Selection on Multiple Attributes

Multi-dimensional (Nd) Selection

Operations for Nd Select

N-dimensional select queries = condition on several attributes.

- *pmr* = partial-match retrieval, e.g.

  ```
  select * from Employees
  where  job = 'Manager' and gender = 'M';
  ```

- *space* = tuple-space queries, e.g.

  ```
  select * from Employees
  where  age > 55 and dept = 'Sales';
  ```

Tuple Space

One view of N-dimensional selection on a relation *R*...

- attribute domains of *R* specify a *D*-dimensional space
- each tuple $(v_1,v_2,...,v_D) \in R$ is a point in that space
- queries specify values/ranges on *N>1* dimensions
- queries identify a point/line/plane/region of the *D*-dim space
- results are tuples lying at/on/within that point/line/plane/region

E.g. if *N=D*, we are checking existence of a tuple at a point

Heaps

Heap files can handle *pmr* or *space* using standard method:

```
// select * from R where Cond
f = openFile(fileName("R"),READ);
for (p = 0; p < nPages(f); p++) {
    buf = getPage(f, p);
    for (i = 0; i < nTuples(buf); i++) {
        tup = getTuple(buf,i);
        if (matches(tup,Cond))
            add tup to result set
    }
}
```

$Cost_{pmr} = Cost_{space} = b$ (i.e. *O(n)*, worst case scenario)

Multiple Indexes

DBMSs already support building multiple indexes on a table.

Which indexes to build, depends on which queries are asked.

If we don't know queries, index all attribute subsets:

```
create table R (a int, b int, c int);
create index Rax on R (a);
create index Rbx on R (b);
create index Rcx on R (c);
create index Rabx on R (a,b);
create index Racx on R (a,c);
```

```
create index Rbcx on R (b,c);
create index Rallx on R (a,b,c);
```

... Multiple Indexes

Characteristics of indexes:

- if have index for all attributes in query ⇒ efficient
- if no query attributes are indexed ⇒ linear scan
- indexes cause space and update overhead proportional to #indexes

Since single-attribute indexes are common in DBMSs, consider first how to use these.

... Multiple Indexes

Generalised view of *pmr* and *space* queries:

```
select * from R
where  $a_1$ $op_1$ $C_1$ and $a_2$ $op_1$ $C_2$
       and ... and $a_n$ $op_n$ $C_n$
```

where, for *pmr*, all op_i are equality tests

Possible approaches to handling such queries ...

1. use index on one a_i to reduce tuple tests
2. use indexes on all a_i, and intersect answer sets

Selection using One Index

Using index on one attribute:

- requires an index for at least one attribute a_s used in query
- consult that index to find matches for a_s op_s C_s
- for each matching tuple, evaluate conjuncts a_i op_i C_i, for $i \neq s$

If several attributes have indexes:

- choose index with greatest *selectivity* (minimum # matches)
- but requires statistics on data distributions for all indexed a_i

---

... Selection using One Index

Abstract algorithm:

```
// select * from R where Cond
// Cond = $a_1op_1C_1$ and $a_2op_2C_2$ and $a_3op_3C_3$ ...
Tids = IndexLookup(R, a[i], op[i], C[i])
// gives { $tid_1$, $tid_2$, ...} for tuples satisfying $a_iop_iC_i$
Pages = { }
foreach tid in Tids {
    if (pageOf(tid) ∉ Pages)
        Pages = Pages ∪ {pageOf(tid)}
}
f = openFile(fileName("R"),READ)
foreach page in Pages {
    Buf = getPage(f,page);
    foreach tup in Buf {
        if (matches(tup, (a1 op1 C1 ...))
            add tup to Results
}   }
```

... Selection using One Index

Cost of this approach to selection:

- same as cost of using index on a_i to answer

  `select * from R where` a_i op_i C_i

- cost of index lookup (depends on index type)
- cost of reading "matching" pages (depends on query, i.e. b_q)

Note: some pages might contain

- some tuples that match on a_i op_i C_i
- but no tuples that satisfy `matches(tup,`*Cond*`)`

Selection using Many Indexes

Using indexes on several attributes:

- requires an index for *>1* attribute a_i used in query
- consult each index → M_i = set of matches on attribute a_i
- form intersection of M_i → set of answers

$M_i = \{ tid(t) \mid t \in R \wedge t[a_i]\ op_i\ C_i \}$

- *tid*s can be taken from index ⇒ no data access needed

... Selection using Many Indexes

Abstract algorithm:

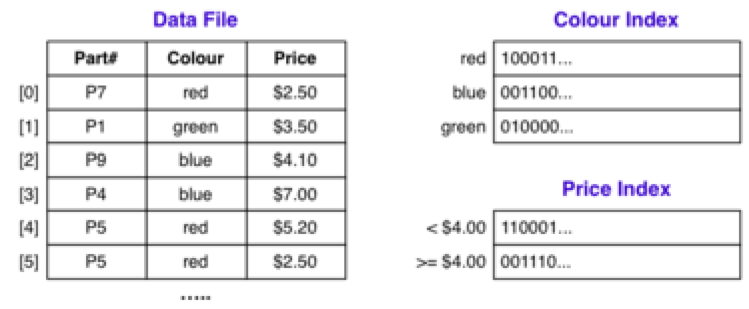
```
// select * from R where a1 op1 C1 and a2 op2 C2 ...
for (i = 1; i <= #QueryTerms; i++) {
    if (no index on a[i]) continue
    Tids = IndexLookup(R, a[i], op[i], C[i])
    PageSets[i] = { }
    foreach tid in Tids {
        if (pageOf(tid) ∉ PageSets[i])
            PageSets[i] = PageSets[i] ∪ pageOf(tid)
}   }
// PageSets[i] contains pages with tuples matching (ai opi Ci)
...
```

... Selection using Many Indexes

```
...
// compute intersection of Pages[i]
Pages = PageSets[1]
for (i = 2; i <= #QueryTerms; i++) {
    Pages = Pages ∩ PageSets[i];
}
// finally ... fetch data pages
f = openFile(fileName("R"),READ);
foreach page in Pages {
    Buf = getPage(f,page);
    // this page has at least 1 match
    foreach tup in Buf {
        if (matches(tup, (a1 op1 C1 ...))
            add tup to Results
}   }
```

Bitmap Indexes

A *bitmap index* assists computation of result sets in *pmr*.



Data File

	Part#	Colour	Price
[0]	P7	red	$2.50
[1]	P1	green	$3.50
[2]	P9	blue	$4.10
[3]	P4	blue	$7.00
[4]	P5	red	$5.20
[5]	P5	red	$2.50
		......	

Colour Index

red	100011...
blue	001100...
green	010000...

Price Index

< $4.00	110001...
>= $4.00	001110...

... Bitmap Indexes

Structure of bitmap index:

- one bitmap for each value/range for a given attribute
- bitmap has *r* bits, one bit for each tuple
- if tuple *i* has value *k* for attribute *a*
    - bitmap[a,v] has i^{th} bit set to 1 where $v=k$
    - bitmap[a,v] has i^{th} bit set to 0 where $v \neq k$
- assume we have a mapping from *i* to $TupleId_i$

... Bitmap Indexes

Consider using bitmap index to solve query:

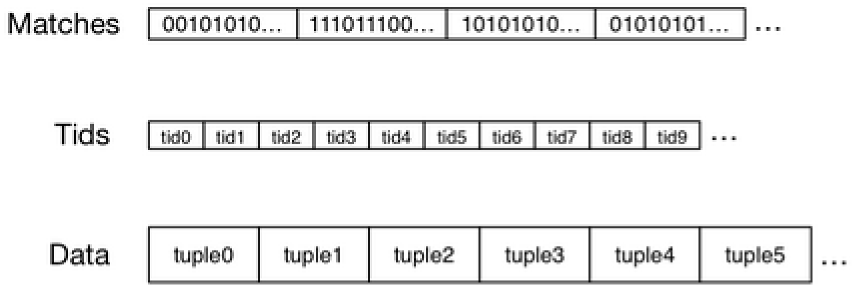
```
select * from Employees where dept='Sales' and gender='M'
```

Method:

```
tSales = bitmap[dept,'Sales']
tMales = bitmap[gender,'M']
matches = tSales & tMales
for (i in 0..r-1) {
   if (matches[i] == 1) {
      tid = slotToTupleId(i)
      fetch tuple t via TupleId tid
}  }
```

... Bitmap Indexes

Also useful to have a file of `tids`, giving file structures:



... Bitmap Indexes

Answering queries using bitmap index:

```
Matches = AllOnes(r)
foreach attribute A with index {
   // select ith bit-string for attribute A
   // based on value associated with A in WHERE
   Matches = Matches & Bitmaps[A][i]
}
// Matches contains 1-bit for each matching tuple
foreach i in 0..r-1 {
   if (Matches[i] == 0) continue;
   Pages = Pages U {pageOf(Tids[i])}
}
foreach pid in Pages {
   P = getPage(pid)
   extract matching tuples from P
}
```

... Bitmap Indexes

Costs associated with bitmap indexes ...

