

Similarity-based Selection

Similarity Selection

1/65

Relational selection is based on a boolean condition C

- evaluate C for each tuple t
- 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 $\{t_1, t_2, \dots, t_n\}$ all of which satisfy C

Uses for relational selection:

- precise matching on structured data

... Similarity Selection

2/65

Similarity selection is used in contexts where

- cannot define a precise matching condition
- *can* define 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 $[(t_1, d_1), (t_2, d_2), \dots, (t_n, d_n)]$ (ordered by d_i)

Uses for similarity matching:

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

Similarity-based Retrieval

3/65

Similarity-based retrieval typically works as follows:

- query is given as a *query object* q (e.g. sample image)
- system finds objects that are *like* q (i.e. small distance)

The system can measure distance between any object and q ...

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

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

... Similarity-based Retrieval

4/65

Tuple structure for storing such data typically contains

- *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*)

Properties of typical distance functions (on objects x, y, z)

- $dist(x, y) \geq 0$, $dist(x, x) = 0$, $dist(x, y) = dist(y, x)$
- $dist(x, z) < dist(x, y) + dist(y, z)$ (triangle inequality)
- often require substantial computational effort

... Similarity-based Retrieval

5/65

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 = read all r feature vectors + compute $distance()$ for each

... Similarity-based Retrieval

6/65

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

\Rightarrow computing $dist(t.val, q)$ for every tuple t is infeasible.

To improve this aspect:

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

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

Further optimisation: dimension-reduction to make vectors smaller

... Similarity-based Retrieval

7/65

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)

Typically use multiple features, concatenated into single vector.

Feature vectors represent points in a very high-dimensional space.

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

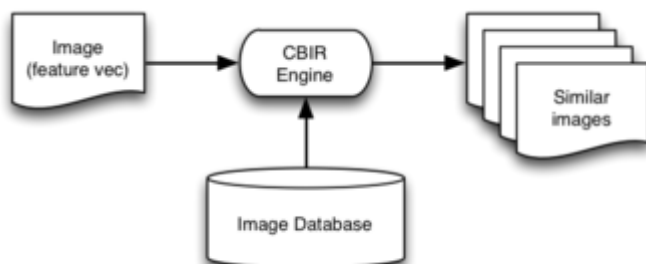
Answer: list of objects "near to" query object in this space.

Example: Content-based Image Retrieval

8/65

User supplies a description or sample of desired image (features).

System returns a ranked list of "matching" images from database.



... Example: Content-based Image Retrieval

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

```
create view Sunset as
select image from MyPhotos
where title = 'Pittwater Sunset'
      and taken = '2009-01-01';

create view SimilarSunsets as
select title, image
from MyPhotos
where (image ~ (select * from Sunset)) < 0.05
order by (image ~ (select * from Sunset));
```

where the \sim operator measures distance between images.

... Example: Content-based Image Retrieval

10/65

Implementing content-based retrieval requires ...

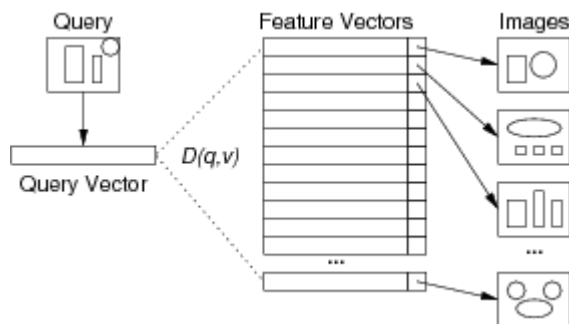
- a collection of "pertinent" image features
 - e.g. colour, texture, shape, keywords, ...
- some way of describing/representing image features
 - typically via a vector of numeric values
- a distance/similarity measure based on features
 - e.g. Euclidean distance between two vectors

$$\text{dist}(x,y) = \sqrt{(x_1-y_1)^2 + (x_2-y_2)^2 + \dots (x_n-y_n)^2}$$

... Example: Content-based Image Retrieval

11/65

Data structures for naive similarity-retrieval:



... Example: Content-based Image Retrieval

12/65

Insertion of an image into the database:

- use image processing algorithms to compute feature vector
- insert image (either into file system or as a BLOB in the DBMS)
- insert tuple (img, vec) associating image and feature vector

Insertion cost:

- image processing (relatively expensive, $\cong T_I$)
- copy image in file sys OR insertion of image as BLOB
- insertion of (img, vec) tuple $(1_I + 1_W)$

... Example: Content-based Image Retrieval

13/65

Inputs to content-based similarity-retrieval:

- a database of r objects $(obj_1, obj_2, \dots, obj_r)$ plus associated ...

