Week 07 Lectures

Signature-based Selection

Indexing with Signatures 2/73

Signature-based indexing: represent a tuple by a compact bit string

- 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*

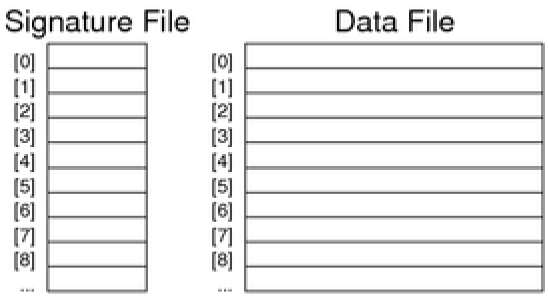
- a compact (lossy) descriptor for the tuple
- formed by combining information from multiple attributes
- stored in a signature file, parallel to data file

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

... Indexing with Signatures 3/73

One file for signature and one file for data

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 4/73

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 $cw(A_i)$, $i=1..n$

- combine by *overlaying* codewords (bitwise-OR)
- aim to have roughly half of the bits set to 1

Generating Codewords 5/73

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
}
```

Set a seed with hash value.

Superimposed Codewords (SIMC)

6/73

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
- $desc(r) = cw(A_1)$ 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])
   desc = desc | cw
}
```

for a descriptor, any position of any attr with value 1, the corresponding position of the desc

SIMC Example

7/73

Consider the following tuple (from bank deposit database)

Branch	AcctNo	Name	Amount
Perryridge	102	Hayes	400

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

Codeword

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

m is the number of bit in total, k is the number of bit that need to be set

SIMC Queries

8/73

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 OR of codewords for known attributes.

E.g. consider the query `(Perryridge, ?, ?, ?)`. There also will be a desc for the query, which is used to match record(tuples)

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

If a tuple match the desc but doesn't match the attr, then it's a false match

---

... SIMC Queries

9/73

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

```
pagesToCheck = {}
for each descriptor D[i] in signature file {
    if (matches(D[i],desc(q))) {
        pid = pageOf(tupleID(i))
        pagesToCheck = pagesToCheck ∪ pid
    }
}
for each pid in pagesToCheck {
    Buf = getPage(dataFile,pid)
    check tuples in Buf for answers
}
// where ...
#define matches(rdesc,qdesc)
               ((rdesc & qdesc) == qdesc)
```

Example SIMC Query

10/73

Consider the query and the example database:

Signature	**Deposit Record**
010000000001	(Perryridge,?,?,?)
100101001001	(Brighton,217,Green,750)
010011000111	(Perryridge,102,Hayes,400)
101001001001	(Downtown,101,Johnshon,512)
101100000011	(Mianus,215,Smith,700)
010101010101	(Clearview,117,Throggs,295)
100101010011	(Redwood,222,Lindsay,695)

Gives two matches: one true match, one *false match*.

SIMC Parameters

11/73

False match probablity p_F = likelihood of a false match

How to reduce likelihood of false matches?

- use different hash function for each attribute (h_i for A_i)
- increase descriptor size (m)
- choose k so that ≅ half of bits are set

Larger m means reading more descriptor data.

Having k too high $\Rightarrow$ increased overlapping.

Having k too low $\Rightarrow$ increased hash collisions.

... SIMC Parameters

12/73

How to determine "optimal" m and k?

1. start by choosing acceptable p_F
   (e.g. $p_F \leq 10^{-5}$ 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:

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

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

Query Cost for SIMC

13/73

Cost to answer *pmr* query: $Cost_{pmr} = b_D + b_q$

q = query page

- read r descriptors on b_D descriptor pages
- then read b_q data pages and check for matches

$b_D = ceil(\ r/c_D \ )$ and $c_D = floor(B/ceil(m/8))$

E.g. $m=64$, $B=8192$, $r=10^4$ $\Rightarrow$ $c_D = 1024$, $b_D=10$

b_q includes pages with r_q matching tuples and r_F false matches

Expected false matches = $r_F = (r - r_q).p_F \cong r.p_F$ if $r_q \ll r$

E.g. Worst $b_q = r_q+r_F$, Best $b_q = 1$, Avg $b_q = ceil(b(r_q+r_F)/r)$

Exercise 1: SIMC Query Cost

14/73

Consider a SIMC-indexed database with the following properties

- all pages are B = 8192 bytes
- tuple descriptors have m = 64 bits (= 8 bytes)
- total records r = 102,400, records/page c = 100
- false match probability p_F = 1/1000
- answer set has 1000 tuples from 100 pages
- 90% of false matches occur on data pages with true match
- 10% of false matches are distributed 1 per page

Calculate the total number of pages read in answering the query.

