course 7 similarity-based selection

1. Similarity Selection

we use similarity selection since we can not a precise matching condition, and we conduct this function by defining a measure d of “distance” between tuples

uses of similarity selection:

- text or multimedia retrieval
- ranked queries in conventional databases

1.1 similarity-based retrieval

SB-retrieval use threshhold and count_k to restrict the solution set to only the “most similar” object

and the navie approach for SB-retrieval is shown below:

```
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
```

the distance computation base feature vector, and what is feature vector:

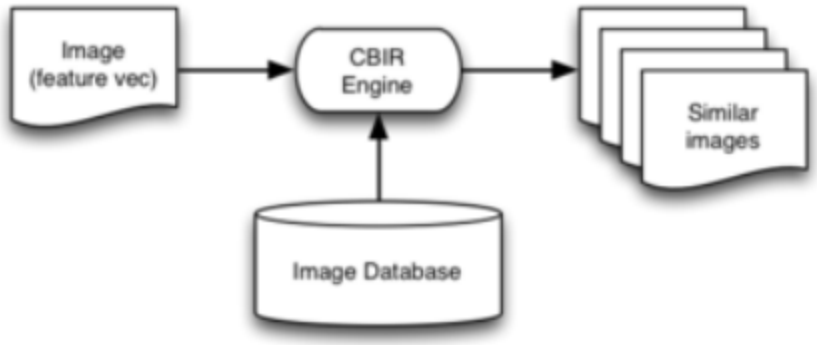
- for text data, it may be term frequencies
- for music data, it may be loudness/pitch/tone
- for image data, it may be colour histogram

and we typically use several features and then concatenated into single vector

and a feature vector represent a point in high-dimension space

Example: content-based Image Retrieva

user supply a description or sample of desired image, and the CBIR engine return a rank list of “matching” images from database



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));
```

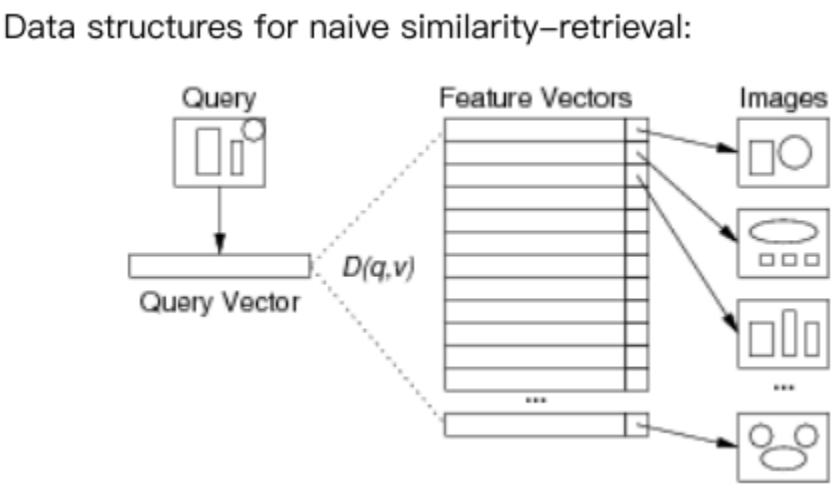
where the ~~ operator measures distance between images.

... Example: Content-based Image Retrieval

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

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



1.2 approaches to KNN retrieval

Partition-based:

- use auxiliary data structure to identify candidates
- space-partitioning methods: Grid file, k-d trees, quad-tree
- data-partitioning methods: R-tree, X-tree, SS-tree, TV-tree

but unfortunately, such methods fail when dimension > 10..20

Approximation-based:

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

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

2. Vector Approximation (VA) Files

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

2.1 VA-file signatures

VA-files have properties:

- much more compact than features vectors

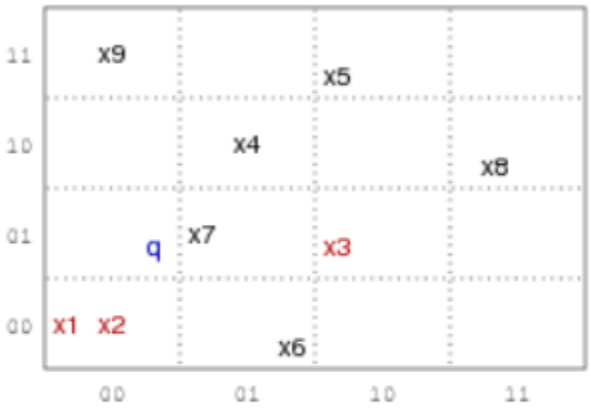
and approach to querying:

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

so how can we computing VA signatures?

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

- feature vector: *v = (255,128,0) = (**11**111111,**10**000000,**00**000000)*
- partition each dimension into 4 regions (⇒ *m=2* bits per *d*)
- VA signature for this vector: *va(v) = (11,10,00)*



v1 = (00000011,00001111)	va1 = 0000	*d = 2*
v2 = (00001111,00001111)	va2 = 0000	*m = 2*
v3 = (10000011,01000011)	va3 = 1001	

2.2 query with VA-Files

input；query vector vq

output: return a list of tids of k database objects nearest to query

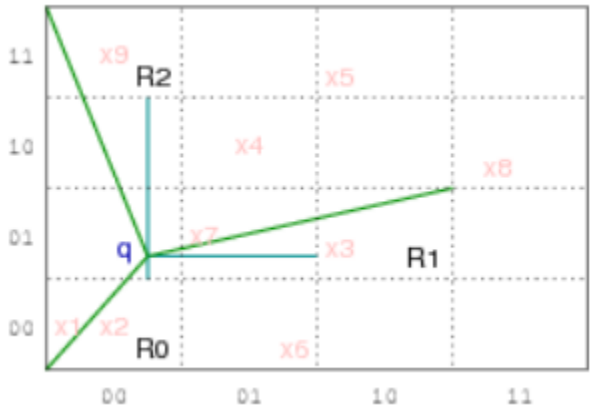
```
results = [];  maxD = infinity;
for (i = 0; i < r; i++) {
   // fast distance calculation
   dist = minDistance(region[va[i]], vq)
   if  (#results < k  ||  dist < maxD)  {
      dist = distance(v[i],vq)
      if (#results < k  ||  dist < maxD)  {
         insert (tid[i],dist) into results
     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