- $r \times n$ -dimensional feature vectors $(v_{obj_1}, v_{obj_2}, \dots, v_{obj_r})$
- 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 $[a_1, a_2, \dots, a_k]$
- ordered by distance $D(v_{a_1}, v_q) \leq D(v_{a_2}, v_q) \leq \dots \leq D(v_{a_k}, v_q)$

Approaches to kNN Retrieval

14/65

Partition-based

- use auxiliary data structure to identify candidates
- space-partitioning methods: Grid file, k-d-B-tree, quadtree
- data-partitioning methods: R-tree, X-tree, SS-tree, TV-tree, ...
- unfortunately, such methods "fail" when #dims > 10..20

Approximation-based

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

... Approaches to kNN Retrieval

15/65

Above approaches mostly try to reduce number of objects considered.

Other optimisations to make kNN retrieval faster

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

VA Files

VA (Signature) Files

17/65

Vector Approximation (VA) file developed by Weber and Blott.

- targetted at very high-dimensional feature vectors
- abandons the idea of search complexity better than $O(n)$

Why give up on sub-linear complexity?

- analysis shows that *all* methods degenerate to this as d increases
- absolute upper bound on d before linear scan is best $d = 610$
- in practice, observe that most methods degenerate for $10 \leq d \leq 40$

Note: d = number of dimensions

... VA (Signature) Files

18/65

Uses a signature file "parallel" to the main feature vector file



... VA (Signature) Files

19/65

VA signatures have properties:

- (much) more compact than feature vectors
- provide approximation to information in vectors

Approach to querying:

- perform filtering by fast scan of signatures, then
- compute expensive D only on (hopefully) small set of candidates

VA-File Signatures

20/65

Computing VA signatures:

- partition space into small number of regions on each dimension
- use region values in object signatures (coarse-grained view of data)

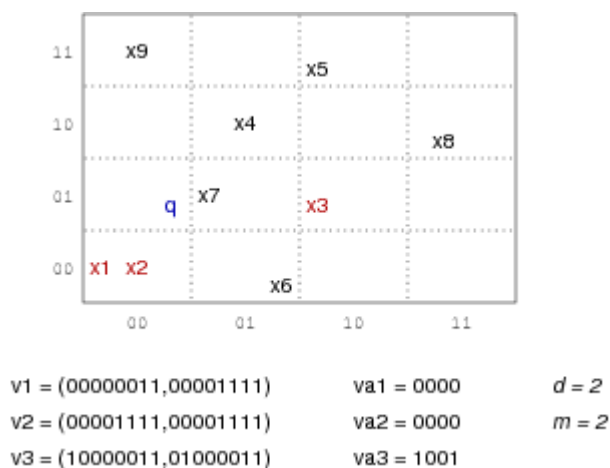
Implemented by taking m high-order bits from each feature vector element.

Consider a 3- d RGB feature vector, with 0..255 for each RGB value:

- feature vector: $v = (255, 128, 0) = (11111111, 10000000, 00000000)$
- partition each dimension into 4 regions ($\Rightarrow m=2$ bits per d)
- VA signature for this vector: $va(v) = (11, 10, 00)$

... VA-File Signatures

21/65



Insertion with VA-Files

22/65

Given: a feature vector v_{obj} for a new object.

Insertion is extremely simple.

```
sig = signature( $v_{obj}$ )
append  $v_{obj}$  to feature vector file
```

append *sig* to signature file

Storage overhead determined by *m* (e.g. 3/32).

Query with VA-Files

23/65

Input: query vector *vq*

```

results = []; maxD = infinity;
for (i = 0; i < r; i++) {
    // fast distance calculation
    dist = minDistance(region[vq[i]], vq)
    if (#results < k || dist < maxD) {
        dist = distance(v[i], vq)
        if (#results < k || dist < maxD) {
            insert (tid[i], dist) into results
            // sorted(results) && length(results) <= k
            maxD = largest distance in results
        }
    }
}

```

Result is a list of *tids* of *k* database objects nearest to query.