Storage costs:

- each bitmap requires $\lceil r/8 \rceil$ bytes
- one 8KB page can hold bitmap for 64K records
  (⇒ ≥ 1 bitmap fits in each page)
- one bitmap for each value/range for each indexed attribute

Query-time costs:

- read one bitmap for each indexed attribute in query (n_q)
- perform bitwise AND on bitmaps (in memory)
- read pages containing matching tuples (b_q)

N-dimensional Hashing

Hashing and *pmr*

Consider these *pmr* queries on a table hashed using `salary`

```
Q1: select * from Employees
    where  salary = 60000 and gender = 'M'

Q2: select * from Employees
    where  dept = 'Sales' and salary = 60K

Q3: select * from Employees
    where  salary > 60000 and dept = 'Sales'
```

$Cost_{Q1} = 1+Ov$, $Cost_{Q2} = b+b_{Ov}$, $Cost_{Q3} = b+b_{Ov}$

... Hashing and *pmr*

Problem: hashing only effective if hash key used in query.

No problem if all queries involve conditions like *key=val*

But frequently tables are accessed in multiple ways.

Can we devise hashing that "involves" attributes?

Yes, using *multi-attribute hashing* (*mah*) ...

- form composite hash by combining hashes from attributes
- a query involves some attributes ⇒ some hash bits known
- using known bits, can limit number of pages we need to read

... Hashing and *pmr*

Multi-attribute hashing ...

- combines hash values for multiple attributes
- to give a single hash value for each tuple
- uses hash to determine page in which to store tuple

Multi-attribute hashing "parameterises" the indexing scheme:

- we choose how much each a_i contributes
- we choose which bits in combined hash come from a_i

Not as good as hashing on one attribute; better than linear scan.

... Hashing and *pmr* 24/152

Multi-attribute hashing parameters:

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

---

MA.Hashing Example 25/152

Consider branch,acctNo,name,amount) table (+ hash values)

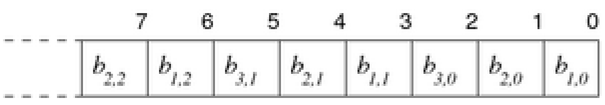
branch	h(B)	acctNo	h(Ac)	name	h(N)	amount
Brighton	1011	217	1001	Green	0101	750
Downtown	0000	101	0101	Johnson	1101	512
Mianus	1101	215	1011	Smith	0001	700
Perryridge	0100	102	0110	Hayes	0010	400
Redwood	0110	222	1110	Lindsay	1000	695
Round Hill	1111	305	0001	Turner	0110	350
Clearview	1110	117	0101	Throggs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

Note that we ignore the `amount` attribute; we are assuming, effectively, that nobody will want to ask queries like `select * from where amount=533`

... MA.Hashing Example 26/152

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

Choice vector:



This choice vector tells us:

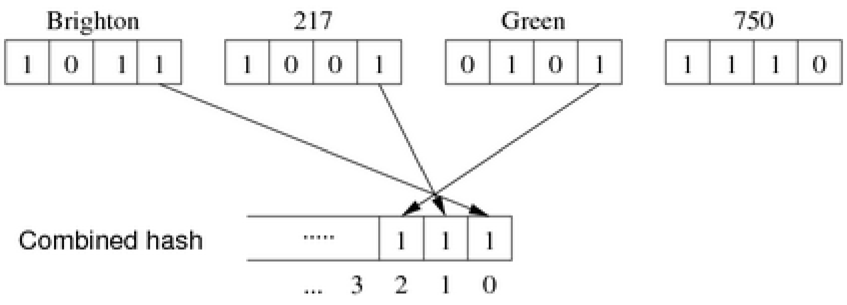
- bit 0 in hash comes from bit 0 of $h_1(a_1)$ ($b_{1,0}$)
- bit 1 in hash comes from bit 0 of $h_2(a_2)$ ($b_{2,0}$)
- bit 2 in hash comes from bit 0 of $h_3(a_3)$ ($b_{3,0}$)
- bit 3 in hash comes from bit 1 of $h_1(a_1)$ ($b_{1,1}$)
- etc. etc. (up to as many bits of hashing as required, e.g. 32)

... MA.Hashing Example

Consider the tuple:

branch	acctNo	name	amount
Brighton	217	Green	750

Hash value (page address) is computed by:

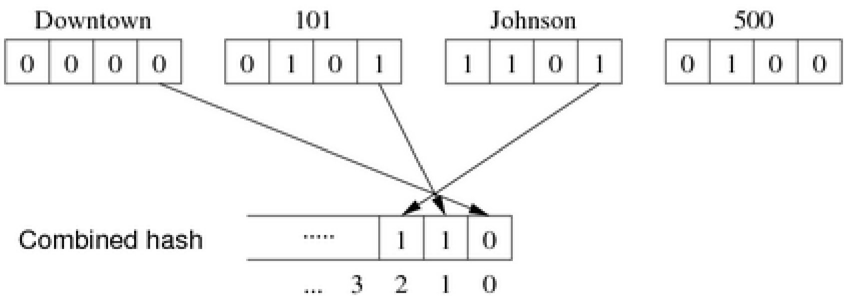


... MA.Hashing Example

Consider the tuple:

branch	acctNo	name	amount
Downtown	101	Johnston	512

Hash value (page address) is computed by:



... MA.Hashing Example

Consider the tuple:

branch	acctNo	name	amount
Round Hill	305	Turner	350

Hash value (page address) is computed by:

[Diagram:Pics/select/mahins3-small.png]

... MA.Hashing Example

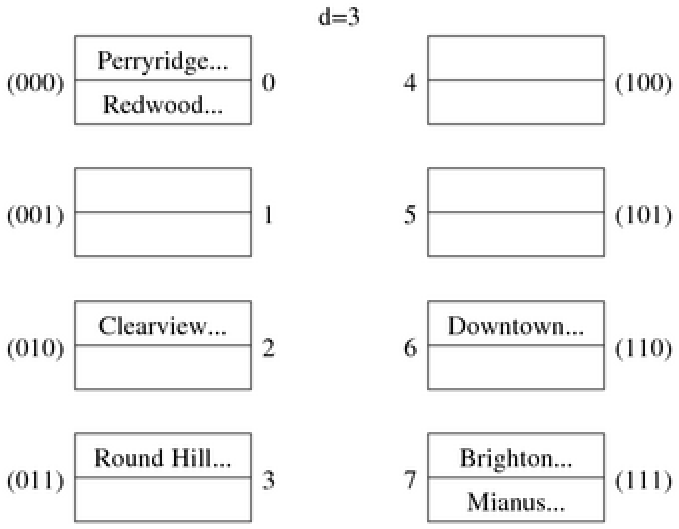
Hash values for all tuples:

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

... MA.Hashing Example

If inserted, in order of appearance, into a database containing 8 pages:



MA.Hashing Hash Functions

Auxiliary definitions:

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

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

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

// compute hash for i'th attribute of tuple t
HashVal hash(Tuple t, int i) { ... }
```

... MA.Hashing Hash Functions

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

