Clustering Billions of Images with Large Scale Nearest Neighbor Search

Ting Liu, Charles Rosenberg, Henry A. Rowley

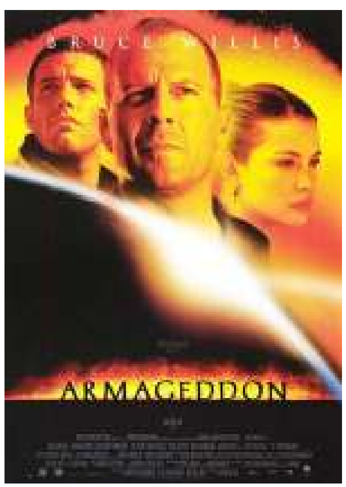
IEEE Workshop on Applications of Computer Vision
February 2007

Presented by Dafna Bitton on May 6th, 2008
for CSE 291

Problem Statement

Goal 1: Find approximate nearest neighbors for a repository of over one billion images

Goal 2: Perform clustering based on the results



Context of the Task

- Billions of images on the web
- Modern image search is text-based, largely due to so many images!
- Scale makes most computer vision tasks infeasible in real time

Nearest Neighbor Search (NNS): Applications

First step for…
- Image clustering
- Object recognition and classification

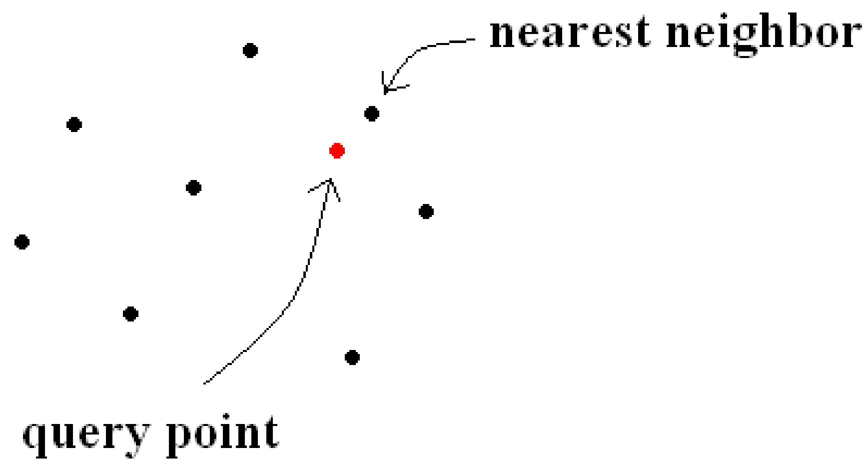
Useful for…
- Organizing the images on the web by finding near duplicate images of items such as CD covers

Outline

- Background
  - Brute-force nearest neighbor search
  - *k*-D trees
  - Metric Trees
  - Spill Trees
  - Hybrid Spill Trees
- Image preprocessing
- Parallel computing framework and data partition
- MapReduce
- Using MapReduce for parallel version of Hybrid Spill Trees
- Results

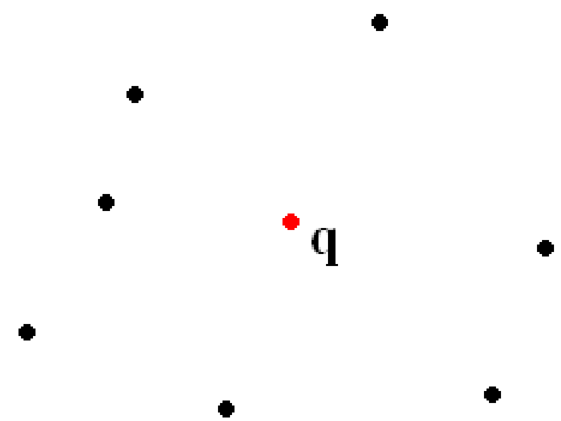
NNS: Math Framework

- Assume a d-dimensional space S
- Assume a set of points $T \subset S$
- Assume a distance measure
- Given a new point $p \in S$, we want to find the point $v \in T$ that is most similar to p



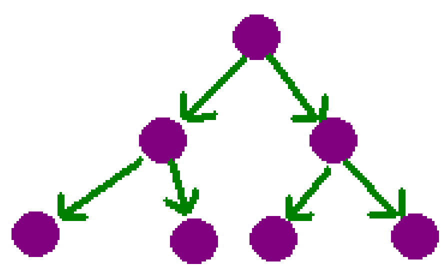
Brute-force NNS

- Given a new point $p \in S$, compute the distance between p and every point $v \in T$.
- Whichever point in T has the smallest distance is the nearest neighbor