Page-level SIMC

Page –> signature –> codeword
The size of m is different

15/73

SIMC has one descriptor per tuple ... potentially inefficient.

Alternative approach: one descriptor for each data page.

Every attribute of every tuple in page contributes to descriptor.

Size of page descriptor (PD) (clearly larger than tuple descriptor):

- use above formulae but with $c.n$ "attributes"

E.g. $n = 4$, $c = 64$, $p_F = 10^{-3}$ $\Rightarrow$ $m \cong 3680bits \cong 460bytes$

Typically, pages are 1..8KB ⇒ 8..64 PD/page (c_{PD}).

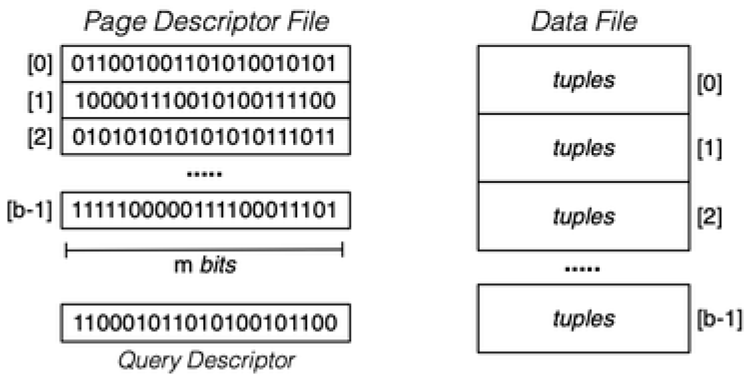
E.g. $m \cong 460$, $B = 8192$, $c_{PD} \cong 17$

... Page-level SIMC

16/73

There is a tupSig for each tuple, the pageSig is generated by doing or operation on all of them

File organisation for page-level superimposed codeword index



... Page-level SIMC

m(page) >= MaxTups

17/73

Algorithm for evaluating *pmr* query using page descriptors

```
pagesToCheck = {}
for each descriptor D[i] in signature file {
    if (matches(D[i],desc(q))) {
        pid = i
        pagesToCheck = pagesToCheck ∪ pid
    }
}
for each pid in pagesToCheck {
    Buf = getPage(dataFile,pid)
    check tuples in Buf for answers
}
```

Exercise 2: Page-level SIMC Query Cost

18/73

Consider a SIMC-indexed database with the following properties

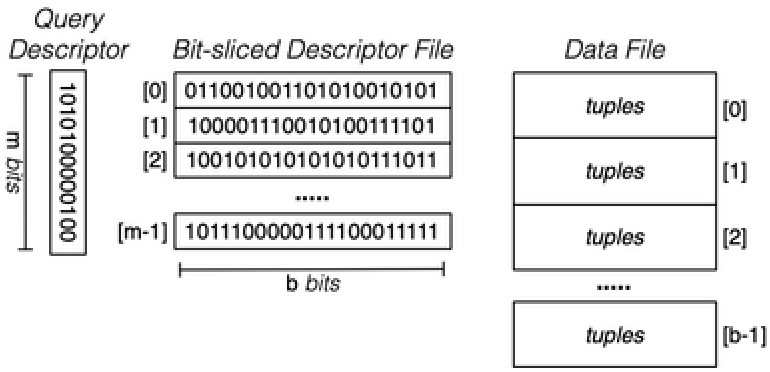
- all pages are $B = 8192$ bytes
- page descriptors have $m = 4096$ bits (= 512 bytes)
- total records $r = 102{,}400$,   records/page $c = 100$
- false match probability $p_F = 1/1000$
- answer set has 1000 tuples from 100 pages
- 90% of false matches occur on data pages with true match
- 10% of false matches are distributed 1 per page

Calculate the total number of pages read in answering the query.

Bit-sliced SIMC

19/73

Improvement: store *b* *m*-bit page descriptors as *m* *b*-bit "bit-slices"



... Bit-sliced SIMC

20/73

Bit –> page –> tuple –> codeword

Algorithm for evaluating *pmr* query using bit-sliced descriptors

```
matches = ~0  //all ones
for each bit i set to 1 in desc(q) {
   slice = fetch bit-slice i
   matches = matches & slice
}
for each bit i set to 1 in matches {
   fetch page i
   scan page for matching records
}
```

Effective because *desc(q)* typically has less than half bits set to 1

Exercise 3: Bit-sliced SIMC Query Cost

21/73

Consider a SIMC-indexed database with the following properties

- all pages are B = 8192 bytes
- r = 102,400, c = 100, b = 1024
- page descriptors have m = 4096 bits (= 512 bytes)
- bit-slices have b = 1024 bits (= 128 bytes)
- false match probability p_F = 1/1000
- query descriptor has k = 10 bits set to 1
- answer set has 1000 tuples from 100 pages
- 90% of false matches occur on data pages with true match
- 10% of false matches are distributed 1 per page

Calculate the total number of pages read in answering the query.