```
HashVal hash(Tuple tup, ChoiceVec cv, int d)
{
    HashVal h[nAttr(tup)];  // hash for each attr
    HashVal res = 0, oneBit;
    int     i, attr, bpos;
    for (i = 0; i < nAttr(tup); i++)
        h[i] = hash(tup,i);
    for (i = 0; i < d; i++) {
        attr = cv[i].attr;  bpos = cv[i].bit;
        oneBit = bit(bpos, h[attr]);
        res = res | (oneBit << (i-bpos));
    }
    return res;
}
```

Queries with MA.Hashing

In a partial match query:

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

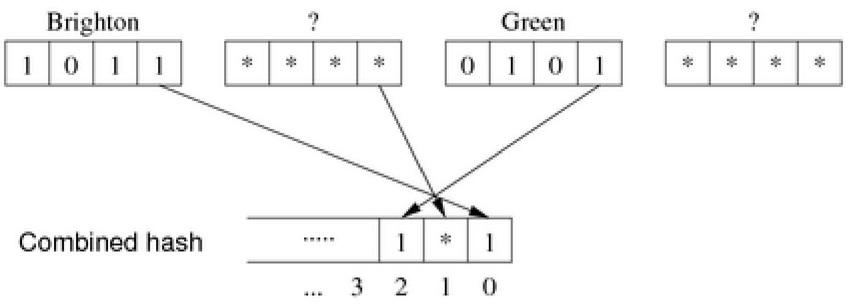
E.g.

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

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

... Queries with MA.Hashing

If we try to form a composite hash for a query, we don't know values for some bits:



... Queries with MA.Hashing

How to answer this query?

query	Hash
(Brighton, ?, Green, ?)	`1*1`

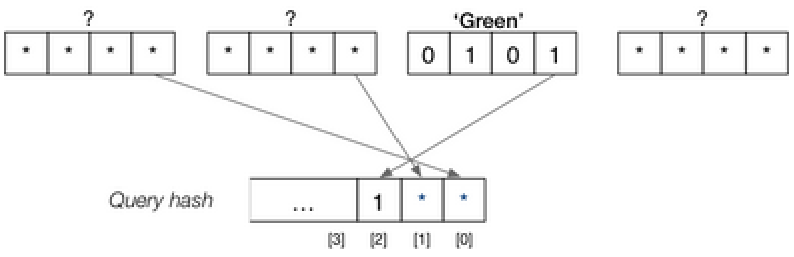
Any matching tuples *must* be in pages with addresses `101` or `111`.

We need to read just these pages.

... Queries with MA.Hashing

Consider the query:

```
select amount
from   Deposit
where  name = 'Green'
```



Need to check pages: `100`, `101`, `110`, `111`.

(With original hashing scheme, would have read all pages)

... Queries with MA.Hashing

Consider the query:

```
select amount
from   Deposit
```

```
where  branch='Brighton' and acctNo=217 and name='Green'
```

[Diagram:Pics/select/mahquery3-small.png]

Need to check only page `111`.

MA.hashing Query Algorithm

39/152

```
// Build the partial hash value (e.g. 10*0*1)
// Treats query like tuple with some attr values missing
nstars = 0;
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
        nstars++;
    }
}
...
```

... MA.hashing Query Algorithm

40/152

```
...
// Use the partial hash to find candidate pages
f = openFile(fileName("R"),READ);
for (i = 0; i < 2**nstars; i++) {
    P = composite hash
    replace *'s in P
        using i and choice vector
    Buf = getPage(f, P);
    for each tuple T in Buf {
        if (T satisfies pmr query)
            add T to results
    }
}
```

Query Cost for MA.Hashing

41/152

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)

... Query Cost for MA.Hashing

42/152

A *query type Q* may be defined via a set of attribute numbers,
indicating which attributes have values specified.

Example Query Types:

Query	Type
(v_1, ?, ?, ?)	*{1}*
(v_1, ?, v_3, ?)	*{1,3}*

$(?, ?, v_3, v_4)$	$\{3,4\}$
(v_1, v_2, v_3, v_4)	$\{1,2,3,4\}$
$(?, ?, ?, ?)$	$\{\}$ (open query)

... Query Cost for MA.Hashing

In practice, different query types are not equally likely.

E.g. maybe most of the queries asked are of the form

```
select * from R where a=1
```

with occasionally a query of the form

```
select * from R where b=3 and c=4
```

For each application, we can define a *query distribution*
which gives the probability p_Q of asking each query type Q.

... Query Cost for MA.Hashing

Consider a query of type Q with m attributes unspecified.

Each unspecified A_i contributes d_i `*`'s.

Total number of `*`'s is $s = \Sigma_{i \notin Q} d_i$.

⇒ Number of pages to read is $2^s = \Pi_{i \notin Q} 2^{d_i}$.

If we assume no overflow pages, $Cost(Q) = 2^s$ (where s is determined by Q)

... Query Cost for MA.Hashing

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:

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

Since all query types are possible, aim to minimise average cost.

Optimising MA.Hashing Cost

For a given application, the aim is 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 query type Q_j more efficient makes Q_k less efficient.

This is a combinatorial optimisation problem, and can be handled by standard optimisation techniques e.g. simulated annealing.

MA.Hashing Cost Example

Our example database has 16 possible query types:

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

Cost values are based on earlier choice vector $(d_{br} = d_{ac} = d_{nm} = 1)$
p_Q values can be determined by observation of DB use.

... MA.Hashing Cost Example

Consider $r=10^6$, $N_r=100$, $b=10^4$, $d=14$.

Attribute *br* occurs in 0.5+0.25 used query types
⇒ allocate many bits to it e.g. $d_1=6$.

Attribute *nm* occurs in 0.5+0.25 of queries
⇒ allocate many bits to it e.g. $d_3=4$.

Attribute *amt* occurs in 0.5 of queries
⇒ allocate less bits to it e.g. $d_4=2$.

Attribute *ac* occurs in 0.25 of queries
⇒ allocate least bits to it e.g. $d_2=2$.

... MA.Hashing Cost Example

With bits distributed as: $d_1=6, d_2=2, d_3=4, d_4=2$

Query type	**Cost**	p_Q
(*br*, ?, ?, ?)	$2^8 = 256$	0.25
(?, *ac*, *nm*, ?)	$2^8 = 256$	0.25
(*br*, ?, *nm*, *amt*)		0.5

	$2^2 = 4$	

Cost = $0.5 \times 2^2 + 0.25 \times 2^8 + 0.25 \times 2^8 = 130$

MA.Hashing vs One-Attribute Hashing

Multi-attribute hashing is basically just a different way of forming hash value for each tuple.

Hence, can be used with all flexible hashing schemes.

As the file grows, the d_i's are increased one by one, using the choice vector.

Difference between *mah* and standard hashing:
queries "suggest" a set of pages rather than a single page.

Cost for insertion, deletion, ordered scan is same as for underlying hashing scheme.

Grid Files

The *mah* approach attempts to combine all attributes in the hash value.

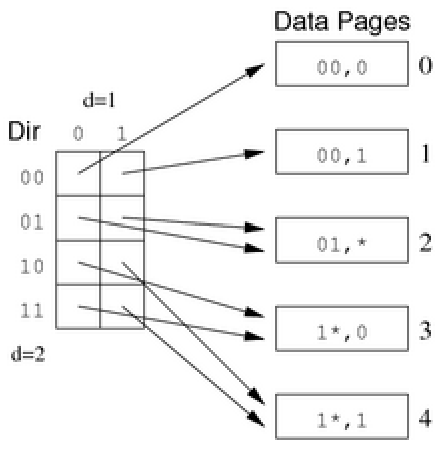
Alternative approach is to keep hash values separate and use auxiliary structure to provide combination.

A generalisation of extendible hashing, called *grid files* provides this.

Ext.hashing uses a directory on one attribute.

Grid files use a *k*-dimensional directory to handle *k* attributes.

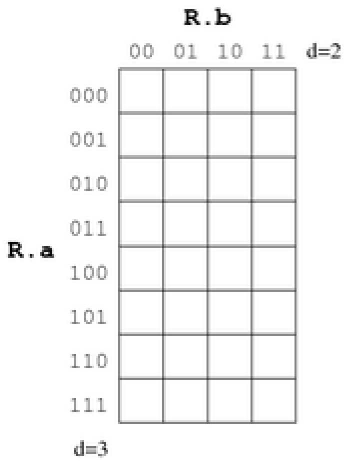
... Grid Files



A simple 2-dimensional grid file

Selection with Grid Files

Consider the following grid file directory:



for a table *R* with two attributes *a* and *b*

... Selection with Grid Files

54/152

If $d_1=3$ and $d_2=2$ ...

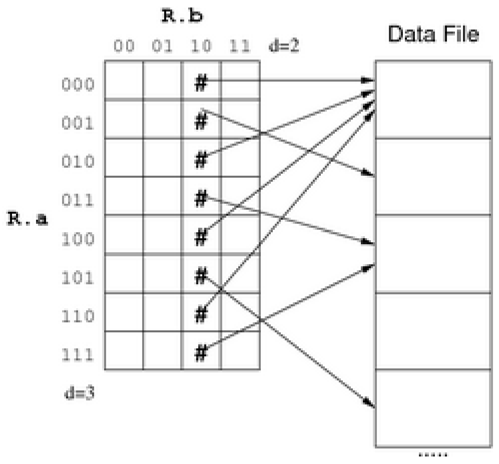
`select...where a=`C_1` and b=`C_2	one cell
`select...where a=`C_1	one row (four cells)
`select...where b=`C_2	one column (eight cells)

Number of directory cells visited is similar to number of pages visted in *mah*.

... Selection with Grid Files

55/152

Execution of `select...where b=`C_2:



where *hash(C_1)* = `...001` and *hash(C_2)* = `...110`

... Selection with Grid Files

56/152

Selection for *pmr* in a 2d grid file (*nAttrs = d*)