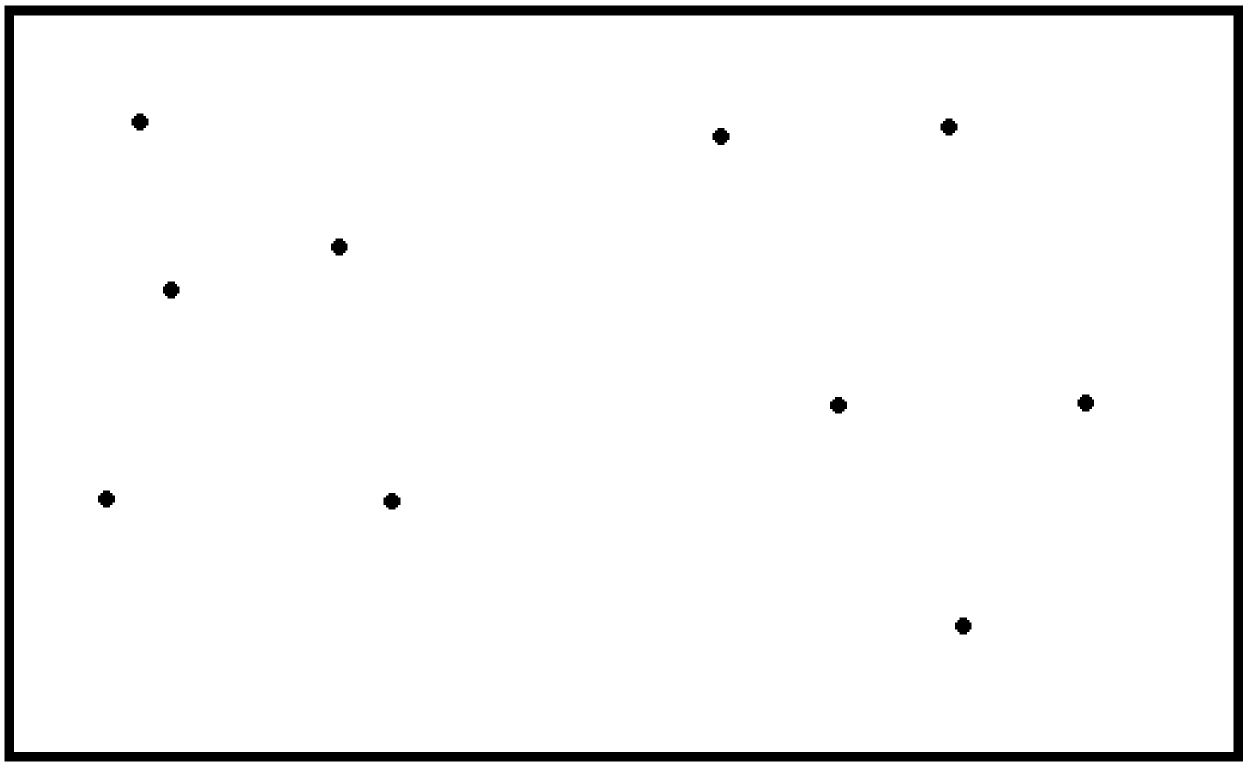


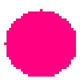
k-D Trees

- Axis-parallel partitions of the data
- Root of the tree represents the entire space
- Invariant: the union of each level of the tree represents the entire space