Assignment 2

Assignment 2

23/73

Aim: implement all variants of SIMC indexing

Implement individual relations and commands to work on them.

Each relation `R` consists of multiple files:

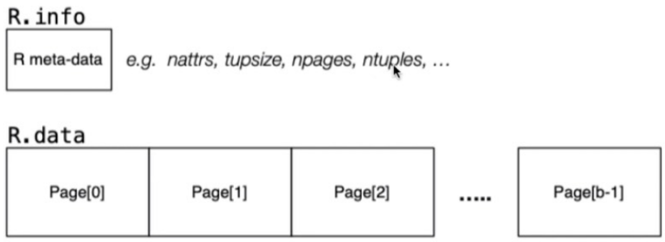
- **`R.info`** ... relation meta-data (e.g. # tuples)
- **`R.data`** ... data file containing pages of tuples
- **`R.tsig`** ... file containing tuple signatures
- **`R.psig`** ... file containing page-level signatures
- **`R.bsig`** ... file containing bit-sliced signatures

... Assignment 2

24/73

File structures for SIMC info + data files

[Diagram:Pics/assignments/file-structs-small.p



Tuples are all `tupSize` bytes long (based on # attributes)

Signatures are *m* bits long, rounded to *ceil(m/8) bytes*

... Assignment 2

25/73

More detailed file structures for SIMC data + signature files

... Assignment 2

26/73

We supply:

- *complete command programs to build and query relations*
- *partially-complete ADTs for operations needed by commands*

You complete the ADTs so that the commands work properly

- **`create`**, **`insert`**, **`select`** ... *build/query commands*
- **`gendata`**, **`stats`**, **`dump`** ... *utility commands*
- `x1`, `x2`, `x3` ... *commands for debugging ADTs*

... Assignment 2

27/73

`./create RelName #tuples #attrs 1/pF`

- *creates a new relation with prefix* `RelName`
- *uses* `#attrs` *to determine tuple size*
- *uses* `#tuples` *to determine b ⇒ length of bit-slices*

  (`#tuples` suggests maximum number of tuples to be stored)
- *uses* `pF` *to determine m and k for signatures*
- *creates files* `RelName.info`, `RelName.data`, *etc. etc.*
- *as supplied, data and signature files are empty after* `create`
- *when complete,* `RelName.bsig` *should contain all-zero bit-slices*

... Assignment 2

28/73

./gendata #tuples #attributes [startID] [seed]

- *generates `#tuples` tuples in a standard format, e.g.*

```
1234567,iuwhfkajewhkfjkwefbx,a3-101,a4-256,a5-013,...
```

- *first attribute is unique id*
- *second attribute is 20-char random string (most likely unique)*
- *rest are `aN-DDD` up to `#attributes`*
- *attributes `a3-DDD` to `aN-DDD` are not unique*
- *if no `startID` given, use `1000000`; if no `seed` given, use 0*

... Assignment 2

29/73

./insert [-v] RelName

- *insert tuples read from stdin into `Relname`*
- *updates all files: info, data, signature files*
- *tuples look like those generated from `gendata`*
- *typical usage*

```
./gendata #tuples #attrs | ./insert RelName
```

- *`-v` displays each tuple and PID of page where inserted*

... Assignment 2

30/73

./select RelName 'Query' SigType

- *finds all matches for pmr query in relation `RelName`*
- *queries are expressed using ? for unknown attributes, e.g.*

```
?,?,?,?,?         # matches all tuples
1234567,?,?,?,?   # matches single tuple with this ID
?,?,a3-101,?,?    # matches tuples with a3-101 as 3rd attr
?,?,a3-101,a4-200,a5-013
```

- *should enclose `Query` in single quotes (to avoid problems with zsh)*
- *prints one tuple per line (note: order of tuples is not important)*
- *prints page read statistics at end of output*