... Query with VA-Files

24/65

VA signatures allow fast elimination of objects by region

- if nearest point in region containing object is further than *maxD*, ignore object

Given query vector *q*, data vector *v*, can quickly compute:

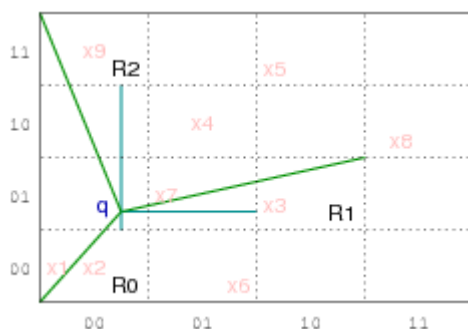
- $lower(q, v)$ = distance between *q* and nearest point in region $R[v]$
- $upper(q, v)$ = distance between *q* and furthest point in region $R[v]$

Thus, instead of *r* expensive *D* computations, we require

- r* cheap computations of $(lower(q, v), upper(q, v))$
- small number of real *distance* computations
(how many are actually required depends on effectiveness of filtering)

... Query with VA-Files

25/65



Marks on each dimension *i* are stored as array $R_i[0], R_i[1], \dots, R_i[2^m]$.

... Query with VA-Files

26/65

Distance bound formula:

$$lower(q, v) = \sqrt{\sum_{i=1}^d low_i^2}$$

where

$$low_i = \begin{cases} q_i - R_i[v_i+1], & \text{if } v_i < q_i \\ 0, & \text{if } v_i = q_i \end{cases}$$

$$= R_i[v_i] - q_i,$$

$$\text{if } v_i > q_i$$

$$\text{upper}(q, v) = \sqrt{\sum_{i=1}^d \text{upp}_i^2}$$

where

$$\begin{aligned} \text{upp}_i &= q_i - R_i[v_i], & \text{if } v_i < q_i \\ &= \max(q_i - R_i[v_i], R_i[v_i+1] - q_i), & \text{if } v_i = q_i \\ &= R_i[v_i+1] - q_i, & \text{if } v_i > q_i \end{aligned}$$

Cost of VA File Searching

27/65

Performing VA search requires:

- read r VA signatures
- compute $\text{lower}(q, v_j)$ for each of r signatures
(could potentially cache region-based distance results in look-up table)
- compute real distance, D , rf times, where f is filtering factor
(expect an f value somewhere near 0.001 under ideal conditions)

A problem with VA files:

- works best with uniform data distribution (many image DBs are not)
- observed filtering levels are 0.01-0.1 rather than 0.001

Can fix this problem using uneven region sizes (but needs more storage)

... Cost of VA File Searching

28/65

Example: $d=200, m=3, r=10^6, B=4096, T_D \approx 0.001T_r$

- size of tuples $(tid, vec) = 4 \text{ bytes} + 400 \text{ bytes} = 404 \text{ bytes}$
- number of tuples per page $\lfloor 4096/404 \rfloor = 10$
- total number of feature vector pages $= \lceil 10^6/10 \rceil = 100000$

Cost (without VA file) $= 100000T_r + 10^6T_D \approx 101000T_r$

- size of VA signature $= 200 \times 3 \text{ bits} = 75 \text{ bytes}$
- number sigs in VA file page $= \lfloor 4096/75 \rfloor = 54$
- total number of VA file pages $= \lceil 10^6/54 \rceil = 18518$
- with $f=0.001$, fetch only 1000 feature vectors

Cost (with VA file) $= 18518T_r + 1000T_r + 1000T_D \approx 19519T_r$

Improved VA Search Algorithm

29/65

An improved query algorithm that guarantees minimal D computations:

```

vaq = vsignature(vq)
results = []; maxD = infinity; pending = [];
for (i = 0; i < r; i++) {
    lowD = lower(vq, region[va[i]])
    uppD = upper(vq, region[va[i]])
    if (#results < k || dist < uppD) {
        sortedInsert(i, uppD) into results
        heapInsert(i, lowD) into pending
    }
}
results = []; heapRemove(i, lowD) from pending
while (lowD < maxD) {
    dist = distance(v[i], vq)
    if (#results < k || dist < maxD) {
        sortedInsert(i, dist) into results
        // sorted(results) && length(results) <= k
    }
}