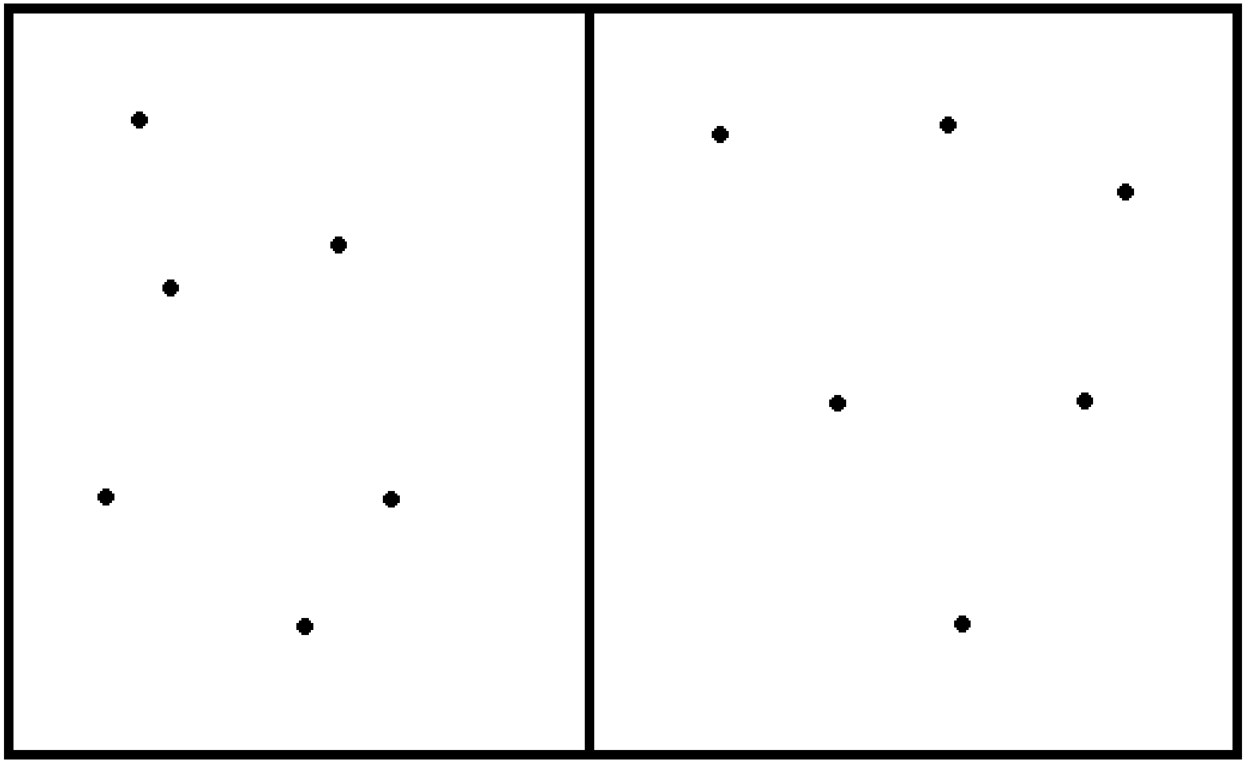


Example of *k*-D Trees

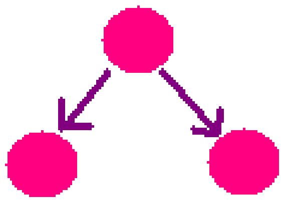


Status of *k*-D tree → 

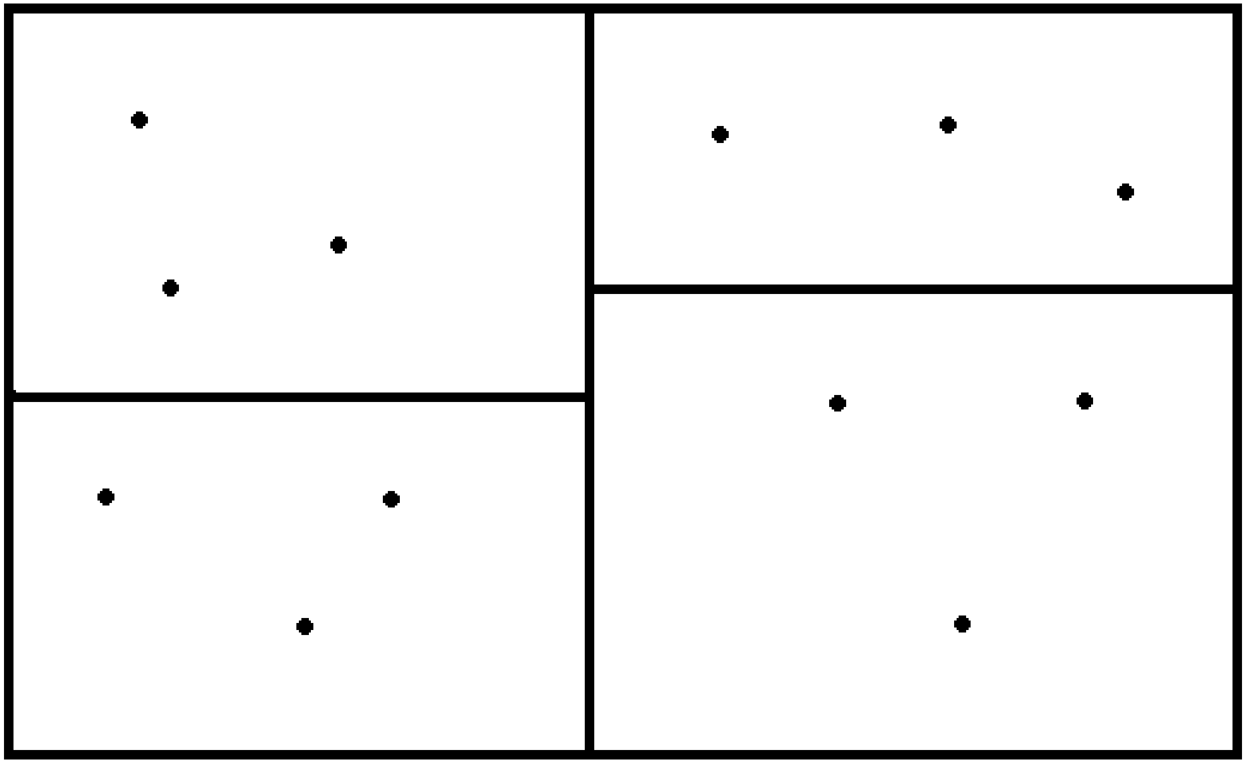
Example of *k*-D Trees (cont.)



