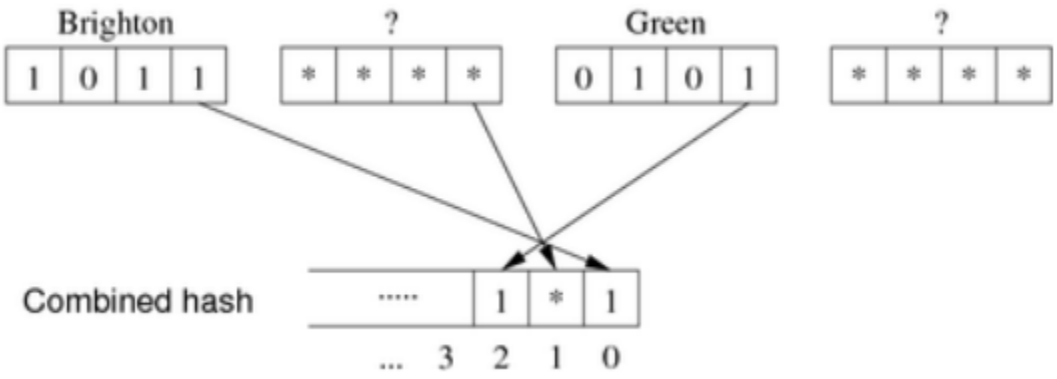
there are several types of queries

Query type	Cost	p_Q
(?, ?, ?, ?)	8	0
(*br*, ?, ?, ?)	4	0.25
(?, *ac*, ?, ?)	4	0
(*br*, *ac*, ?, ?)	2	0
(?, ?, *nm*, ?)	4	0
(*br*, ?, *nm*, ?)	2	0
(?, *ac*, *nm*, ?)	2	0.25
(*br*, *ac*, *nm*, ?)	1	0
(?, ?, ?, *amt*)	8	0
(*br*, ?, ?, *amt*)	4	0
(?, *ac*, ?, *amt*)	4	0
(*br*, *ac*, ?, *amt*)	2	0
(?, ?, *nm*, *amt*)	4	0
(*br*, ?, *nm*, *amt*)	2	0.5
(?, *ac*, *nm*, *amt*)	2	0
(*br*, *ac*, *nm*, *amt*)	1	0

from example:

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

if we try to form a composite hash for a query, we don’t know **values** for some bits:



for this query, we can only obtain that the hash code is 1*1, which contian the combinations of 101,111.

and for others types of query, we can extract a pattern that number of combinations = 2^(num of * bits).

2.4 optimising Ma.Hashing Cost

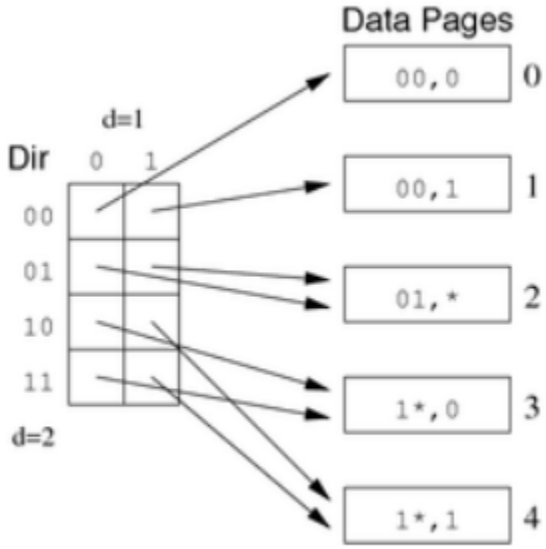
and for retireval, the more num of vague bit that hash string contain, the hard the retrieval conduct

so we should distribute more bit to those frequently used attributes, we can use heuristics methods to solve this problem with the distribution of query.