```

```

    maxD = largest distance in results
  }
  heapRemove (i,lowD) from pending
}

```

Curve-based Similarity Search

Curve-based Searching

31/65

The strategy when using curve-based indexing is as follows:

- dealing with many dimensions ($d > 10$) is too difficult
- so, project the feature space onto one dimension (line/curve)
- this gives a linear ordering of the objects
- can then perform search over this linear ordering
- using existing efficient 1d access methods (e.g. B-trees)

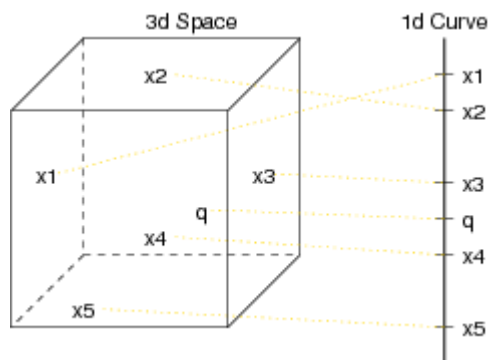
Indexing based on curves needs to ...

- minimise information loss to reduce missing answers
- at the same time, maximise filtering effectiveness

... Curve-based Searching

32/65

Mapping from 3-d to 1-d

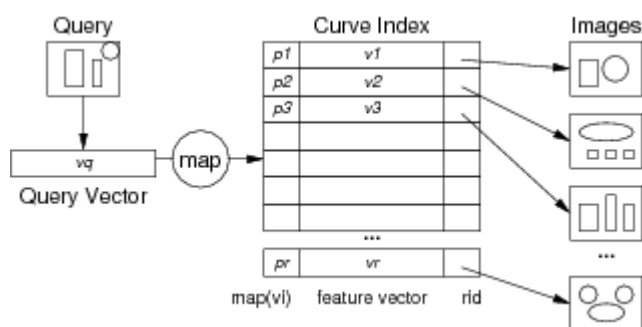


Note: impossible to preserve all NN-in-space as NN-on-curve.

Curve Index File Organisation

33/65

Data structures for curve-based searching:



Insertion with Curve Index

34/65

For each image inserted into the database:

- determine a feature vector v for the image
- determine a tid for the image (file name?)

- map the feature vector onto the curve $p = \text{map}(v)$
- p is the primary key for the curve index file
- insert (p, v, tid) into the curve index file using e.g. B-tree

Searching with Curve Index

35/65

Overview of curve-based searching algorithm:

Input: query feature vector v_q

```

p = map(vq)
cur1 = cur2 = lookup(p)
while (not enough answers) {
    (pt, vec, tid) = next(cur1)
    remember tid if D(vec, vq) small enough
    (pt, vec, tid) = prev(cur2)
    remember tid if D(vec, vq) small enough
}

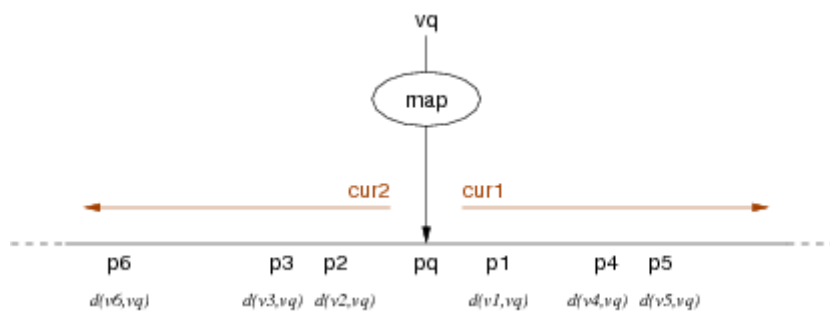
```

Output: collected *tids* of similar objects

... Searching with Curve Index

36/65

How curve is scanned to find potential near-neighbours:



... Searching with Curve Index

37/65

What kind of curves are required to make this approach viable?

- must pass through every data point exactly once
- must pass through every "point" in the underlying space exactly once

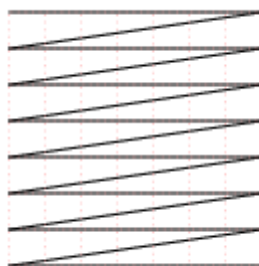
Possible candidate curves:

- space-filling curves (e.g. Hilbert curve, Peano curve, Gray ordering)

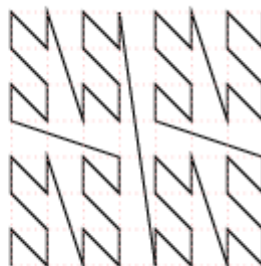
Experiments have shown that the Hilbert curve is the most suitable for data access.