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]*



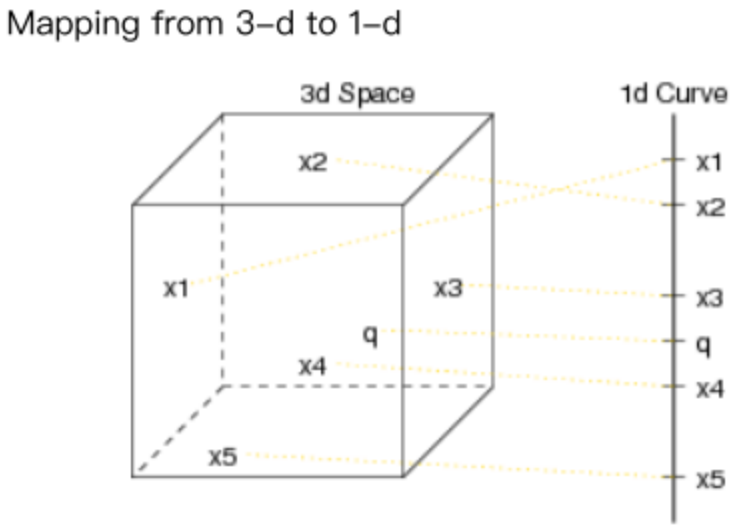
An improved query algorithm that guarantees minimal *D* computations:

```
vaq = vasignature(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
}
```

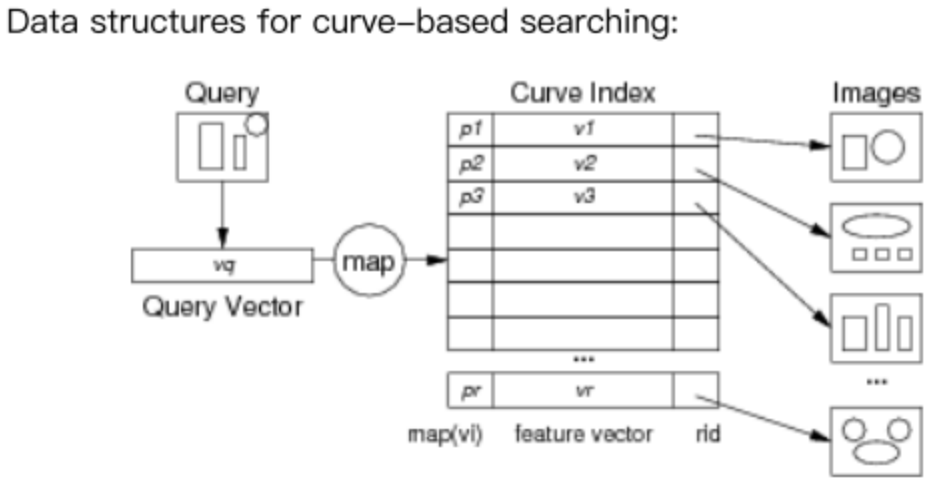
3. Curve-based Similarity Search

3.1 curve-based searching

when the dimension of the data is to high, we can use a dimension-reducation methods to project thee data onto 1-d (line/curve), and this give a linear ordering of the project, which can then peform search over this linear ordering with B-trees