2.5 Grid File

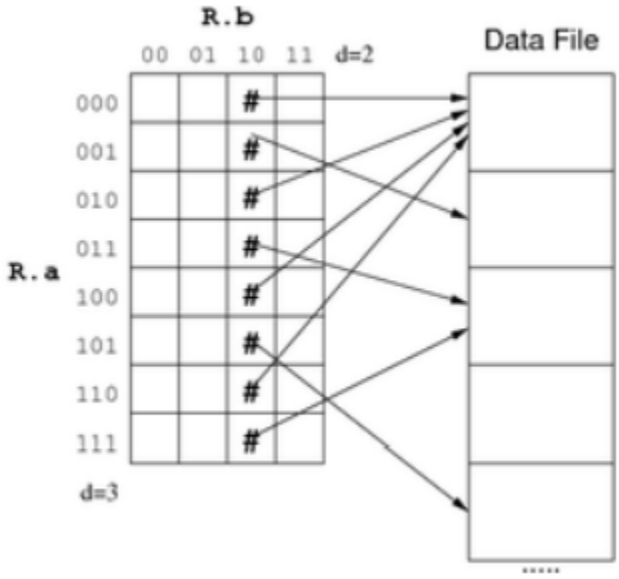
the MA.Hashing aims to combine all attributes in a hash values, but an alternative approach is to keep hash values seperate by using a special structure **grip files.**

- for one attribute hash, we use Ext.hash, which consist of one diretory
- and for multi-attribute hash, we use grip files, which consist of k-dimension to handle k attributes



A simple 2–dimensional grid file

and for selection with grid files, there was an example:



3. N-dimensional Tree indexes

we have seen that hashing can be extended in multiple attributes, so as index?

yes, and there are several tree-based indexs methods to handle this problem:

- kd-trees
- Quad-trees
- R-trees

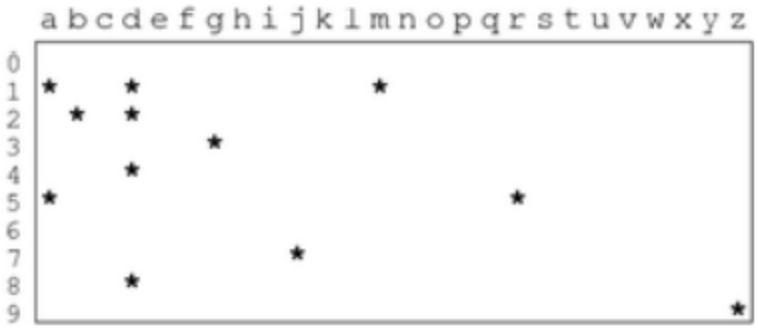
and we will illustrate those data-structure above with a 2-d relation:

```
create table Rel (
    A1 char(1),  # 'a'..'z'
    A2 integer   # 0..9
);
```

with example tuples:

```
Rel('a',1)  Rel('a',5)  Rel('b',2)
Rel('d',1)  Rel('d',2)  Rel('d',4)
Rel('d',8)  Rel('g',3)  Rel('j',7)
Rel('m',1)  Rel('r',5)  Rel('z',9)
```

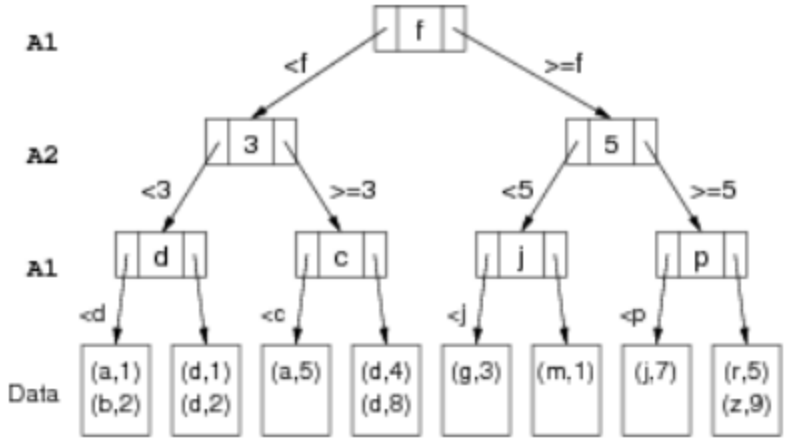
and the tuple space of the above tuples is shown below:

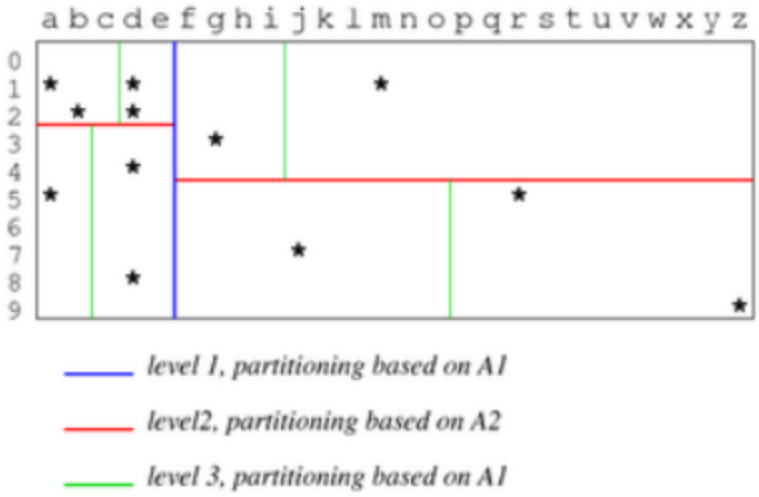


3.1 kd-Trees

a kd-trees are a multi-way search tree where:

- each level of the tree use different attributes to partition tuples
- each node contains n-1 value, point to n subtrees





3.1.1 search in k-d trees

k-d trees can reduce the search space, just like the tuples space shown above, can the algorithm like that:

- for pmr queries, like ```select * from student where age=18```

```
Search(Query Q, Level L, Node N)
{
    if (isLeaf(N)) {
        Buf = getPage(f,N.page)
        check Buf for matching tuples
    } else {
        ai = attrLev[L]
        if (hasValue(Q,ai)) {
            Val = getAttr(Q,ai)
            newN = search N for Val
            Search(Q, L+1, newN)
        } else {
            for each child newN of N
                Search(Q, L+1, newN)
}   }   }
```

- for range queries, like ```select * from student where age > 18 and age < 30```

```
if (hasValue(Q,ai)) {
    Val = getAttr(Q,ai)
    newN = search N for Val
    Search(Q, L+1, newN)
}
```

becomes

```
if (isRange(Q,ai)) {
    for each node N on current level {
        for each value V in range(Q,ai)) {
            newN = childOf(N,V)
            Search(Q, L+1, newN)
}   }   }
```

Query examples:

Open query: `(?,?,?,?,?,...)` (zero attributes specified)

- traverses entire tree (LNR,depth–first) and fetches every data page

Point query: `(a,b,c,d,e,...)` (all attributes specified)

- traverses a single path through the tree and fetches a single data page

Generic *pmr* query: `(a,?,c,?,e,...)`

- one path from top–level node, then all paths on second–level, ...

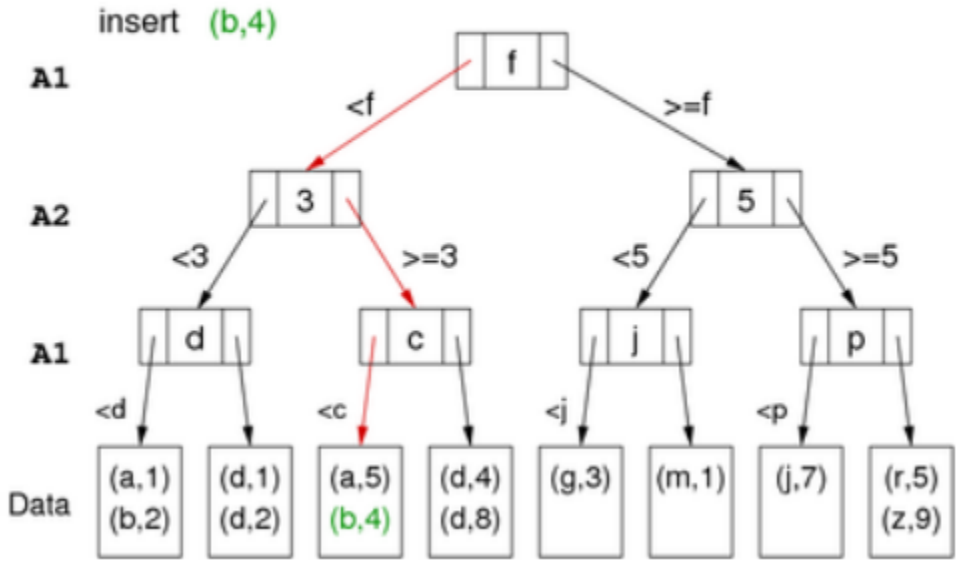
3.1.2 insertion into kd-trees

```
traverse tree to leaf node
    (reaching data page P1)
if (not full P1)
    insert tuple
else {
    create new data page P2
    compute new partition point Part
    distribute tuples between P1 and P2
        (using Part)
    create new internal node N with Part
```