2d Space-filling Curves

38/65

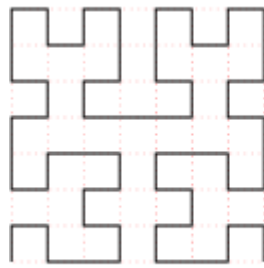


Row-wise enumeration

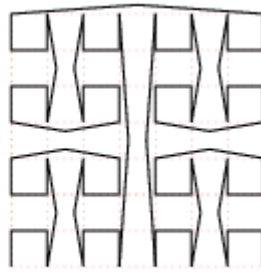


Peano curve

... 2d Space-filling Curves



Hilbert curve



Gray ordering

The CurveIx Scheme

40/65

What we would like from the above curve mapping:

- object close to the query on the curve \Rightarrow close in feature space
- object close to the query in feature space \Rightarrow close on the curve

With a single Hilbert curve (although not some other curves)

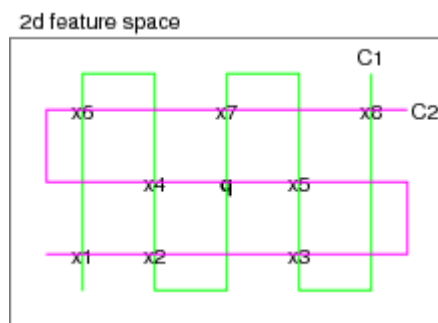
- $map(v) \cong map(v_q) \Rightarrow D(v, v_q) \cong 0$ is generally true
(in other words, we usually find similar objects near the query on the curve)
- $D(v, v_q) \cong 0 \Rightarrow map(v) \cong map(v_q)$ is sometimes false
(in other words, we sometimes fail to find objects that are similar to the query)

One curve is not enough, so use several "complementary" curves.

... The CurveIx Scheme

41/65

Example curveix scheme with 2 curves on a 2d space:



... The CurveIx Scheme

42/65

The basic idea:

- project d -dimensional vector onto 1-d space-filling curve
- collect set of curve-neighbours (should contain some kNN)
- repeat above for several curves with different paths through space
- union of neighbour sets from all curves gives candidates
- retrieve and compute distance only for candidates

We want candidate set to contain most/all of kNN ...

- how many curves do we need?
- how many neighbours do we examine on each curve?

Curve Mapping Functions

43/65

Each C_i has the following properties:

- maps d -dimensional vectors onto points on a line i.e. $[0,1)^d \rightarrow [0,1)$
- mapping is one-to-one and defines an order on $[0,1)^d$
- maps onto a space-filling Hilbert curve

Points within a small region of the curve are *likely* to have been mapped from vectors that are close in d -space.

How to generate multiple C_i s from a single vector:

- reorder elements in vector, translate individual elements

Data Structures for Curvelx

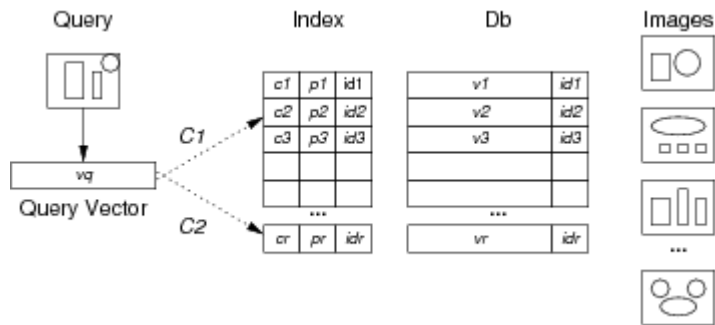
44/65

Derived from data structures for the QBIC system:

- C_i a mapping function for each of the m different space-filling curves ($i=1..m$)
- Db a database of r d -dimensional feature vectors; each entry is a pair (imageId, vector) where imageId forms a primary key
- $Index$ a database of rm curve points; each point is represented by a tuple (curveId, point, imageId); the pair (curveId, point) forms a primary key with an ordering, where $point = C_{curveId}(v_{imageId})$
- v_q feature vector for query q