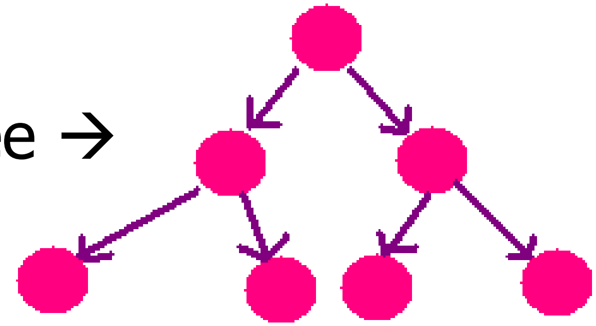
Status of *k*-D tree →



Example of *k*-D Trees (cont.)

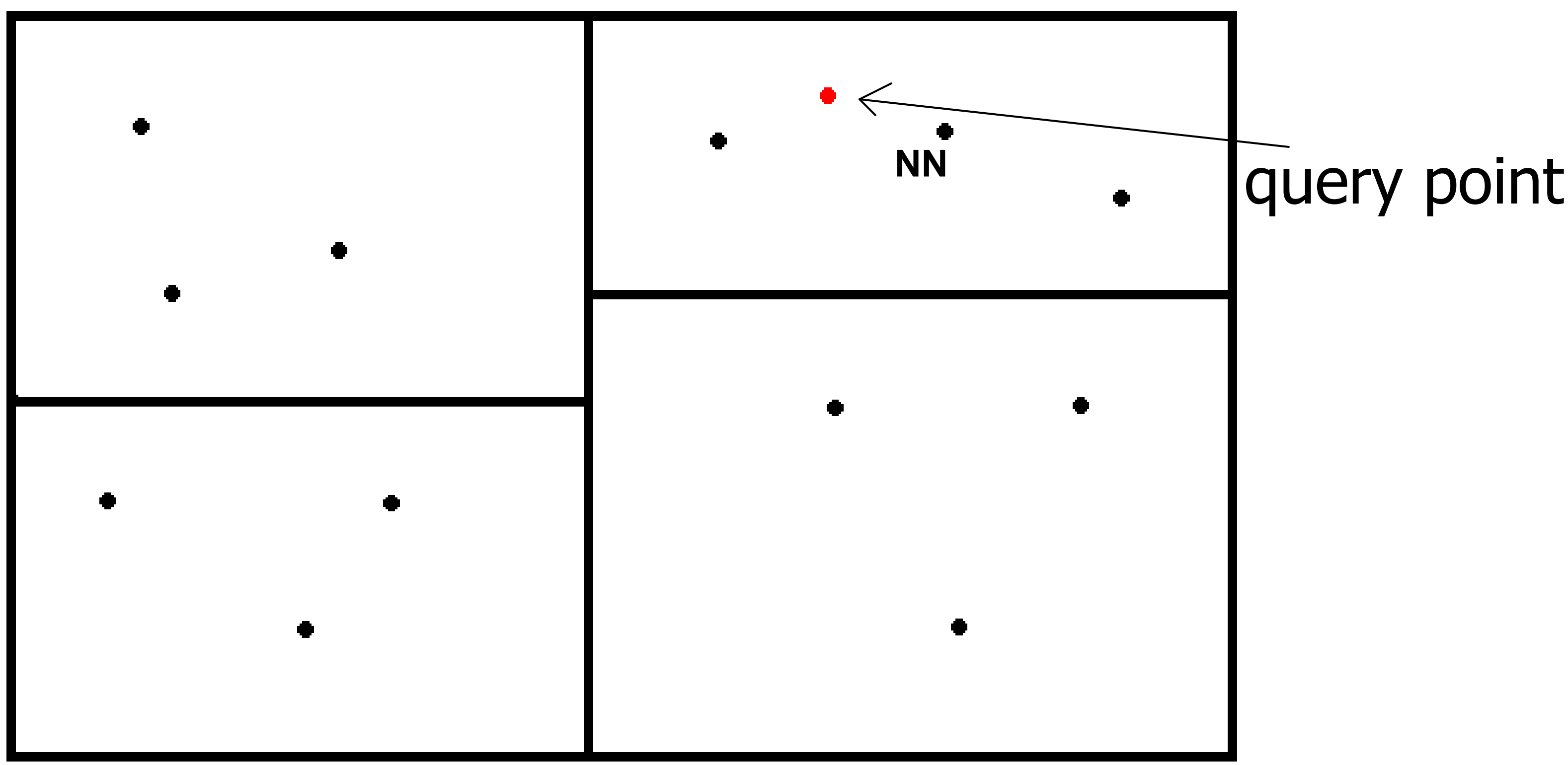