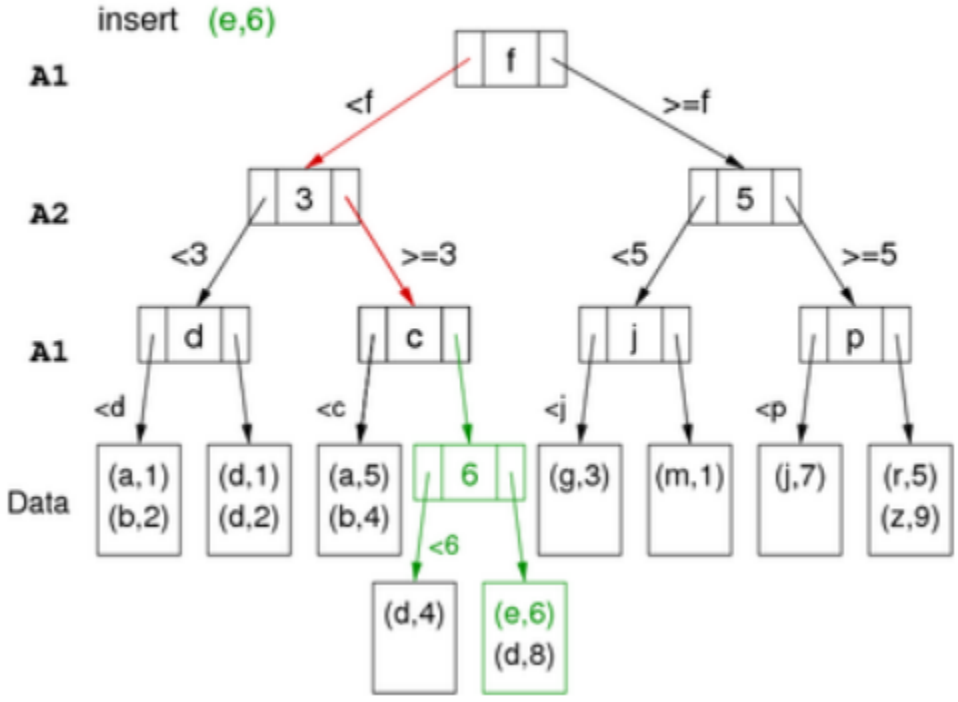


```
    link N into tree, link N to P1 and P2
}
write any modified pages (data or index)
```

Example insertion into non–full data page:

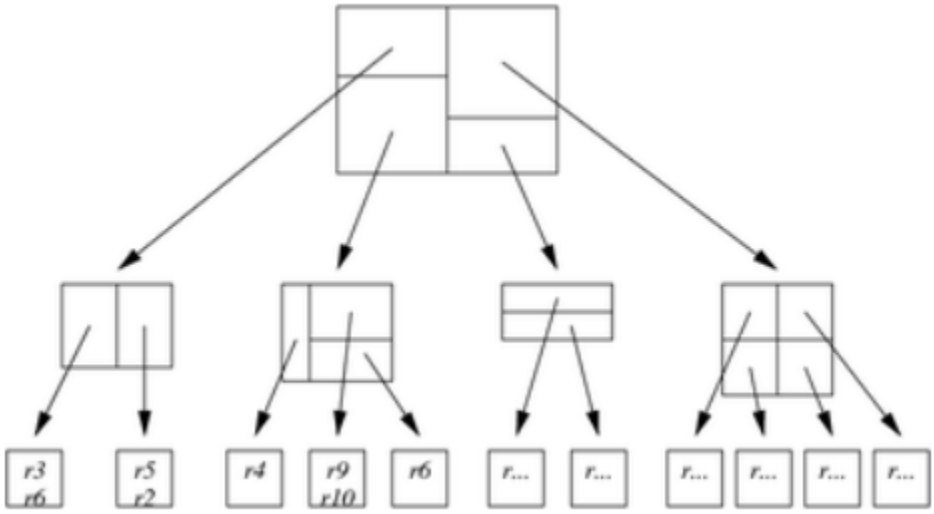


Example insertion into full data page:



3.2 kd-b trees

kd-b trees is a disk-based tree structure that use kd tree methods, it distribute kd-tree structure over a number of pages