... Data Structures for Curvelx

45/65



Database Construction

46/65

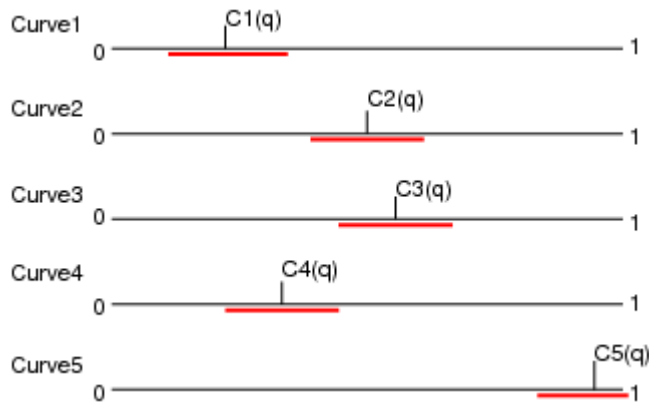
For each image obj , insertion requires:

```

id = identifier for obj
for (i in 1..m) {
    p =  $C_i(v_{obj})$ 
    insert (i, p, id) into Index
}
insert (id,  $v_{obj}$ ) into Db
  
```

Example: Search with 5 Curves

47/65



Finding k -NN in Curvelx

48/65

Given: v_q , $C_1 \dots C_m$, $Index$, Db

```

for (i in 1..m) {
   $p = C_i(v_q)$ 
  lookup (i,p) in Index
  fetch (i,p1,j)
  while (p1 "close to" p on curve i) {
    collect j as candidate
    fetch next (i,p1,j)
  }
}
for each candidate j {
  lookup j in Db
  fetch (j,v_j)
   $d = D(v_j, v_q)$ 
  include j in k-NN if d small enough
}

```

Curvelx vs. Linear Scan

49/65

Linear scan: Cost = read all N vectors + compute D for each

Curvelx: Cost = m B-tree lookups + compute D fN times

Some observations ...

- we can't afford to have too many curves ($m < 10?$)
- Curvelx has to do effective filtering ($f \ll 1$)

Difficult to formally analyse further, so we implemented the system to see how it performed ...

C_i Values as Keys

50/65

One problem raised early in the implementation:

- the Hilbert numbers produced by C_i could be expanded to arbitrary precision in order to distinguish between $C_i(v_1)$ and $C_i(v_2)$

To use such values as database keys, we need them as fixed precision, so we limit expansion (to 4 levels).

Problems:

- gives keys that are 96 bytes long (producing very large Index files)
- different v_j can be mapped to same point (candidate set size artificially inflated)

Solution:

- use a "prefix" index structure (something like a trie)

Since Curvelx does not guarantee to deliver the k -NN among its candidates, we set an "acceptable" accuracy level of 90%.

In other words, Curvelx must deliver $0.9 k$ nearest-neighbours to be considered useful.

The initial experiments aimed to answer the questions:

- How many curves are needed to achieve 90% accuracy?
- How many curve-neighbours do we need to examine?
- Can all of this be done reasonably efficiently?

Experiments

52/65

- Measures for accuracy:
- *Acc1* average top-10 entries in Curvelx top-10
 - *Acc2* how frequently Curvelx gives 10 out of 10

- Measures for efficiency:
- *Size* size of *Db* file + *Index* file
 - *Dist* number of distance calculations required
 - *IO* total amount of I/O performed

... Experiments

53/65

- To determine how these measures vary:
- built databases of size 5K, 10K, 15K, 20K (supersets)
 - for each database, ran 25 query "benchmark" set
 - for each query, ran for 3,5,10,20,30,40 curve-neighbours (but because of curve-mapping problem, only got 20,30,40)
 - for each query, ran for 20,40,60,80,100 curves

Also implemented a linear scan version for comparison and to collect the exact answer sets.

Sample Comparison

54/65

Curvelx Indexing	Linear Scan
QUERY: img00102	QUERY: img00102
0.000000 img00102	0.000000 img00102
0.005571 img00713	0.005571 img00713
0.008826 img00268	0.008826 img00268
0.011413 img05054	0.011413 img05054
0.011811 img00631	0.011811 img00631
0.014259 img04042	0.014259 img04042
0.027598 img00203	0.027598 img00203
0.037471 img00287	0.037471 img00287
0.063639 img00244	0.063639 img00244
0.067583 img00306	0.067583 img00306
# dist calcs: 524	# dist calcs: 20000
1.67user 0.23system ...	1.93user 1.12system ...