```
for (i = 0; i < nAttr(tup); i++) {
    if (!hasValue(tup,i))
        { lo[i] = 0; hi[i] = 2**d-1 }
    else {
        val = bits(d[i], getAttr(tup,i))
        lo[i] = val; hi[i] = val
}   }
Pages = []
for (i = lo[1]; i <= hi[1]; i++) {
    for (j = lo[2]; j <= lo[2]; j++) {
        if (!(grid[i][j] in Pages))
            add grid[i][j] to Pages
}   }
for each P in Pages {
    Buf = getPage(f,P)
    check Buf for solutions
}
```

Query Cost for Grid Files

57/152

Cost depends on:

- how many attributes are unspecified in query
- "page density" along each dimension of grid

For *pmr* with all attributes specified

- determine single grid cell coordinate
- read page containing that cell
- read correpsonding data page

$C_{(x,y,z)} = 2$

... Query Cost for Grid Files

58/152

For *pmr* with $j < k$ attributes specified

- access all grid cells in k-j dimensional region
- i.e. g cells located in g_q grid directory pages
- then need to read $b_q \leq m \leq g$ pages

$C_{(?,y,?)} = (g_q + m)$

Typically, $1 \leq g_q \leq 50$ (i.e. grid directories are relatively small)

Other Operations on Grid Files

59/152

Insertion has several cases:

- no splitting required ($2_r + 1_w$)
- splitting but no dir. expansion ($2_r + 3_w$)
  (write the two data pages from split, plus page to update on directory cell)
- splitting + directory expansion (expensive)

Deletion can be achieved by marking (so $Cost_{pmr} + b_{q\ w}$)

As with all hashing schemes, grid file does not help ordered scan.

N-dimensional Tree Indexes

Multi-dimensional Tree Indexes

61/152

We have seen how hashing can be extended to handle multiple attributes.

Can tree-based index structures be generalised as well?

Yes ... and a very large number of techniques have been suggested in research literature.

E.g. BSP-trees, Cell-trees, LSD-trees, SS-trees, TV-trees, X-trees, ...

We consider three popular examples: kd-trees, Quad-trees, R-trees.

Example 2d Relation

62/152

As an example for multi-dimensional trees, consider the following relation:

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

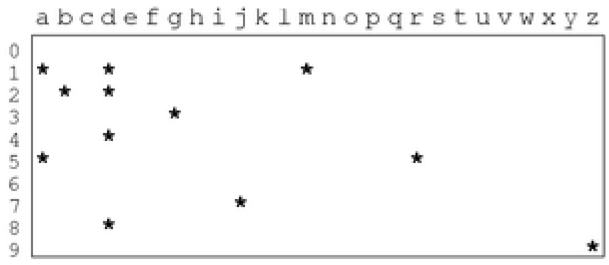
with example tuples:

```
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)
```

... Example 2d Relation

63/152

And the tuple-space for the above tuples:



kd-Trees

64/152

kd-trees are multi-way search trees where

- each level of the tree uses a different attribute to partition tuples
- each node contains *n-1* key values, pointers to *n* subtrees

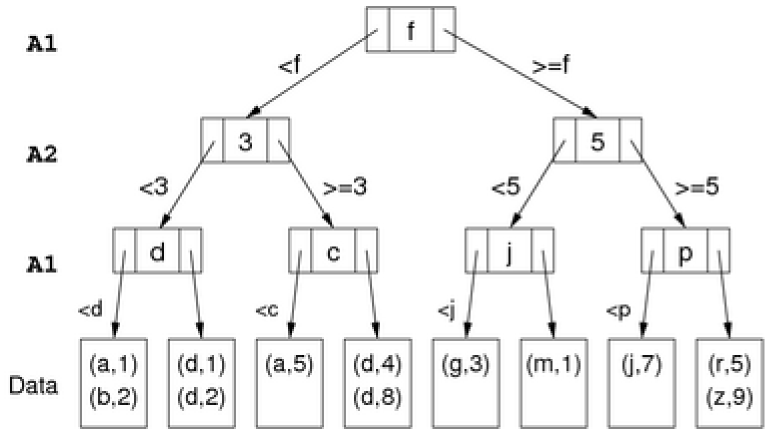
For a tree with *L* levels and *k* attributes:

- if $L < k$, some attributes are not indexed
- if $L = k$, each attribute is used once in partitioning
- if $L > k$, "cycle through" attributes
  (e.g. $L=5$, $k=3$ ... level 1 uses a_1, level 2 uses a_2, level 3 uses a_1, level 4 uses a_2, level 5 uses a_1)

Partitions "tuple space" into smaller and smaller regions at each level.

Example kd-Tree

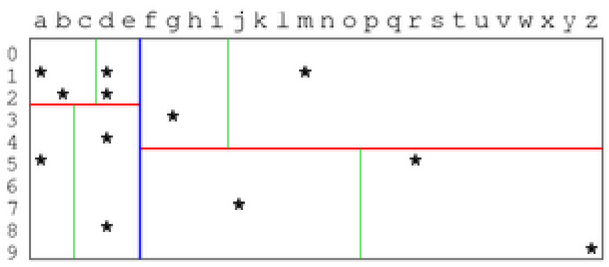
65/152



Each leaf node contains a reference to a bucket containing tuples from a small region of *k*-dimensional space.

... Example kd-Tree

How this tree partititons the tuple space:



🟦 level 1, partitioning based on A1

🟥 level2, partitioning based on A2

🟩 level 3, partitioning based on A1

Searching in kd-Trees

Consider first how to answer *pmr* queries ...

If all attributes have values specified:

- use A_1 value in root node
- use A_2 value in first level node
- use A_1 value in second level node, ...
- until reach leaf node

If e.g. attribute A_2 is unspecified

- use A_1 value in root node
- examine all sub-trees of first level node
- use A_1 value in second level node, ...
- until reach leaf node

... Searching in kd-Trees

As a tree traversal, the algorithm is best specified recursively.

Parameters for search function:

- *Q* ... the query, including unspecified attributes
- *L* ... current level in search tree
- *N* ... current node in search tree

Other available information:

- *attrLev* ... array indicating which a_i for which level

The search commences via `Search(Q, 0, kdTreeRoot)`

... Searching in kd-Trees

```
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)
}   }   }
```

... Searching in kd-Trees

For *space* queries ...

We require a change to the above `Search` algorithm.

```
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 cost for kd-Trees

Different kinds of queries (i.e. different specified attributes)

- lead to different amounts of the tree being traversed
- lead to different numbers of data pages being accessed

The query cost will clearly also be affected by the depth of the tree.

If we include all *n* attributes as indexes, then tree has depth ≥ *n*.

Depth of tree is also affected by branching factor at each node.

Note: kd-trees are not necessarily balanced.

... Query cost for kd-Trees

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, ...

Insertion into kd-Trees

Input: tuple with all values specified.