... Assignment 2

31/73

Method for ***./select RelName 'Query' SigType***

```
q = startQuery(r, qstr, type):
    check for valid query (e.g. #tuples)
    T = type of signature (t,p,b,x)
    Sig = build query signature of type T
    use Sig to determine list of interesting pages
    q->pages = bit-string of interesting pages
if (q == NULL) fatal error
scanAndDisplayMatchingTuples(Query q):
    foreach PID in q->pages {
        Buf = get page PID
        scan Buf for real matches and display each
    }
```

`Bits` ADT

32/73

ADT to represent arbitrary-length bit-strings

Bits newBits(int n);

- *create a new bit-string of length `n`*

Bool bitIsSet(Bits b, int i);

- *check whether bit `i` is set to 1 in bit-string `b`*

void setBit(Bits b, int i);

- *set the `i`th bit of bit-string `b` to 1*

void unsetBit(Bits b, int i);

- *set the* i^{th} *bit of bit-string* `b` *to 0*

Bit-strings of length `n` *are indexed from* `0` *(least sig) ..* `n-1` *(most sig)*

`Reln` ADT

33/73

ADT to represent relations (where `RelnRep *Reln`)

`Status newRelation(name, nattrs,pF,tk,tm,pm,bm)`

- *make all files for relation* `name`*, based on parameters*

`Reln openRelation(char *name)`

- *create a* `RelnRep`*, populate it and open all files*

`void closeRelation(Reln r)`

- *close all files and clean up* `RelnRep` *data structure*

`PageID addToRelation(Reln r, Tuple t)`

- *insert tuple* `t` *into open relation* `r`*; return PID where inserted*

`Query` ADT

34/73

ADT to represent queries (where `QueryRep *Query`)

`Query startQuery(Reln r, char *qry, char sigType)`

- *set up* `QueryRep` *for query* `qry` *for specific type of signature*

`void scanAndDisplayMatchingTuples(Query q)`

- *evaluate the query* `q` *and display result tuples, one per line*

`void queryStats(Query q)`

- *print statistics from the* `QueryRep`*, typically after query finishes*

`void closeQuery(Query)`

- *clean up* `QueryRep` *data structure (i.e.* `free`*)*

Signature ADTs

35/73

Three different types of signature: tuple, page, bit-slice

Each has its own ADT, but all ADTs are similar

`Bits makeXSig(Reln r, Tuple t)`

- *build a signature of type* `X` *for tuple* `t`

`void findPagesUsingXSigs(Query q)`

- *uses* `Xsig` *to build bit-string of potentially-matching pages*
- *result is stored in* `q->pages` *(of type* `Bits`*)*

Note: don't need `makeBitSliceSig()`*; it uses page signatures*

`Psig` ADT

36/73

ADT to represent page signatures

`Bsig` ADT

37/73

ADT to represent bit-sliced signatures

... `Bsig` ADT

38/73

What to do now?

- *review the notes on superimposed codewords*
- *read the spec carefully*
- *read the code for the commands (to see how they use the ADTs)*
- *read the ADT `*.h` files; read the ADT `*.c` files*
- *write and test your code (suggest: tsig, then psig, then bsig)*

Testing script available next week.

Don't wait. It's easy to devise your own tests.

N-d Tree Indexes

Multi-dimensional Tree Indexes

40/73

Over the last 20 years, from a range of problem areas

- *different multi-d tree index schemes have been proposed*
- *varying primarily in how they partition tuple-space*

Consider three popular schemes: kd-trees, Quad-trees, R-trees.

Example data for multi-d trees is based on the following relation:

```
create table Rel (
    X char(1) check (X between 'a' and 'z'),
    Y integer check (Y between 0 and 9)
);
```

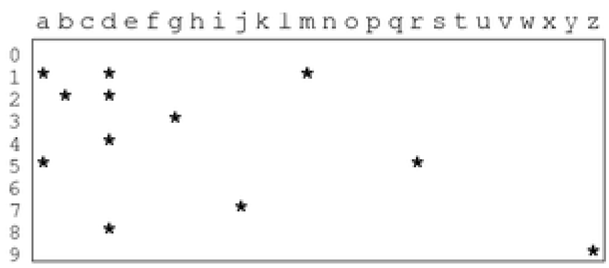
... Multi-dimensional Tree Indexes

41/73

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)
```

The tuple-space for the above tuples:

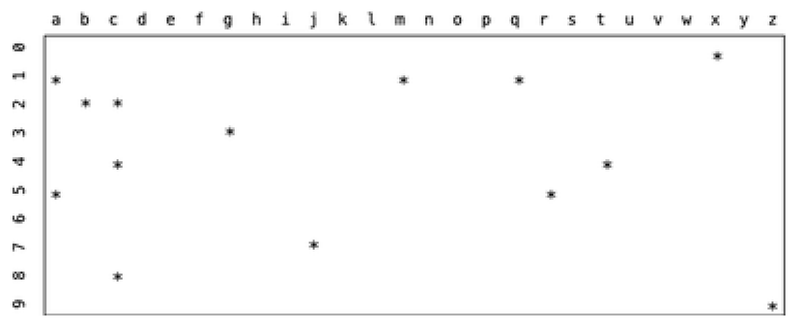


Exercise 4: Query Types and Tuple Space

42/73

Which part of the tuple-space does each query represent?