Experimental Results

55/65

For fixed database (20K), effect of varying *Range*, *Ncurves*

#Curves	Range	Acc1	Acc2	#Dist
---------	-------	------	------	-------

20	20	6.72	0.20	426
20	30	7.28	0.28	695
20	40	7.68	0.36	874
40	30	8.16	0.40	1301
40	40	8.60	0.44	1703
60	30	8.40	0.44	1905
60	40	8.60	0.48	2413
80	30	8.87	0.58	2485
80	40	9.20	0.72	3381
100	30	9.10	0.70	3061
100	40	9.28	0.72	4156

Results: Size vs. Accuracy

56/65

For fixed Curvelx parameters (80 curves, 30 range), effect of varying database size:

#Images	Acc1	Acc2
5K	9.72	0.76
10K	9.44	0.80
15K	9.16	0.70
20K	9.04	0.64

iDistance

iDistance

58/65

iDistance (Jagadish, Ooi, Tan, Yu, Zhang)

- adaptive B-tree based indexing method
- aimed at handling kNN queries in high-d spaces

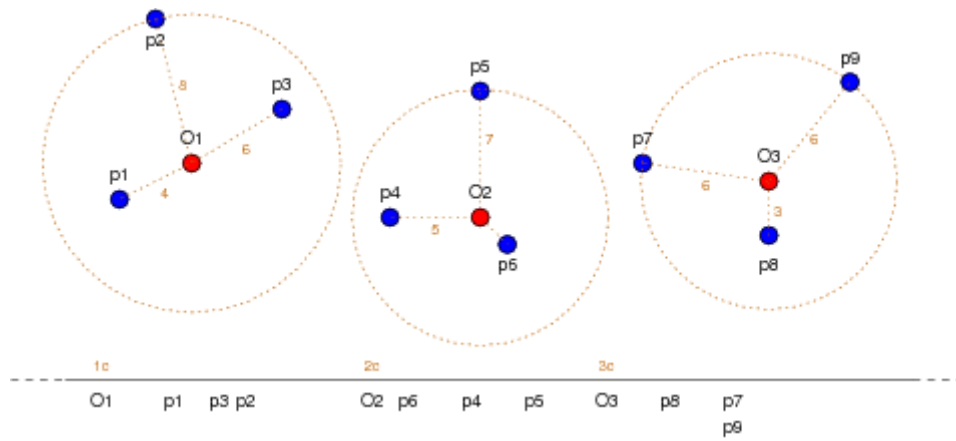
The basic idea:

- determine a set of reference points O_j
(reference points partition the data space)
- build index on $(p_i, D(p_i))$
(distance of each point p_i to its nearest reference point O_j)
- use index to quickly find p_i likely to be close to q

... iDistance

59/65

Computing the iDistance: $D(p) = j.c + dist(p, O_j)$



where c is a constant to "spread" the partitions over the iDistance line

Searching with iDistance

60/65

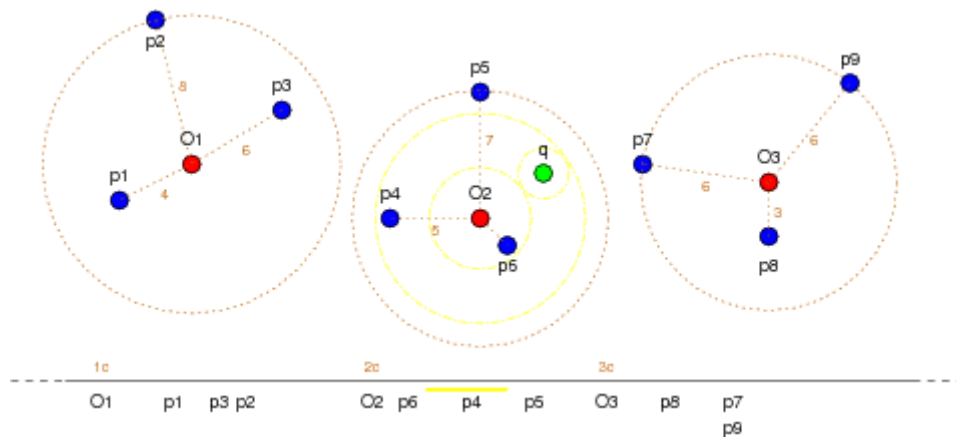
The approach:

- start with query point q
- choose a radius r around query, giving hypersphere S
- for all partitions intersected by S
 - determine $a = \min(\text{dist}(S, O_j))$ and $b = \max(\text{dist}(S, O_j))$
 - find all data points p with $D(p)$ in range $j.c+a \dots j.c+b$
 - maintain a sorted list of $(p, \text{dist}(p, q))$ pairs
- increase radius by $\text{delta-}r$ and repeat above steps
- continue until kNN are found

... Searching with iDistance

61/65

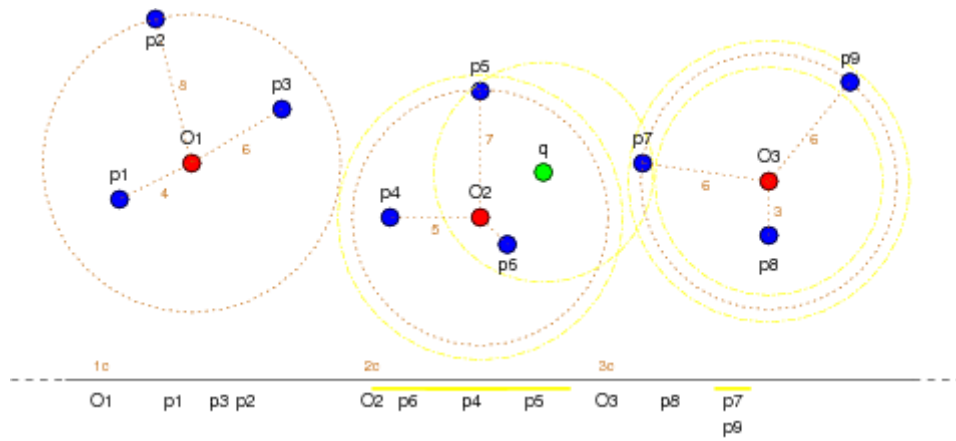
First iteration of search (small r):



... Searching with iDistance

62/65

Later iteration of search (larger r):



... Searching with iDistance

63/65

Details of search method:

```
//Inputs:
// q = query point, k = # nearest-neighbours
// O[m] = set of reference points
// rad[m] = radius of partition around O[i]
// deltaR = search radius increase on each step
// maxR = maximum search radius
int r = 0; // search radius around q
int stop = 0; // flag for when to halt
int seen[]; // partition i already searched?
Obj S[]; // result set, k nearest neighbours
Obj lp[], rp[]; // lists of limits with partitions

while (!stop) {
    r = r + deltaR
    stop = SearchRefs(q,r)
}
...
```

... Searching with iDistance

64/65

```
int SearchRefs(q,r) {
    f = furthest(S,q)
    stop = (dist(f,q) < r && |S| == k)
    for (i = 0; i < m; i++) {
        d = dist(O[i],q)
        if (!seen[i]) {
            hs[i] = sphere(O[i],rad[i])
            if (hs[i] contains q) {
                seen[i] = 1
                leaf = BtreeSearch(i*c+d)
                lp[i] = SearchInward(leaf, i*c+d-r)
                rp[i] = SearchOutward(leaf, i*c+d+r)
            }
            elseif (hs[i] intersects sphere(q,r)) {
                seen[i] = 1
                leaf = BtreeSearch(rad[i])
                lp[i] = SearchInward(leaf, i*c+d-r)
            }
        }
        else {
            if (lp[i] != null)
                lp[i] = SearchInward(left(lp[i]), i*c+d-r)
            if (rp[i] != null)
                rp[i] = SearchOutward(right(rp[i]), i*c+d+r)
        }
    }
    return stop
}
```


... Searching with iDistance

65/65

Cost depends on data distribution and position of O_j

See analysis in ACM Trans on DB Systems, v.30, Jun 2005, p.364

Determining the set of reference points:

- space-partitioning ... divide space up using geometric partitions
- data-partitioning ... try to have equal # points in each partition

Determining the size of δ :

- too small ... more iterations, more access to B-tree index
- too large ... "overshoot" and fetch unnecessary pages (in last iteration)

Produced: 17 Jul 2019