Status of *k*-D tree →



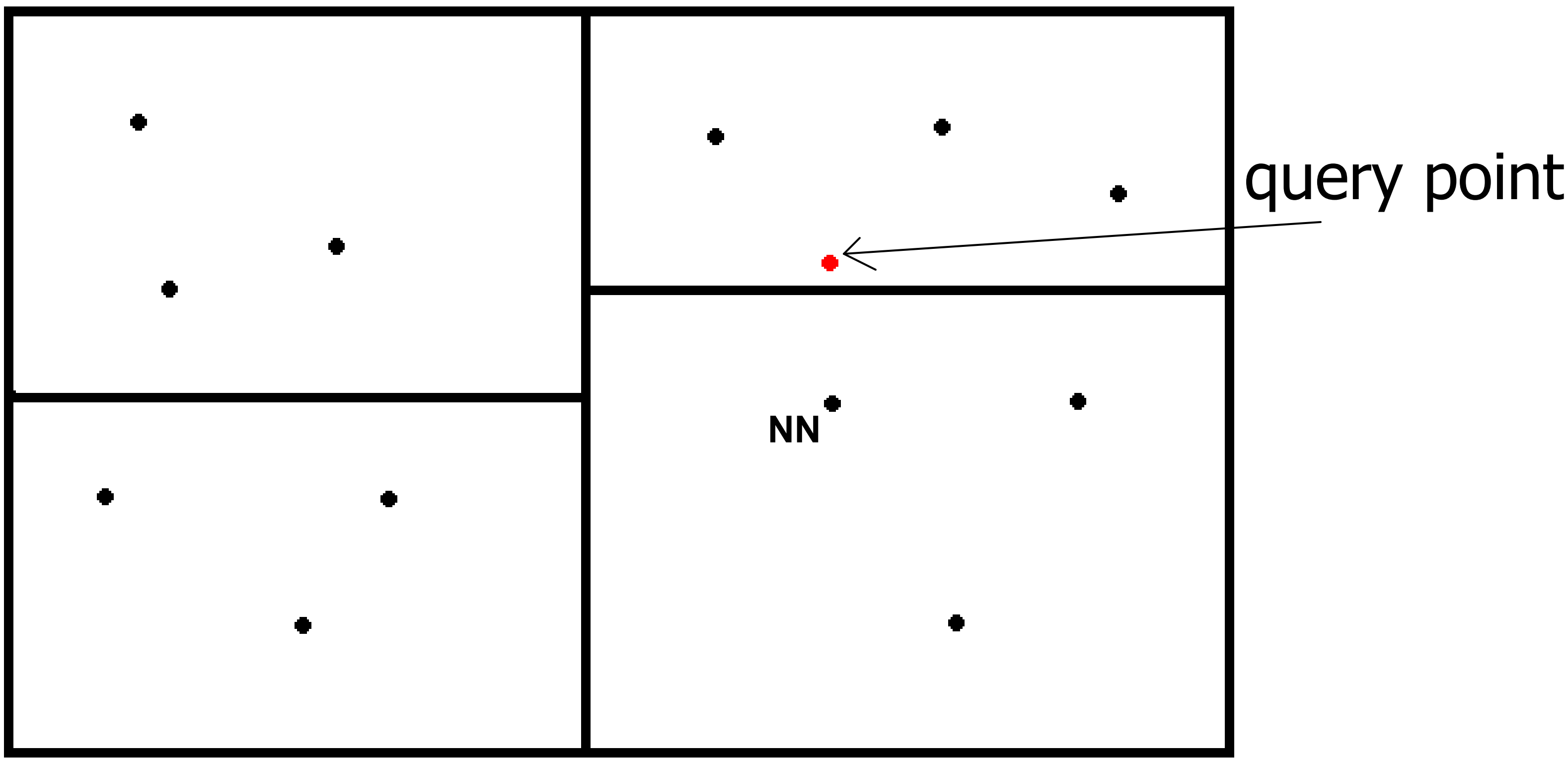
Example of *k*-D Trees (cont.)