3.2 Curve Index File organisation



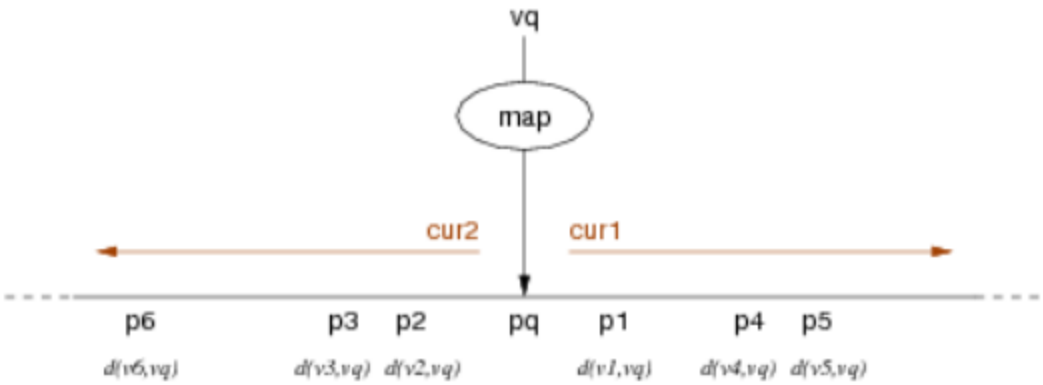
3.2.1 searching with curve index

input: query feature vector vq

output: collected tids of similar objects

```
p = map(v_q)
cur_1 = cur_2 = lookup(p)
while (not enough answers) {
    (pt,vec,tid) = next(cur_1)
    remember tid if D(vec,v_q) small enough
    (pt,vec,tid) = prev(cur_2)
    remember tid if D(vec,v_q) small enough
}
```

How curve is scanned to find potential near–neighbours:



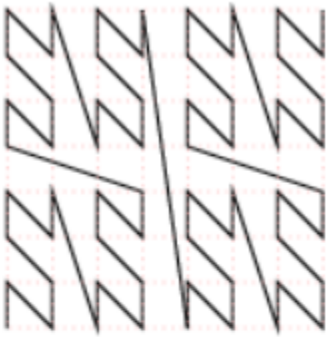
overall, when perform searching with curve index, we need a curve fill all feature space, and then we can project the high level data onto a line/curve, which is in 1-d space, after that we can perform search over this linear ordering, finaly we can use some existing effcient 1-d access methods (such as B-trees) to access data.

so there is a question, what kind of curves are required to make this approach viable?

- must pass through every data point exactly once
- must pass thourgh every “point” in the underlying space exactly once