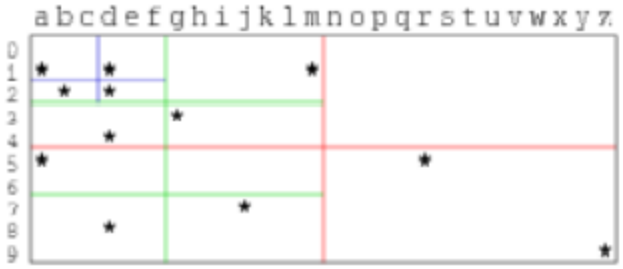


and for pmr queries:

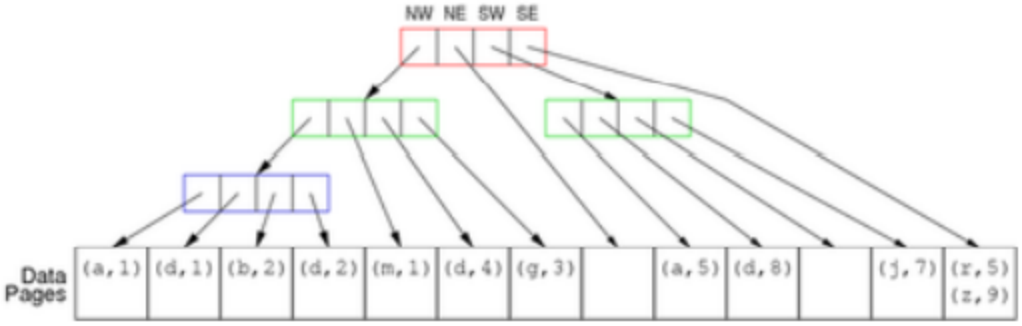
```
Pages = kdBsearch(query,root)
for each page P in Pages {
    Buf = getPage(f,P)
    scan Buf for matching tuples
}

kdbSearch(Query, Page)
{
    if (Page is a data page)
        Pages = Page
    else {
        Buf = getPage(kdbf,Page)
        search kd-tree in Buf
            using attributes from Query
        Pages = []
        for each indicated external subtree S {
            P = kdbSearch(Query, S)
            add P to Pages
    }   }
    return Pages
}
```

3.3 Quad trees

quad trees use a regular, recurisive partitioning of kd tuple space, it always divide a space into four pieces:



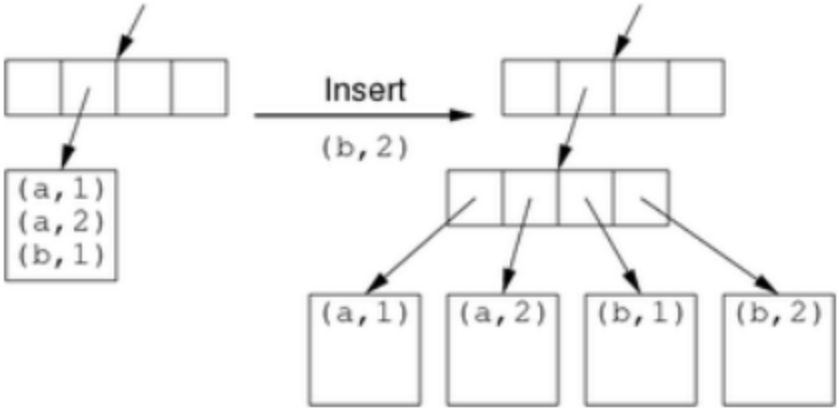
the quad trees structure is shown below:



3.3.1 insertion in quad trees

```
traverse tree to leaf node
    (reaching data page P1)
if (not full P1)
    insert tuple
else {
    create new data pages P2, P3, P4
    distribute tuples between P1 .. P4
    create new internal node N
    link N into tree, link N to P1 .. P4
}
write any modified pages (data or index)
```

Example:



Cost = traversal cost + update cost

= read $\geqslant 1$ node pages + write $\geqslant 1$ data pages

3.3.2 queries with quad trees

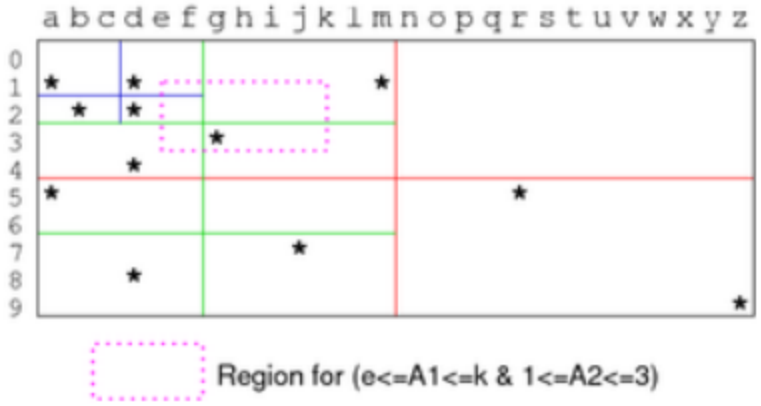
for pmr:

- if zero attributes provided, quad trees provides no assistance
- if one attributes provided, from each node quad tree need to traverse half of braches
- if two attributes provided, following single branch for each node

for space queries:

- need to traverse all branches which intersect this region

Space query example:

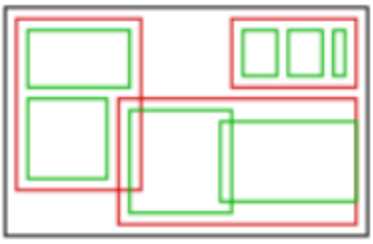


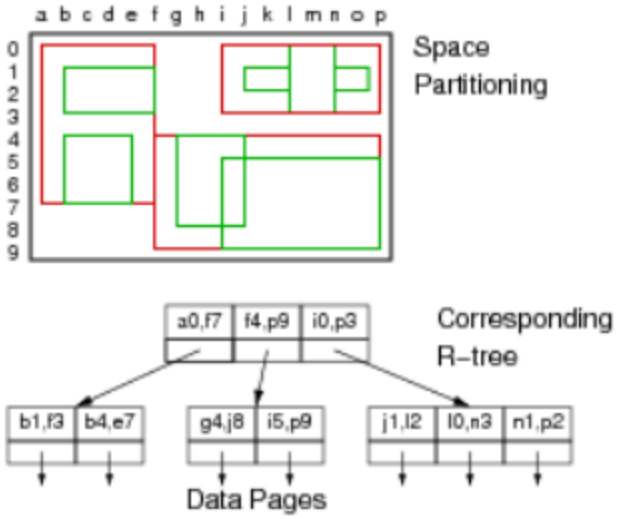
Need to traverse: red(NW), green(NW,NE,SW,SE), blue(NE,SE).

3.4 R-Trees

R-trees use a flexible, hierachical partitioning of kd tuples space, the distinct features fo R-trees is that the child regions do not need to cover parent region

Example 2d R−tree node (with two levels of children):





the goal of building a R-tree is:

- every node except for root contains between m and M entries
- the root node has at least two entries
- the tree is high-balanced

3.4.1 query with R-trees

R-trees is designed to handle “where am i” and space query

- for “where am i” work

"Where–am–I" query: find all regions containing a given point *P*:

- start at root, select all children whose subregions contain *P*
- if there are zero such regions, search finishes with *P* not found
- otherwise, recursively search within node for each subregion
- once we reach a leaf, we know that that region contains *P*

Algorithm for "where am I" query in 2d R-tree:

```
Regions = RtreeSearch(a,b,root)

RtreeSearch(x,y,Node) {
    Regions = []
    if (leaf(Node)) {
        for each (x1,y1,x2,y2,P) in Node {
            if (x1,y1,x2,y2) contains (a,b) {
                add (x1,y1,x2,y2) to Regions
    }   }   }
    else {
        for each (x1,y1,x2,y2,Child) in Node {
            if (x1,y1,x2,y2) contains (a,b) {
                Rs = RtreeSearch(x,y,Child)
                add Rs to Regions
    }   }   }
    return Regions;
}
```

- for space queries, we traverse down any path that interacts the query region

R-tree variants

R+ tree does not permit overlapping regions.

- objects that intersect *n* regions are stored *n* times
  (i.e. stored in each region that they intersect)

R* tree uses delayed splitting on insertion.