→Ideal case when searching:
nearest neighbor falls into the same node as the query



Example: *k*-D Trees (cont.)

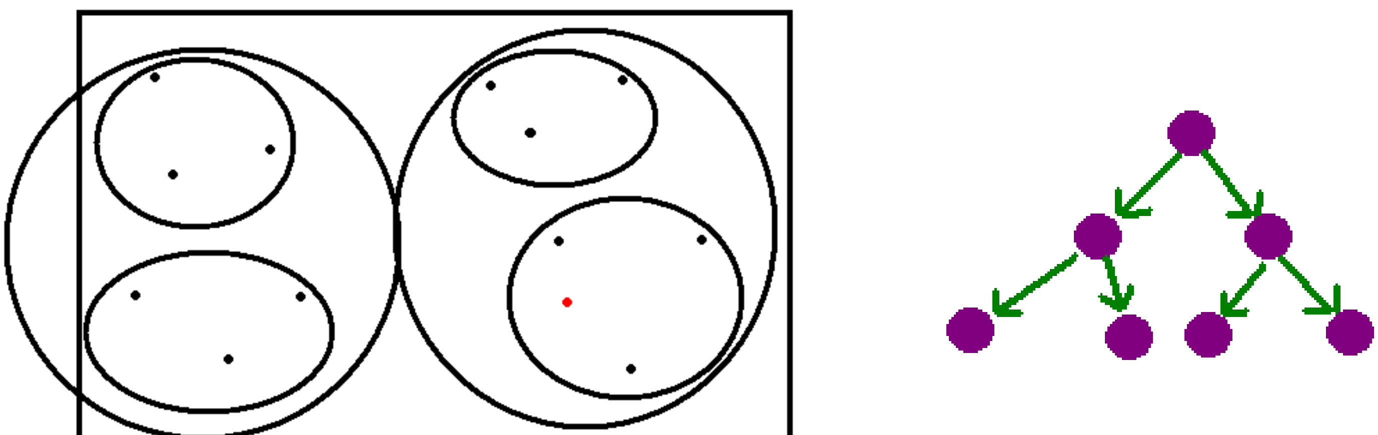
→Unfortunate case when searching: nearest neighbor falls into a different node as the query



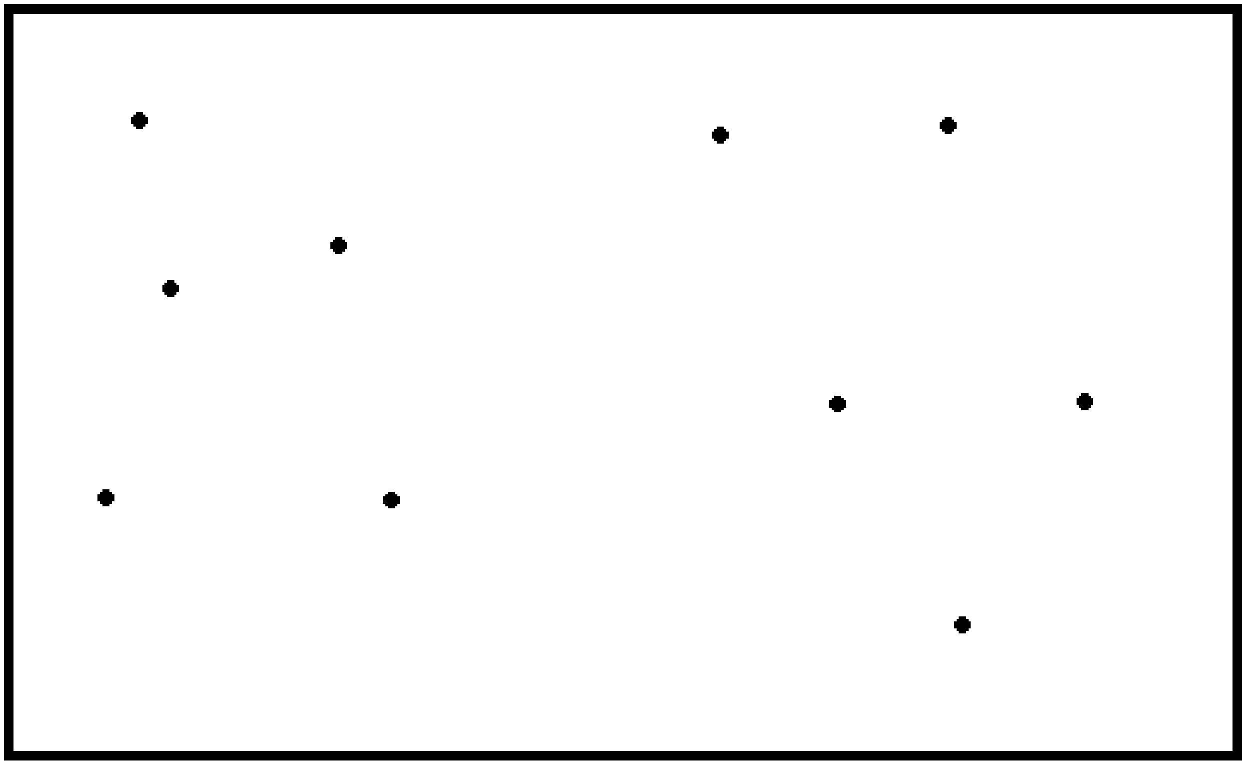
Must do backtracking!