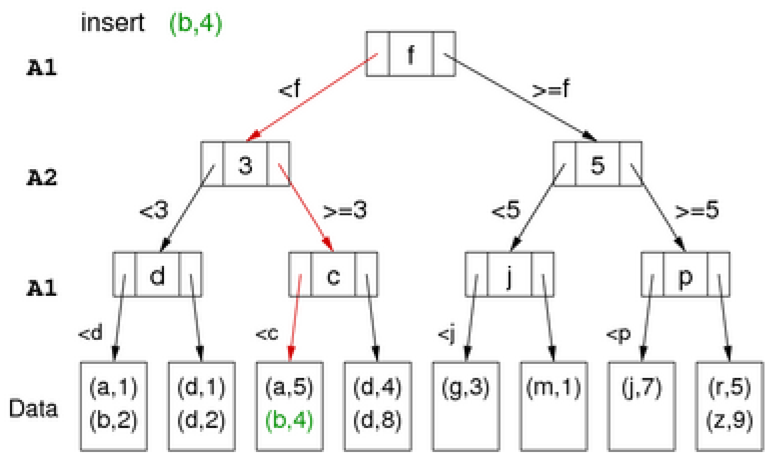
Method:

```
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)
```

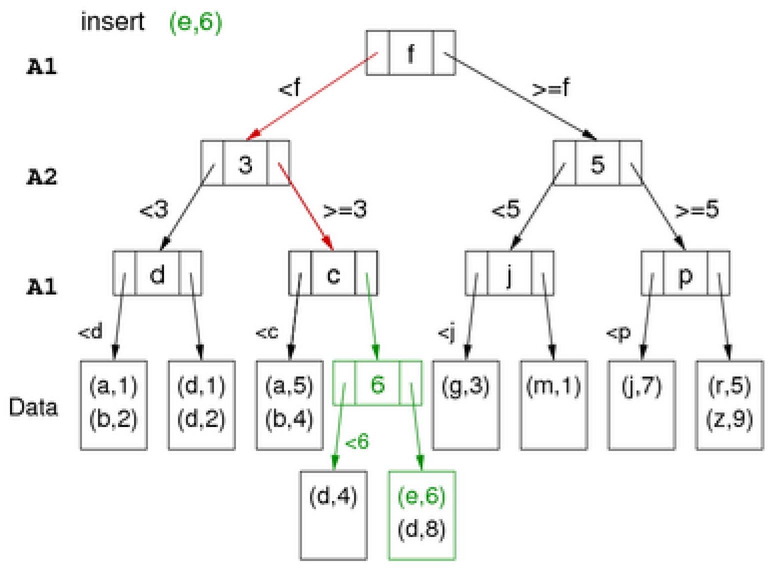
... Insertion into kd-Trees

Example insertion into non-full data page:



... Insertion into kd-Trees

Example insertion into full data page:



kd-B-Trees

kd-B-trees: a disk-based index structure using kd-tree ideas.

Aims to have a tree which is:

- is reasonably well-balanced (⇒ shallow)
- reasonable occupancy of nodes

Basic idea: distribute kd-tree structure over a number of pages:

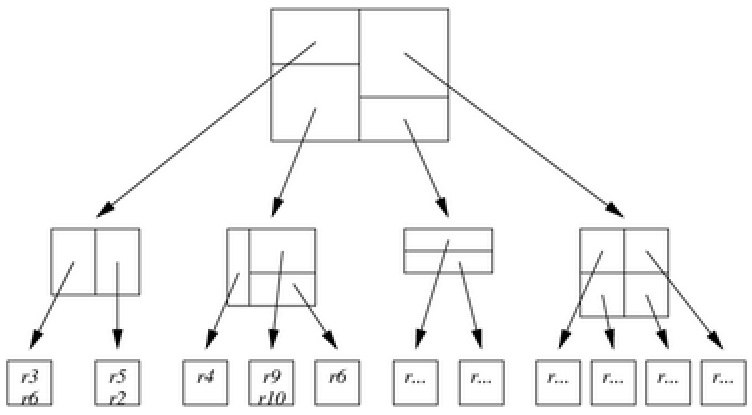
- root page contains levels *1..m* of kd-tree

- child pages contain levels *m+1..2m* of kd-tree, etc.

Requires modification of insertion method to do re-balancing
(changes to insertion make it more complex and potentially much more expensive)

... kd-B-Trees

Simple example kd-B-tree (max 2 kd-tree levels in each node):



... kd-B-Trees

What this produces:

- each index page partitions a region of tuple space (disjoint)
- root page partitions whole space
- child nodes partition one sub-region from parent
- union of partitions on one level always covers whole space

Potential problems:

- some index pages are "under-occupied" (sparse region of space)
- re-balancing can require significant tree re-organisation

Selection in kd-B-Trees

To answer a *pmr* query:

```
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
}
```

... Selection in kd-B-Trees

Size of kd-B-tree:

- each kd-tree node is (lchild,keyVal,rchild)
- for integer key, node could be represented in 12 bytes
- for 4KB index pages, #nodes/page is approx 300
- for balanced kd-tree, can store 8 levels per index page
- gives 128 level-8 nodes ⇒ 256 possible child index pages
- also, gives total "depth" of tree *d = 2*
  (max number of page reads needed to reach any leaf kd-tree node)
- this allows us to index 64K data pages

Obviously, depth increases for not perfectly-balanced trees.

... Selection in kd-B-Trees

If all attributes are specified (i.e. point search):

$Cost_{pmr} = (d+1)$

If some attributes unspecified ...

- need to search multiple paths for unknown attributes
- leading to many leaf nodes and thus many data pages

Can potentially search a large region of tree:

- which requires us to read e.g. dozens of index pages
- and may require us to read hundreds of data pages

space queries can be handled similarly to *pmr* queries.

Insertion/Deletion in kd-B-Trees

Conceptually, insertion is same as for kd-tree:

- first perform point search to find data page
- if not full, add new tuple
- if full, add new data page and kd-node, and partition

Added complication: kd-tree is spread over many index pages

- if index page not full, split needs no new index page
- if index page is full, need to consider
  - adding new index page and expanding tree
  - reorganising tree within existing index page
    (which requires re-partitioning tuples between data pages)
    (may need to propagate re-organisation up to parent index page)

... Insertion/Deletion in kd-B-Trees

Minimum insert cost is 1_w (no split, no change to kd-tree)

Insert cost with split and kd-tree update: 3_w
(need to write re-partitioned data pages and modified index page)

Worst case insert cost is high (tree/data re-organisation).

Deletion is straightforward (mark as deleted).

If data page occupancy falls too low, might consider

- merging content of sparse data pages
- removing corresponding leaf nodes from kd-tree

Quad Trees

Quad trees use a regular, recursive partitioning of *k*d tuple space.

At each level, partition space into non-overlapping *k*d hypercubes.

For 2d case (others are much harder to visualise):

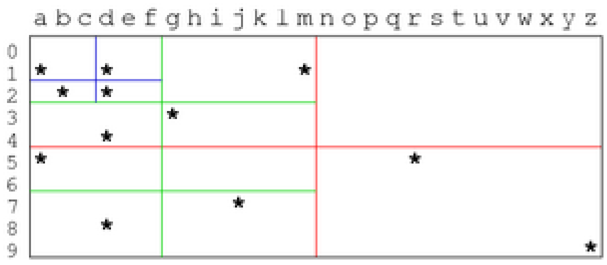
- partition space into quadrants (NW, NE, SW, SE).
- each quadrant can be further subdivided into four, etc.

Aim: each "leaf" partition contains approx same number of tuples.

... Quad Trees

85/152

Quad-tree on example 2d tuple space:



... Quad Trees

86/152

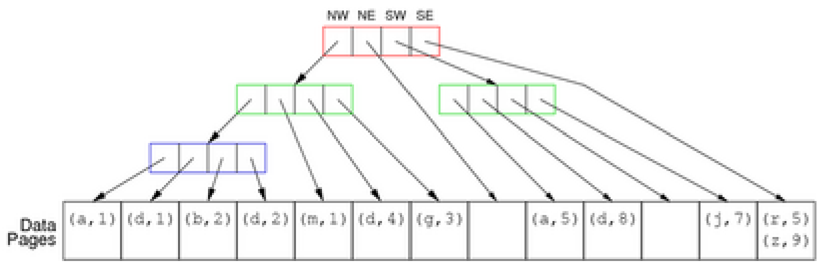
Basis for the partitioning:

- a quadrant that has no sub-partitions is a *leaf quadrant*
- each leaf quadrant maps to a single data page
- subdivide until points in each quadrant fit into one data page
- ideal: same number of points in each leaf quadrant (balanced)
- point density varies over space
  ⇒ different regions require different levels of partitioning
- this means that the tree is not necessarily balanced

... Quad Trees

87/152

The previous partitioning gives this tree structure, e.g.



... Quad Trees

88/152

Each node of a *k* dimensional quad-tree has 2^k children.

Quad-trees originally devised as memory based structures, for spatial data.

In this domain, $k = 2$ or $k = 3$ and branching is fairly small.