- the aim is to minimise region overlap, within original framework

SS-tree uses (hyper)spherical regions rather than (hyper)cubes

- the aim is to reduce amount of overlap among regions

4. Signature-based Selection

4.1 signature

a signature is a compact descriptor for a tuple

- it must summarise all fields in a tuple
- must be able to be checked efficiently in queries

only one requirement for signature that after a tuple has been placed in data file, a signature must be placed in the corresponding slot in the signature file

there are two types of signatures:

- superimposed codewords
- disjoint codewords

the adavantages of signature files:

- scanning the signature file and comparing decriptors is faster than scanning the tuple file and checking tuples

4.2 generating codewords

use the value of attribute as seed of random function:

```
seed = hash(A[i])
nbits = 0
codeword = 0  /* m zero bits */
while (nbits < k) {
     i = random(0,m-1)
     if (bit i not set in codeword) {
          set bit i in codeword
          nbits++
     }
}
```

4.3 superimposed codewords (SIMC)

Consider the following tuple (from a bank deposit database)

branch	acctNo	name	amount
Perryridge	102	Hayes	400

It has the following codewords/descriptor (for $m = 12$, $k = 2$)

A_i	$cw(A_i)$
Perryridge	010000000001
102	000000000011
Hayes	000001000100
400	000010000100
desc(r)	010011000111

Signature	Tuple
100101001001	(Brighton,217,Green,750)
010101010101	(Clearview,117,Throggs,295)

101001001001	(Downtown,101,Johnshon,512)
101100000011	(Mianus,215,Smith,700)
010011000111	(Perryridge,102,Hayes,400)
100101010011	(Redwood,222,Lindsay,695)
011110111010	(Round Hill,305,Turner,350)

4.3.1 SIMC queries

the procedure of a SIMC queries is:

- first generate a query descriptor desc(q)
- then use the query descriptor to search the signature files

E.g. consider the query `(Perryridge, ?, ?, ?)`.

A_i	$cw(A_i)$
Perryridge	010000000001
?	000000000000
?	000000000000
?	000000000000
desc(q)	010000000001

and then with desc(q)=010000000001, we scan the signature file and find several qualified tuples:

branch	acctNo	name	amount
Clearview	117	Throggs	295
Perryridge	102	Hayes	400

The first is an example of a *false match*.

False matches are caused by:

- codeword hashing collisions (two different values generate same codeword)
- ``unfortunate'' overlapping (bitwise OR from several attributes)

Example:

A_i	$cw(A_i)$
Perryridge	010000000001
102	000000000011
Hayes	000001000100
400	000010000100
desc(r)	010011000111
A_i	$cw(A_i)$
Clearview	010000010000
117	000100000001
Throggs	000001000100
295	000100000100
desc(r)	010101010101

How to reduce likelihood of false matches?

- hash each attribute using a different hash function (h_j for A_j)
- increase descriptor size (m)
- choose k so that ≈ half of bits are set (maximises different possible descriptors)

Increasing m helps, but at the expense of reading more descriptor data.

Having k too high ⇒ increased overlapping.

Having k too low ⇒ increased codeword collisions.

Thus, for SIMC schemes, there is the notion of choosing optimal m, k.

4.3.2 design with SIMC

so how to choose the best combination of (m, k) ?

there is a notion of false match probability **P_f**
so we can formula a optimization function is find the best combination of (m, k)

As suggested above, p_F is affected by m, k and n (#attributes)

In designing a SIMC index for a data file, the aim is to choose indexing parameters (m, k) to minimise the likelihood of false matches.

1. start by choosing acceptable p_F (e.g. $p_F \leqslant 10^{-5}$ i.e. one false match in 10,000)
2. then choose m and k to achieve no more than this p_F.

Formulae to derive m and k given p_F and n:

$$k = 1/\log_e 2 \,.\, \log_e (1/p_F)$$

$$m = (1/\log_e 2)^2 \,.\, n \,.\, \log_e (1/p_F)$$

Design example:

- a relation with 4 attributes ($n=4$)
- with acceptable false match probability $p_F=10^{-5}$
- optimal indexing parameters are $m=96$, $k=16$ (i.e. 12-byte descriptors)