Metric (ball) Trees

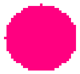
- Same as *k*-D trees except we use hyperspheres to partition the data



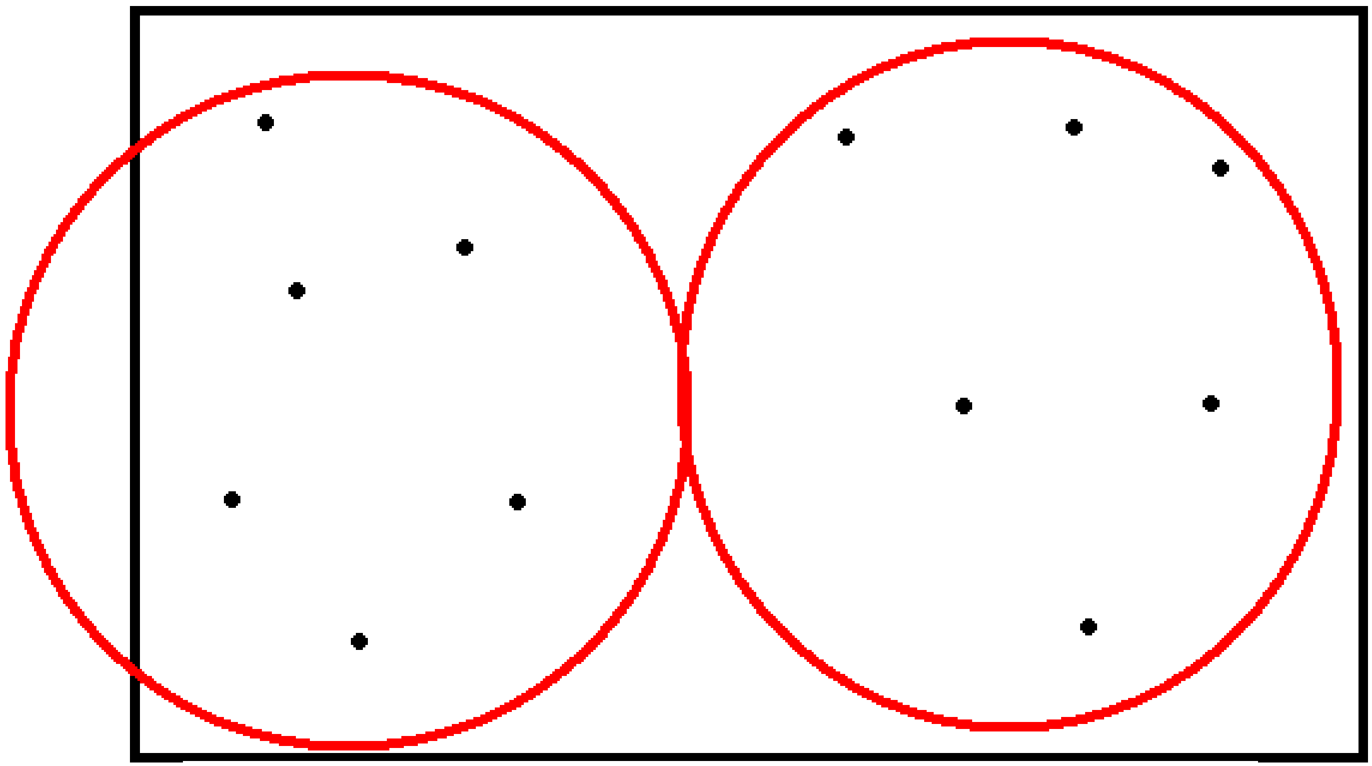
Example of Metric Trees



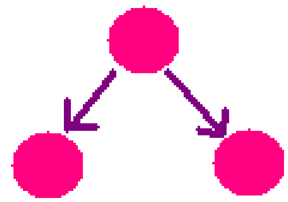