For index pages in a disk-based data structure:

- if $k < 6$, then can store multiple nodes in a page
- if $8 \le k \le 10$, then one node per page
- if $k > 10$, this storage structure becomes less feasible

89/152

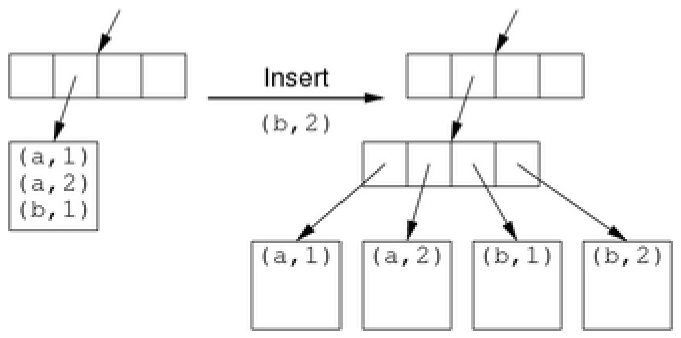
Insertion into Quad Tree

Inserting a new tuple into a *2d* quad-tree indexed file involves:

```
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)
```

... Insertion into Quad Tree

Example:



Cost = traversal cost + update cost

= read ≥ *1* node pages + write ≥ *1* data pages

... Insertion into Quad Tree

Potential problems with quad-trees:

- creating empty data pages during partitioning
- creating tree-nodes with 2^k entries
  (many of which are likely to be empty on the first split)

The problems can be overcome by:

- not allocating pages until there is data to go in them
  (so, also need to mark corresponding entries in tree-nodes)
- storing tree-nodes in a more compact form
  (don't store entries that don't (yet) have a pointer to a data page)

Query with Quad Tree

For *pmr* query

- if zero attributes known, quad-tree provides no assistance
- if one attribute known, need to traverse two branches from each node ⇒ examine half of tree
- if two attributes known, follow single branch from each node

For *k*d quad-tree, and query with *n* specified attributes:

- need to follow 2^{k-n} branches at each level
- e.g. for *k=5, n=3*, follow 4 (out of 32) branches at each level

---

... Query with Quad Tree

For *space* query

- query condition identifies a region of tuple-space
- need to examine all branches of tree which intersect this region

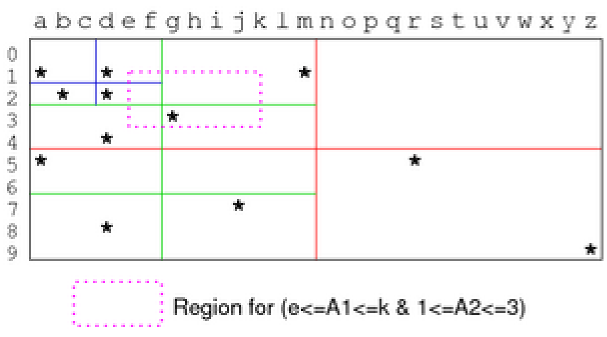
Example: *2d* quad-tree with query $a \le A1 \le b \wedge c \le A2 \le d$

- identifies rectangular region with vertices (a,c),(a,d),(b,d),(b,c)
- read all leaf quadrants (⇒ data pages) intersecting this region

---

... Query with Quad Tree

Space query example:


a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
Region for (e<=A1<=k & 1<=A2<=3)

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

R-Trees

R-trees use a flexible, hierachical partitioning of *k*d tuple space.

- each node in the tree represents a *k*d hypercube
- its children represent (possibly overlapping) subregions
- the child regions do not need to cover the entire parent region

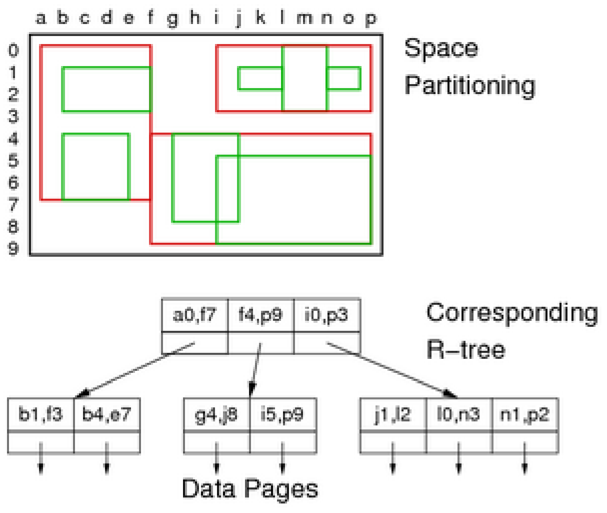
Could be viewed as a more flexible version of quad-tree.

... R-Trees

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

[Diagram:Pics/select/r-tree-node-small.png]

... R-Trees



... R-Trees

The R-tree was initally designed for indexing polygons
(via their minimum bounding rectangles (MBR))

It adapts naturally to tuple-space points (i.e. tuples)
(each MBR represents a region of the tuple-space)

Goals in constructing R-trees (cf. B-trees)

- every node, except root, contains between *m* and *M* entries
- the root node has at least two entries, unless it is also a leaf
- the tree is height-balanced (all data pages at same level)

Query with R-trees

Designed to handle *space* queries and "where-am-I" queries.

"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*

Space (region) queries are handled in a similar way

- we traverse down any path that intersects the query region

... Query with R-trees

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;
}
```

Insertion into R-tree

Insertion of an object *R* occurs as follows:

- start at root, look for children that completely contain *R*
- if no child completely contains *R*, *choose one* of the children and expand it so that it does contain *R*
- if several children contain *R*, *choose one* and proceed to child
- repeat above containment search in children of the current node
- once we reach data page, insert *R* if there is room
- if no room in data page, replace by two data pages
- *partition* existing objects between two data pages
- update node pointing to data pages
  (may cause B-tree-like propagation of node changes up into tree)

Note that *R* may be a point or a polygon.

... Insertion into R-tree

Heuristics in R-tree insertion ...

Choose child to expand to contain *R*

- maximise overlap with child region; minimise expansion of child

Choose a child (from several) to contain *R*

- use information about regions further down tree
- if not available, make an arbitrary choice

Partition objects between old/new data pages

- requires finding a suitable "split point"
- different to B-tree (*1d*) where can always find 50:50 split point
- for *d>1*, cannot generally do this without overlap (minimise)
- common technique for splitting: quadratic (time) split

... Insertion into R-tree

Quadratic split method to partition objects in *S*:

```
find objects a and b in S
    that produce the largest MBR
place a in S_1
place b in S_2
for every other object c in the S {
    s1 = current size of S_1
    i1 = size of S_1 with c added
    s2 = current size of S_2
    i2 = size of S_2 with c added
    if ((s1-i1) < (s2-i2))
        add c to S_1
    else
        add c to S_2
}
```

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

R-trees in PostgreSQL

Up to version 8.2, PostgreSQL had separate R-tree implementation

- src/backend/access/rtree
- **`rtget.c`** ... iterator for R-tree scans
- **`rtree.c`** ... R-tree construction operations
- uses quadratic split (**`rtpicksplit()`** in **`rtree.c`**)

From version 8.2 on, R-trees implemented using GiST.

GiST in PostgreSQL

PostgreSQL provides an implementation of Generalized Search Trees (GiST).

Above discussion of tree-based indexes shows commonalities:

- trees are built of nodes containing many index entries
- choice of branches is determined by a predicate *p*
- for each index entry *(p,ptr)* in a leaf node,
  page *ptr* holds tuples staisfying *p*
- for each index entry *(p,ptr)* in a non-leaf node,
  all tuples reachable via *ptr* satisfy *p*

Some examples:

- for B-trees, predicates are order operators on keys
- for R-trees, predicates are containment on MBRs
- for kd-trees, predicates alternate keys on each level

... GiST in PostgreSQL

GiST trees have the following structural constraints:

- every node is at least fraction *f* full (e.g. 0.5)
- the root node has at least two children (unless also a leaf)
- all leaves appear at the same level

However, the "predicate" constraints are looser than those for any trees discussion so far.

Users are free to implement their own *p* to produce their own "flavour" of search tree.

Details: **`src/backend/access/gist`**

Signature-based Selection

Indexing with Signatures

Signature-based indexing: an "efficient" linear scan.

Why abandon sub-linear complexity? (e.g. *O(1)* for hashing)

Arising from work with high-*d* feature vectors in multimedia:

- all indexing methods degenerate to *O(n)* as *d* increases
- absolute bound on *d* before linear scan is best *d = 610*
- in practice, most methods degenerate for *10 ≤ d ≤ 40*

We consider two applications of signatures to indexing:

- superimposed codewords for *pmr* queries on relational data
- VA files for *kNN* queries on multimedia data

... Indexing with Signatures

A *signature* is a compact (lossy) descriptor for a tuple.

Scanning the whole data file is too expensive, so

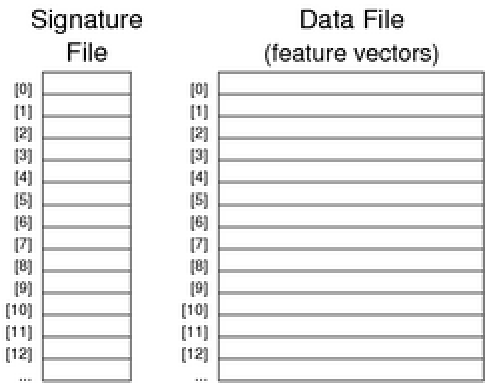
- maintain a signature file in parallel with the data file
- to answer queries, scan all signatures to choose likely tuples
- fetch only data pages which are now known to contain "matches"

For this to be better than a linear scan of the data, require:

- signatures are *substantially smaller* than data tuples
- cost of checking signatures is *much less* than checking tuples
- that signatures must provide effective filtering (minimal false matches)

Signature Files

File organisation for signature indexing (two files)



(One signature slot per tuple slot in the data file; unused signature slots are zeroed)

... Signature Files

Note that signatures do not determine tuple placement.

The only requirement is that

- after a tuple has been placed in the data file
- a signature must be placed in the corresponding slot in the signature file
  (clearly, this means that if there are *r* tuples in the data file, there will be *r* signatures)

The data file could be organised via hashing, B-tree, curves, etc.

Thus, signatures can be used in conjunction with other indexing schemes.

Signatures

Signatures (*descriptors*)

- must ``summarise'' all (indexed) fields in a tuple
- must be able to be "checked" efficiently in queries

There are two main types of tuple signature:

- superimposed codewords (overlay "signatures" for all attributes)
- disjoint codewords (concatenate "signatures" for all attributes)

We use the following terminology consistently:

- descriptor = signature for an entire tuple
- codeword = signature for a single attribute

... Signatures

A tuple *r* consists of *n* attributes $A_1..A_n$.

A *codeword* $cw(A_i)$ is

- a bit-string, *m* bits long, where *k* bits are set to 1 ($k \ll m$)
  (this kind of representation is called *k-in-m encoding*)
- derived from the value of a single attribute A_i

A *tuple descriptor* is built by combining *n* codewords.

Generating Codewords

A method for generating a *k-in-m* codeword for attribute A_i

```
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++
    }
}
```

Requires bit-string operations and a pseudo-random number generator.

Superimposed Codewords (SIMC)

In a superimposed codewords (simc) indexing scheme

- a tuple descriptor is formed by overlaying attribute codewords

A tuple descriptor *desc(r)* is

- a bit-string, *m* bits long, where $j \le nk$ bits are set to 1
- formed by bitwise `OR` of *n* attribute codewords
- $desc(r) = cw(A_1)$ `OR` $cw(A_2)$ `OR` ... `OR` $cw(A_n)$

SIMC Example

Consider the following tuple (from a bank deposit database)

branch	acctNo	name	amount
Perryridge	102	Hayes	400

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

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

... SIMC Example

Consider a very small example database, with associated signatures:

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)

SIMC Queries

119/152

To answer query *q* in SIMC

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

desc(q) is formed by bitwise `OR` of codewords for supplied attributes.

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

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

... SIMC Queries

120/152

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

```
for each descriptor D[i] in signature file {
    if (D[i] matches desc(q))
        mark tuple R[i] as a match
}
for each marked tuple R {
    fetch tuple R
}
```

The critical assumption to make this approach viable:

> scanning the signature file and comparing descriptors
> is faster than scanning the tuple file and checking tuples

... SIMC Queries

121/152

What sort of signature comparison is required to answer the example query?

Clearly, any matching tuple must have the value "Perryridge" for A_1.

Any tuple with "Perryridge" must have these bits `010000000001` set.

So, checking whether each descriptor has these bits set, gives us the matches.

This can be generalised to multiple supplied values in the query:

- we superimpose all the codewords for all supplied values
- any matching tuples must have all of those bits set

In other words ...
> for any tuple *r* that is a match for query *q*
> the 1-bits in *desc(q)* are a subset of the 1-bits in *desc(r)*

... SIMC Queries

Query/tuple descriptor matching can be implemented efficiently using logic operations on bit-strings.

Note that if *bits(A)* ⊆ *bits(B)* then *A* AND *B = A*.

Thus, the query/tuple descriptor matching can be implemented as:

matches(R_i,q) if *desc(q)* AND *desc(R_i) = desc(q)*

Example SIMC Query

Consider the query and the example database:

Signature	Tuple
010000000001	(Perryridge,?,?,?)
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)

... Example SIMC Query

The query has potential matches:

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)

False Matches

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

... False Matches

How to reduce likelihood of false matches?

- hash each attribute using a different hash function (h_i for A_i)
- 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.

Design with SIMC

SIMC schemes have the notion of *false match probability* p_F.

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. $p_F \le 10^{-5}$ 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 with SIMC

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)

Query Cost for SIMC

A page-oriented view of how SIMC queries are answered:

```
determine query descriptor QD
for each page SB of tuple descriptors {
    for each tuple descriptor RD in page SB {
        if (RD matches QD) {
            add tuple R to list of matches
            add pageOf(R) to Pages
        }
    }
}
for each page P in Pages {
    Buf = getPage(f,P)
    check Buf for matching tuples
}
```

With a buffer manager, the previous tuple-oriented version would behave like this.

... Query Cost for SIMC

To answer *pmr* query, must read *r* signatures.

Since one signature per tuple, cannot feasibly store them in memory.

If signature contains *m* bits, can fit $N_D = \lfloor 8B/m \rfloor$ signatures per page.

Thus, must read $b_D = \lceil r/N_D \rceil$ pages of signature data.

E.g. $m=32, \quad B=1024, \quad r=10^5 \quad \Rightarrow \quad N_D = 256, \quad b_D=391$

How many data pages do we read?

... Query Cost for SIMC

Number of data pages read depends on:

- number of matching tuples r_q
- number of false matches r_F
- how matching tuples are distributed

Total tuples to read = $r_q + r_F$.

Expected false matches = $r_F = (r - r_q).p_F$.

Assume matching tuples are uniformly distributed over the data pages.

Then total data pages read = $b_q = \lceil (r_q+r_F)/r \times b \rceil$

$Cost_{pmr} = b_D + b_q$

... Query Cost for SIMC

For *one* queries, can use same optimisation as for Heaps.

That is, stop the search as soon as the single matching tuple is found.

The average cost in the case would be:

- reading half of the descriptors $b_D/2$
- reading the page containing the match
- reading some page(s) containing false matches $\delta \le r_F$

$Cost_{one}$: Best = 2 Average = $b_D/2 + \delta$ Worst = $b_D + b$

... Query Cost for SIMC

SIMC provides no assistance in answering *range*, *space* queries, unless the data file is sorted.

If the data file is sorted,

- search for the first tuple with the low value (as for a *one* query)
- then use sequential scan to find the rest

SIMC provides no asistance at all for *sim* queries (linear scan).

Applications of SIMC

SIMC is useful for *pmr*, which allows a wide range of query types, e.g.

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

The above discussion has assumed that *n* attributes are indexed.

In fact, SIMC is reasonably flexible about how many attributes are involved

Thus, it also has applications in information retrieval:

- form a document signature by superimposing codewords for each keyword in the document
- form a query signature by superimposing codewords for each keyword in the query
- allows us to find a set of documents containing all query keywords

Two-level SIMC

Scanning one descriptor for every tuple is not efficient.

How to reduce number of descriptors?

Have a smaller number of larger descriptors.

E.g. one descriptor for each data page.

Every attribute of every tuple in page contributes to descriptor.

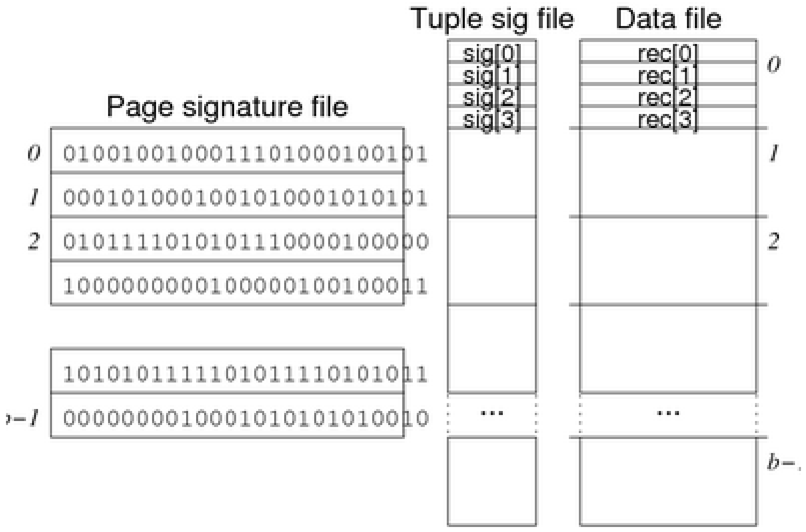
How large is a page descriptor (BD)?

Use formulae above, but with $N_r.n$ ``attributes''.

Typically, pages are 1..8KB $\Rightarrow$ 8..64 BD/page (N_{BD}).

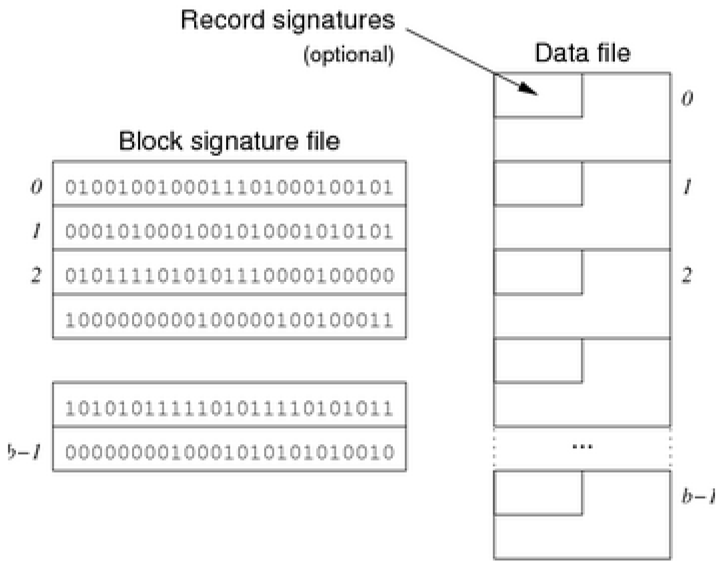
Two-Level SIMC Files

File organisation for two-level superimposed codeword index



... Two-Level SIMC Files

Alternative file organisation:



(Assumes that we need to read the data pages anyway, so may as well include tuple descriptors there ... or simply omit them altogether)

Queries with Two-level SIMC

Method for answering queries:

```
form query descriptor QD
for each page descriptor BD {
    if (BD matches QD)
        add P to Pages
}
for each page P in Pages {
    read page P
    check for matching tuples
    // could do this using tuple descriptors
}
```

$Cost_{pmr} = \lceil b/N_{BD} \rceil + b_q$

Bit-sliced SIMC

Reading all page descriptors is still not efficient, especially when only a few bits are set.

To improve efficiency, reconsider the matching process:

Signature	**Tuple**
`010000000001`	`(Perryridge,?,?,?)`
`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)

... Bit-sliced SIMC

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

[Diagram:Pics/select/bit-slice-small.png]

... Bit-sliced SIMC

141/152

To answer queries:

```
form query descriptor QD
// Result is a bit-slice
Result = AllOnes
for each bit b set in QD {
    read bit-slice b
    Result = Result AND b
}
```

`Result` has bit *i* set if page *i* is candidate page.

Queries much more efficient because often only a few bits set in QD.

However, updates are expensive ... need to set *k* different slices for each insert.

Changing *b* is *very* expensive.

Query Cost for Bit-sliced SIMC

142/152

A major cost in SIMC queries is reading the page-descriptor slices.

Each slice is *b* bits long (one bit for each data file page).

Thus, each slice occupies $\lfloor 8B/b \rfloor$ bytes.

Generally, we can fit several page descriptor slices per page.

However, we make the pessimistic assumption ...

> reading a slice means reading one page from the page descriptor slices file

... Query Cost for Bit-sliced SIMC

143/152

Consider a query `(val1, ?, val2, ?, ...)` for an *n* attribute relation:

- if *m* attribute values are supplied in the query
- then approximately *mk* bits are set in the query page-descriptor
- and so we need to read *mk* descriptor slices

In fact, we can optimise further by always reading just the first *j* of these *mk* slices.

Thus, the descriptor-reading cost in queries can be made constant.

The downside of this trick is a (slight) increase in likelihood of false matches.

$Cost_{pmr} \ = \ j + b_q$

Disjoint Codewords (DJC)

144/152

In a disjoint codewords (djc) indexing scheme

- a tuple descriptor is formed by concatenating attribute codewords

A tuple descriptor *desc(t)* is

- a bit-string, *nm* bits long, where *nk* bits are set to 1
- formed by concatentation of *n* attribute codewords
- $desc(t) = cw(A_1)cw(A_2)...cw(A_n)$

145/152

DJC example

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*, *k = 2*)

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

(Note: length of tuple = ~28 bytes, length of descriptor = 16 bits = 2 bytes)

DJC Queries

Use a similar approach to SIMC:

- first generate a query descriptor *desc(q)*
- then use query descriptor to search signature file

desc(q) is formed by concatentation of codewords

- each supplied attribute A_i contributes $cw(A_i)$
- each unknown attribute contributes `0000` (as many zero bits as needed)

... DJC Queries

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

A_i	$cw(A_i)$
Perryridge	`0101`
?	`0000`
Hayes	`1001`
?	`0000`
desc(t)	`0101000010010000`

If the query has *s* supplied attributes, then *desc(q)* has *sk* bits set to 1.

... DJC Queries

Searching the signature file is done in the same way as for SIMC:

```
Matches = {}
for each descriptor D[i] in sig file {
    if (D[i] matches desc(q))
        Matches = Matches ∪ TupleId[i]
}
for each Tid in Matches
    fetch tuple t via Tid
```

The matching process is also the same as for SIMC

matches(T_i,q) if *desc(q)* AND *desc(T_i) = desc(q)*

Example DJC Query

Consider the query and the example database:

Signature	Tuple
`0101 0000 1001 0000`	`(Perryridge,?,Hayes,?)`
`1010 0110 1001 0110`	(Brighton,217,Green,750)
`0101 0101 0101 0101`	(Clearview,117,Throggs,295)
`1010 1010 1001 1010`	(Downtown,101,Johnson,512)
`1010 0011 0011 0101`	(Mianus,215,Smith,700)
`0101 0011 1001 1010`	(Perryridge,102,Hayes,400)
`1001 0101 0011 1010`	(Redwood,222,Lindsay,695)
`0101 0110 1001 0110`	(Round Hill,305,Turner,350)

False Matches

The query has potential matches:

branch	acctNo	name	amount
Perryridge	102	Hayes	400
Round Hill	305	Turner	350

DJC also suffers from the "false match" problem, due to:

- codeword hashing collisions (two different values generate same codeword)

Note:

- there are no false matches due to overlapping in DJC
- likelihood of hash collisions is much higher for DJC than for SIMC
  (e.g. *k=2, m=4 ⇒ 6* codewords for DJC vs *k=2, m=16 ⇒ 120* codewords for SIMC)

... False Matches

How to reduce likelihood of false matches?

- increase descriptor size (*m*)
- choose *k* so that ≅ half of bits are set (maximises distinct descriptors)

Since descriptor is formed from concatentation of codewords:

- need to keep codewords short to keep descriptor reasonably small
- no need to use the same number of bits for each attribute codeword
  ⇒ choose more bits for attributes that are often supplied in queries

Thus, for each A_i, m_i bits in the codeword, with $m_i/2$ 1-bits.

DJC can be "tuned" to the mix of *pmr* queries used on the relation.

Query Cost for DJC

If we assume that

- DJC descriptors are the same length as SIMC descriptors
- p_F is similar for both schemes (reasonable)

then the average query cost analysis is exactly the same as for SIMC.

However, we can make sure that certain query types are answered with less likelihood of false matches by adjusting the codeword lengths.

The optimisations from SIMC (two-level, bit-slice) can also be applied.

As for SIMC, DJC assists with *one* and *pmr*, but not *range*, *space*, *sim*.

Produced: 12 Jul 2019