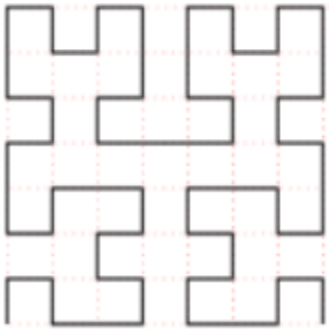
2d Space-filling Curves

Row-wise enumeration

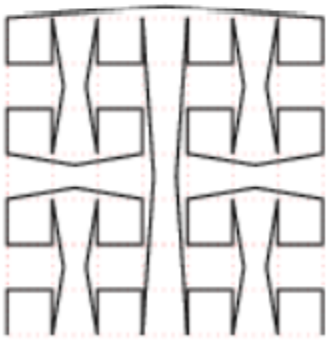


Peano curve

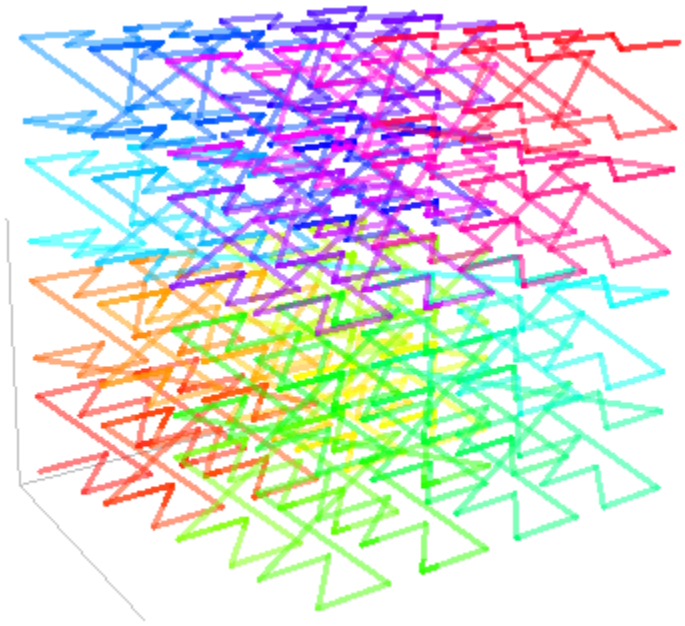
... 2d Space-filling Curves



Hilbert curve



Gray ordering



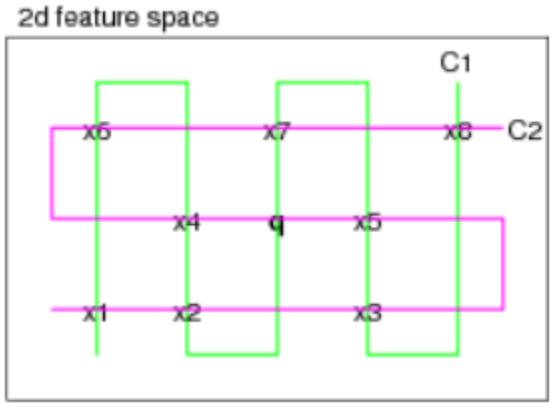
3.3 The CurveIx Scheme

becasue of the problem shown below, we need use several “complementary” curves

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)

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



so with one more complementary curve, for single query point, we can find two set of kNN, and the find the intersection.

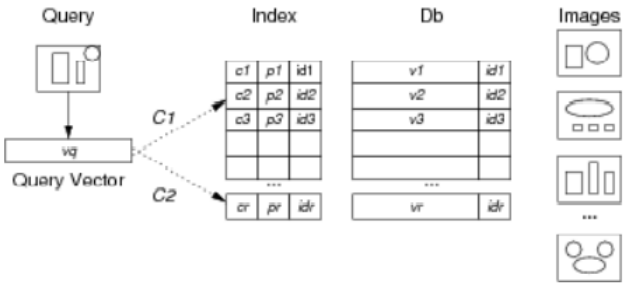
so how many curves we need and how many neighbours do we examine on each curve?

3.3.1 data structure for curlex

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 CurveIx 45/65



For each image *obj*, insertion requires:

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

3.3.2 Finding k-NN in CurveIx

Given: v_q, $C_1 .. C_m$, *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
}
```

3.3.3 Performance Analysis

we have several question to answer:

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

experiement:

Measures for accuracy:

- Acc1 average top-10 entries in Curvelx top-10
- Acc2 how frequently Curvelx gives 10 out of 10

Measures for effciency:

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

to determine how these measures vary:

- built databases of size 5K, 10K, 15K, 20K
- for each database, run 25 query “benchmark” set
- for each query, run for 3,5,10,20,30,40 curve-neighnours
- for each query, run for 20, 40,60,80,100 curves

also implement a linear-scanning version for comparsion and to collect the exact answer sets.

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

For fixed CurveIx 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

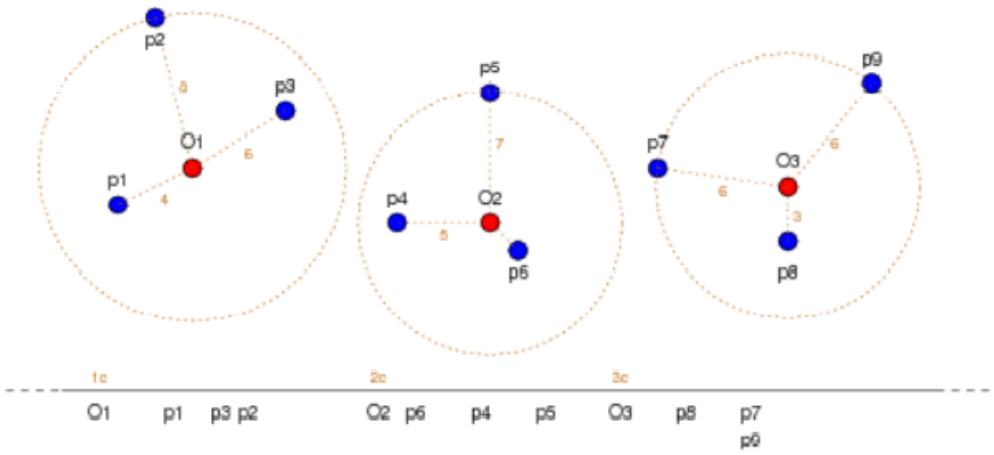
4. iDistance

iDistance:

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

the basic idea:

- determine a set of reference points Oj
- build index on (p_i, D(p_i))
- use index to quickly find p_i likely to be close to q



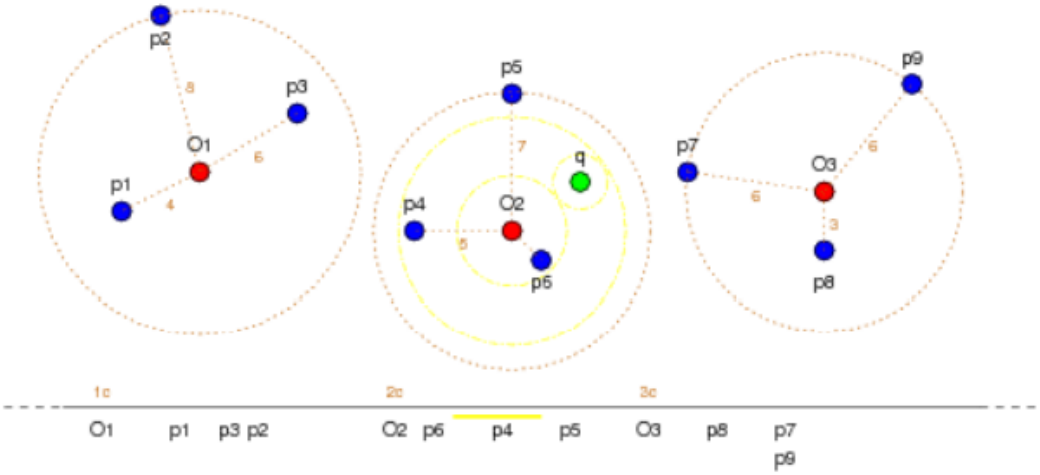
where *c* is a constant to "spread" the partitions over the iDistance line

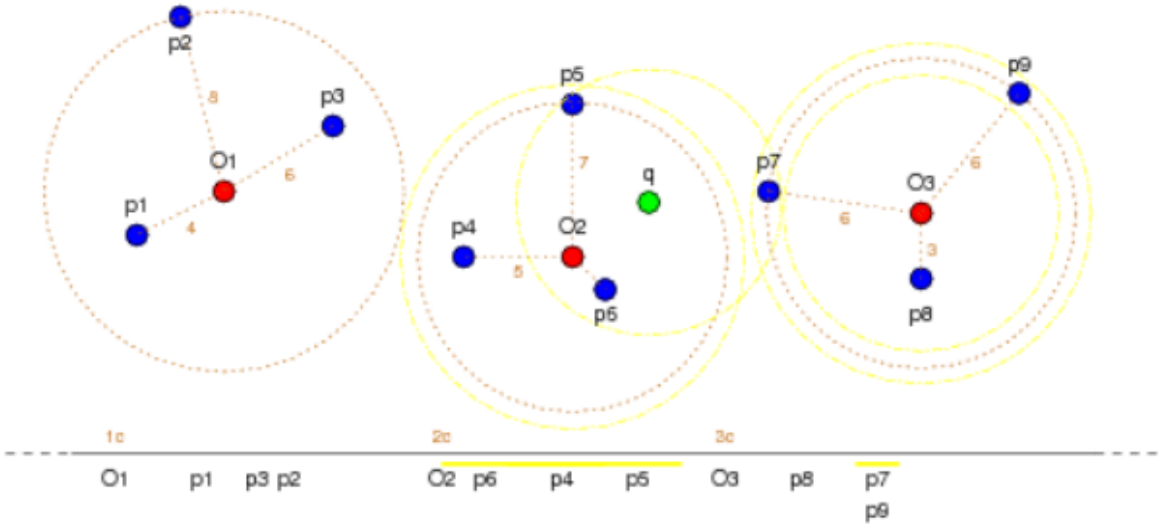
4.1 searching in iDistance

The approach:

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

First iteration of search (small *r*):





cost depends on data distribution and position of O_i

and you can see detail analysis in ACM Trans on DB System

Determine the set of reference points:

- space-partitioning: divide space up using geometric partitions
- data-partitioning: try to have equal

Determining the size of deltaR:

- too small: more interactions, more access to B-tree index
- too large: “overshoot” and fetch unnecessary pages