4.3.3 query cost for SIMC

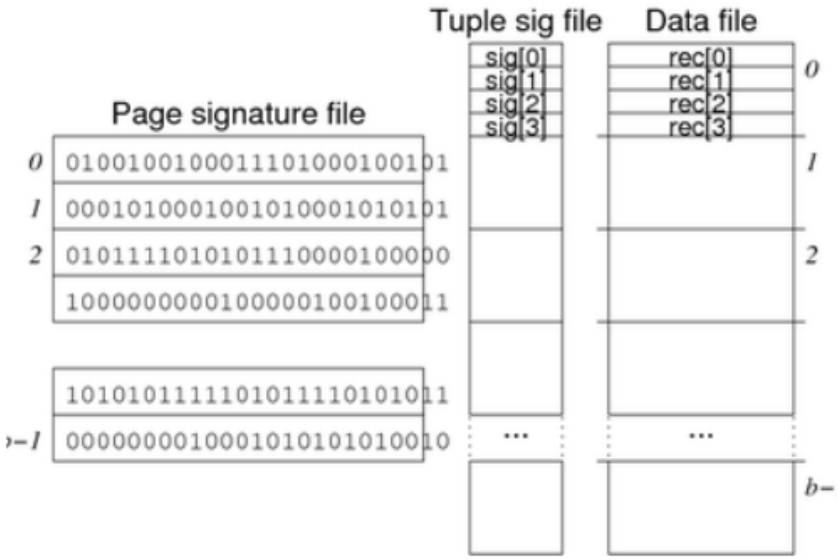
SIMC provide no assistance for range and space queries unless the data file is sorted

SIMC is of benefit for pmr queries

4.3.4 application of SIMC

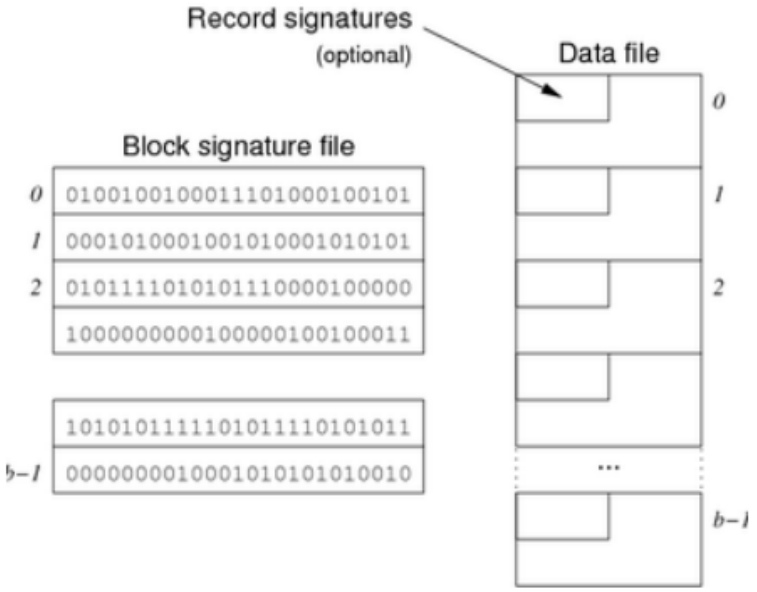
1. Two-level SIMC files
    a. because scanning one descriptor for every tuple is not efficient, so use page descriptor to replace tuple descriptor

File organisation for two-level superimposed codeword index



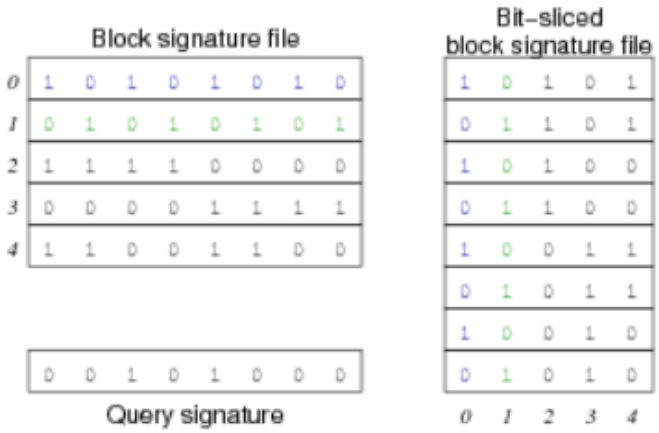
... Two-Level SIMC Files

Alternative file organisation:



2. Bit-sliced SIMC
    a. reading all page descriptors is still not efficient, especially when only a few bits are set

Rather than storing b m–bit page descriptors, store m b–bit descriptor slices.



4.4 Disjoint Codewords (DJC)

Consider the following tuple (from a bank deposit database)

branch	acctNo	name	amount
Perryridge	102	Hayes	400

It has the following codewords/descriptor (for $m = 4, \quad k = 2$)

A_i	$cw(A_i)$
Perryridge	0101
102	0011
Hayes	1001
400	0101
desc(t)	0101001110010101

and queris wirh Disjoint codewords is similar to superimposed codewords