```
Q1: select * from Rel where X = 'c' and Y = 4
Q2: select * from Rel where 'j' < X ≤ 'r'
Q3: select * from Rel where X > 'm' and Y > 4
Q4: select * from Rel where 'k' ≤ X ≤ 'p' and 3 ≤ Y ≤ 6
```



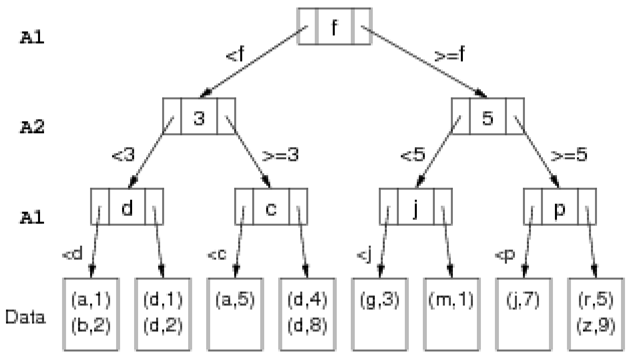
Q1 ... Q2 ... Q3 ... Q4

43/73

kd-Trees

kd-trees are multi-way search trees where

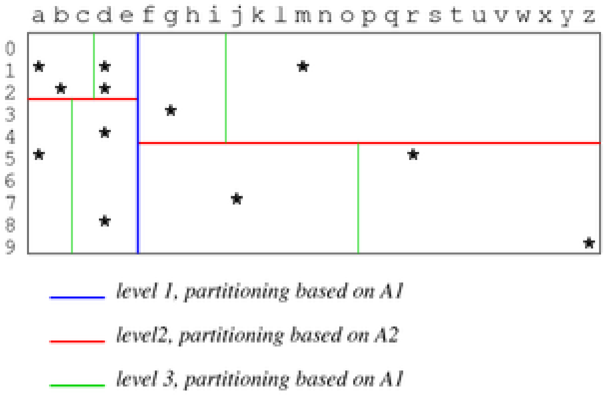
- *each level of the tree partitions on a different attribute*
- *each node contains n-1 key values, pointers to n subtrees*



... kd-Trees

44/73

How this tree partitions the tuple space:



Searching in kd-Trees

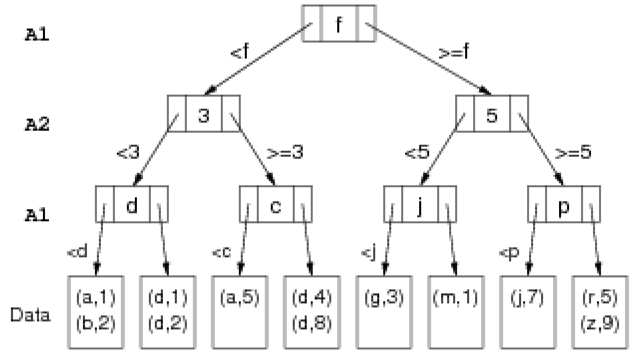
45/73

```
// Started by Search(Q, R, 0, kdTreeRoot)
Search(Query Q, Relation R, Level L, Node N)
{
   if (isDataPage(N)) {
      Buf = getPage(fileOf(R),idOf(N))
      check Buf for matching tuples
   } else {
      a = attrLev[L]
      if (!hasValue(Q,a))
         nextNodes = all children of N
      else {
         val = getAttr(Q,a)
         nextNodes = find(N,Q,a,val)
      }
      for each C in nextNodes
         Search(Q, R, L+1, C)
}  }
```

Exercise 5: Searching in kd-Trees

46/73

Using the following kd-tree index



Answer the queries `(m,1), (a,?), (?,1), (?,?)`

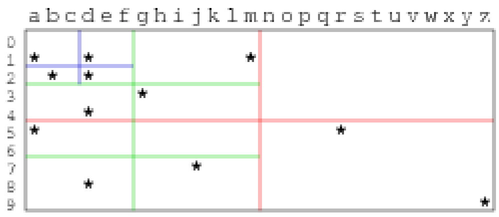
Quad Trees

47/73

Quad trees use regular, disjoint partitioning of tuple space.

- *for 2d, partition space into quadrants (NW, NE, SW, SE)*
- *each quadrant can be further subdivided into four, etc.*

Example:



... Quad Trees

48/73

Basis for the partitioning:

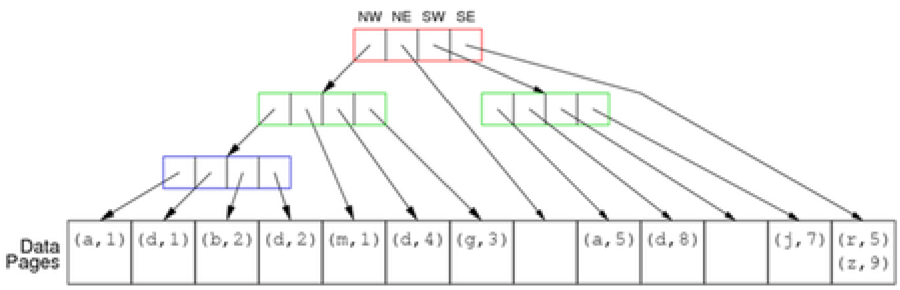
- *a quadrant that has no sub-partitions is a leaf quadrant*
- *each leaf quadrant maps to a single data page*
- *subdivide until points in each quadrant fit into one data page*
- *ideal: same number of points in each leaf quadrant (balanced)*
- *point density varies over space*
  ⇒ different regions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

Note: effective for d≤5, ok for 6≤d≤10, ineffective for d>10

... Quad Trees

49/73

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

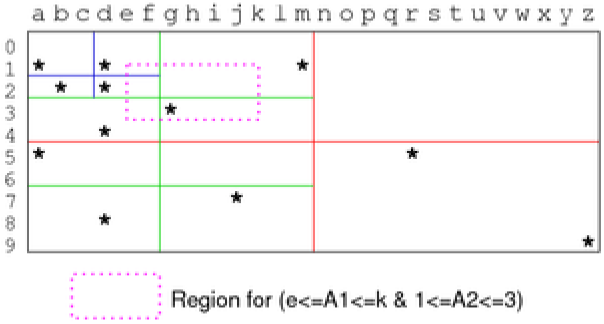


In this and following examples, we give coords of top-left,bottom-right of a region

Searching in Quad-tree

50/73

Space query example:

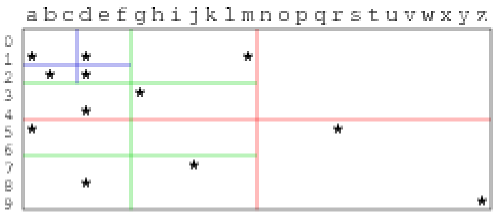


Need to traverse: red(NW), green(NW,NE,SW,SE), blue(NE,SE).

Exercise 6: Searching in Quad-trees

51/73

Using the following quad-tree index



Answer the queries `(m,1), (a,?), (?,1), (?,?)`

R-Trees

52/73

R-trees use a flexible, overlapping partitioning of tuple space.

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

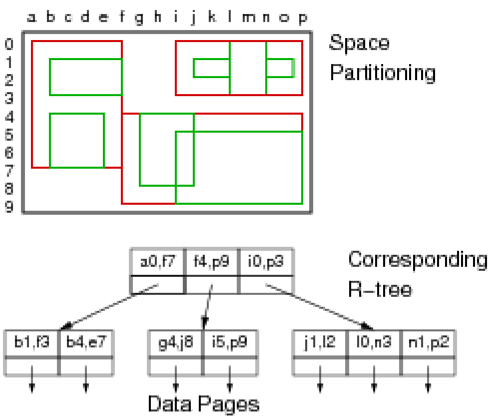
Overlap and partial cover means:

- *can optimize space partitioning wrt data distribution*
- *so that there are similar numbers of points in each region*

Aim: height-balanced, partly-full index pages *(cf. B-tree)*

... R-Trees

53/73



Insertion into R-tree

54/73

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 its boundaries so that it does contain R*
- *if several children contain R, choose one and proceed to child*
- *repeat above containment search in children of 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.

Query with R-trees

55/73

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 region contains P*

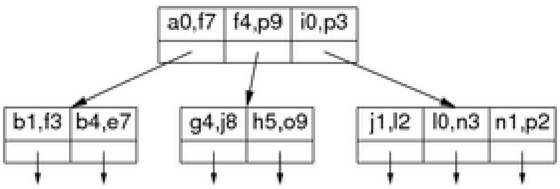
Space (region) queries are handled in a similar way

- *we traverse down any path that intersects the query region*

Exercise 7: Query with R-trees

56/73

Using the following R-tree:



Show how the following queries would be answered:

```
Q1: select * from Rel where X='a' and Y=4
Q2: select * from Rel where X='i' and Y=6
Q3: select * from Rel where 'c'≤X≤'j' and Y=5
Q4: select * from Rel where X='c'
```

Note: can view unknown value `X=?` as range min(`X`) ≤ `X` ≤ max(`X`)

Multi-d Trees in PostgreSQL

57/73

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

Superseded by GiST = Generalized Search Trees

GiST indexes parameterise: data type, searching, splitting

- *via seven user-defined functions (e.g. `picksplit()`)*

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*

Details: src/backend/access/gist

Costs of Search in Multi-d Trees

58/73

Difficult to determine cost precisely.

Best case: pmr query where all attributes have known values

- *in kd-trees and quad-trees, follow single tree path*
- *cost is equal to depth D of tree*
- *in R-trees, may follow several paths (overlapping partitions)*

Typical case: some attributes are unknown or defined by range

- *need to visit multiple sub-trees*
- *how many depends on: range, choice-points in tree nodes*

Similarity-based Selection

Relational vs Similarity Selection

60/73

Relational selection is based on a boolean condition C

- *evaluate C for each tuple t   (or a likely subset of tuples)*
- *if C(t) is true, add t to result set*
- *if C(t) is false, t is not part of solution*
- *result is a set of tuples { t_1, t_2, ..., t_n } all of which satisfy C*

Uses for relational selection:

- *precise matching on structured data*
- *using individual attributes with known, exact values*

... Relational vs Similarity Selection

61/73

Similarity selection is used in contexts where

- *cannot define a precise matching condition*
- *but can identify a notion of (S similar to T)*
- *requires a measure d of "distance" between tuples*
- *d=0 is an exact match, d>0 is less accurate match*
- *result is a list of pairs [(t_1,d_1), (t_2,d_2), ..., (t_n,d_n)]   (ordered by d_i)*

Uses for similarity matching:

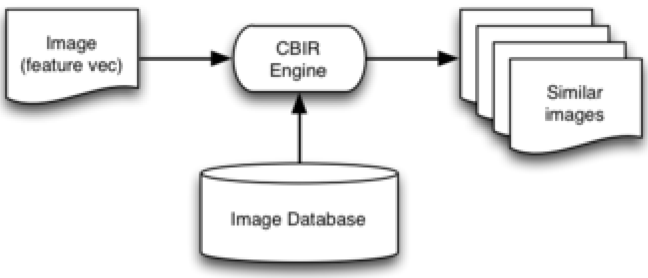
- *text or multimedia (image/music) retrieval*
- *ranked queries in conventional databases*

Example: Content-based Image Retrieval

62/73

User supplies a description or sample of desired image.

System returns a ranked list of similar images from database.



... Example: Content-based Image Retrieval

63/73

At the SQL level, this might appear as ...

```
// relational matching
create view Sunset as
select image from MyPhotos
where  title = 'Pittwater Sunset'
       and taken = '2012-01-01';
// similarity matching with threshold
create view SimilarSunsets as
select title, image
from   MyPhotos
where  (image ~~ (select * from Sunset)) < 0.05
order  by (image ~~ (select * from Sunset));
```

where (imaginary) `~~` *operator measures how "alike" images are*

Similarity-based Retrieval

64/73

Database contains media objects, but also tuples, e.g.

- `id` *to uniquely identify object   (e.g. PostgreSQL* `oid`*)*
- *metadata   (e.g. artist, title, genre, date taken, ...)*
- `value` *of object itself   (e.g. PostgreSQL BLOB or* `bytea`*)*

BLOB = Binary Large OBject

- *BLOB stored in separate file; tuple contains reference (cf. TOAST)*
- *BLOBs are typically MB in size (1MB..2GB)*

... Similarity-based Retrieval

65/73

Similarity-based retrieval requires a distance measure

- *$dist(x,y) \in 0..1$, $dist(x,x) = 0$, $dist(x,y) = dist(y,x)$*

where x and y are two objects (in the database)

Note: distance calculation often requires substantial computational effort

How to restrict solution set to only the "most similar" objects:

- *threshold d_{max} (only objects t such that $dist(t,q) \le d_{max}$)*
- *count k (k closest objects (k nearest neighbours))*

BUT both above methods require knowing distance between query object and all objects in DB

... Similarity-based Retrieval

66/73

Naive approach to similarity-based retrieval

```
q = ...     // query object
dmax = ... // dmax > 0  =>  using threshold
knn = ...  // knn > 0   =>  using nearest-neighbours
Dists = [] // empty list
foreach tuple t in R {
    d = dist(t.val, q)
    insert (t.oid,d) into Dists  // sorted on d
}
n = 0;  Results = []
foreach (i,d) in Dists {
    if (dmax > 0 && d > dmax) break;
    if (knn > 0 && ++n > knn) break;
    insert (i,d) into Results  // sorted on d
}
return Results;
```

Cost = fetch all r objects + compute distance() for each

... Similarity-based Retrieval

67/73

For some applications, Cost(dist(x,y)) is comparable to T_r

⇒ computing `dist(t.val,q)` for every tuple `t` is infeasible.

To improve this ...

- *compute feature vector to capture "critical" object properties*
- *store feature vectors "in parallel" with objects (cf. signatures)*
- *compute distance using feature vectors (not objects)*

i.e. replace dist(t,q) by dist'(vec(t),vec(q)) in previous algorithm.

Further optimisation: dimension-reduction to make vectors smaller

---

... Similarity-based Retrieval

68/73

Feature vectors ...

- *often use multiple features, concatenated into single vector*
- *represent points in a very high-dimensional (vh-dim) space*

Content of feature vectors depends on application ...

- *image ... colour histogram (e.g. 100's of values/dimensions)*
- *music ... loudness/pitch/tone (e.g. 100's of values/dimensions)*
- *text ... term frequencies (e.g. 1000's of values/dimensions)*

Query: feature vector representing one point in vh-dim space

Answer: list of objects "near to" query object in this space

... Similarity-based Retrieval

69/73

***Inputs** to content-based similarity-retrieval:*

- *a database of r objects ($obj_1, obj_2, ..., obj_r$) plus associated ...*
- *r x n-dimensional feature vectors ($v_{obj_1}, v_{obj_2}, ..., v_{obj_r}$)*
- *a query image q with associated n-dimensional vector (v_q)*
- *a distance measure $D(v_i,v_j) : [0..1)$ ($D=0 \rightarrow v_i=v_j$)*

***Outputs** from content-based similarity-retrieval:*

- *a list of the k nearest objects in the database $[a_1, a_2, ... a_k]$*
- *ordered by distance $D(v_{a_1},v_q) \le D(v_{a_2},v_q) \le ... \le D(v_{a_k},v_q)$*

Approaches to *k*NN Retrieval

70/73

Partition-based

- *use auxiliary data structure to identify candidates*
- *space/data-partitioning methods: e.g. k-d-B-tree, R-tree, ...*
- *unfortunately, such methods "fail" when #dims > 10..20*
- *absolute upper bound on d before linear scan is best d = 610*

Approximation-based

- *use approximating data structure to identify candidates*
- *signatures: VA-files*
- *projections: iDistance, LSH, MedRank, CurveIX, Pyramid*

... Approaches to *k*NN Retrieval

71/73

Above approaches try to reduce number of objects considered.

- *cf. indexes in relational databases*

Other optimisations to make kNN retrieval faster

- *reduce I/O by reducing size of vectors (compression, d-reduction)*
- *reduce I/O by placing "similar" records together (clustering)*
- *reduce I/O by remembering previous pages (caching)*
- *reduce cpu by making distance computation faster*

Similarity Retrieval in PostgreSQL

72/73

PostgreSQL has always supported simple "similarity" on strings

```
-- for most SQL implementations
select * from Students where name like '%oo%';
-- and PostgreSQL-specific
select * from Students where name ~ '[Ss]mit';
```

Also provides support for ranked similarity on `text` values

- *using **`tsvector`** data type (stemmed, stopped feature vector for `text`)*
- *using **`tsquery`** data type (stemmed, stopped feature vector for strings)*
- *using **`@@`** similarity operator*

... Similarity Retrieval in PostgreSQL

73/73

Example of PostgreSQL text retrieval:

```
create table Docs
   ( id integer, title text, body text );
// add column to hold document feature vectors
alter table Docs add column features tsvector;
update Docs set features =
   to_tsvector('english', title||' '||body);
// ask query and get results in ranked order
select title, ts_rank(d.features, query) as rank
from   Docs d,
       to_tsquery('potter|(roger&rabbit)') as query
where  query @@ d.features
order  by rank desc
limit  10;
```

For more details, see PostgreSQL documentation, Chapter 12.

Produced: 31 Mar 2020