Status of metric tree →



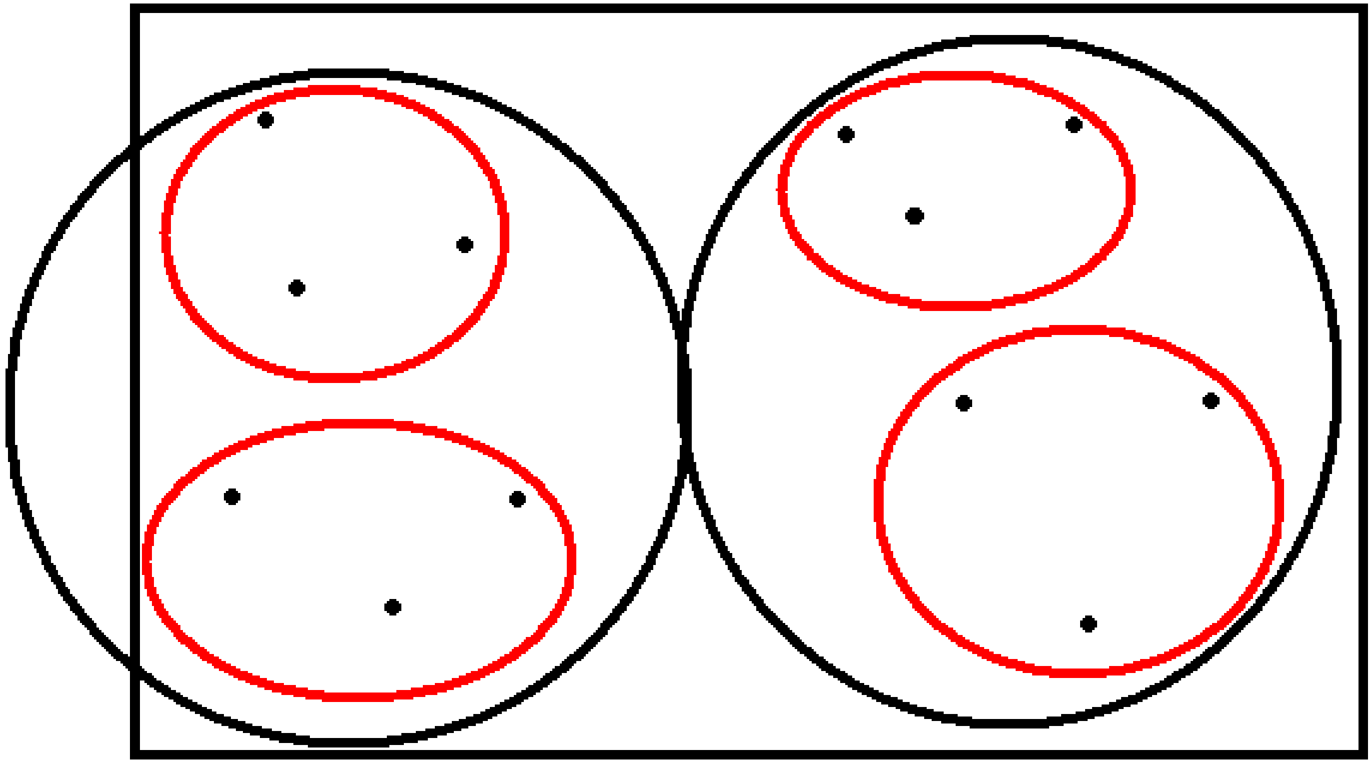
Example of Metric Trees (cont.)



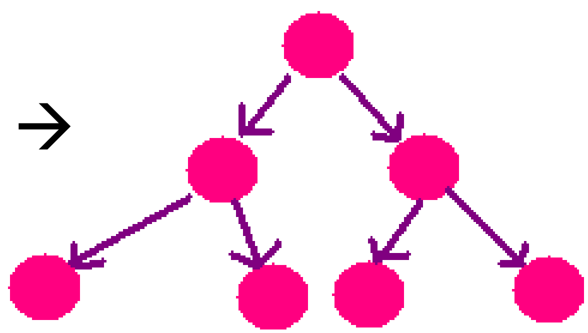
Status of metric tree →



*Child ownership cannot overlap, but spheres can

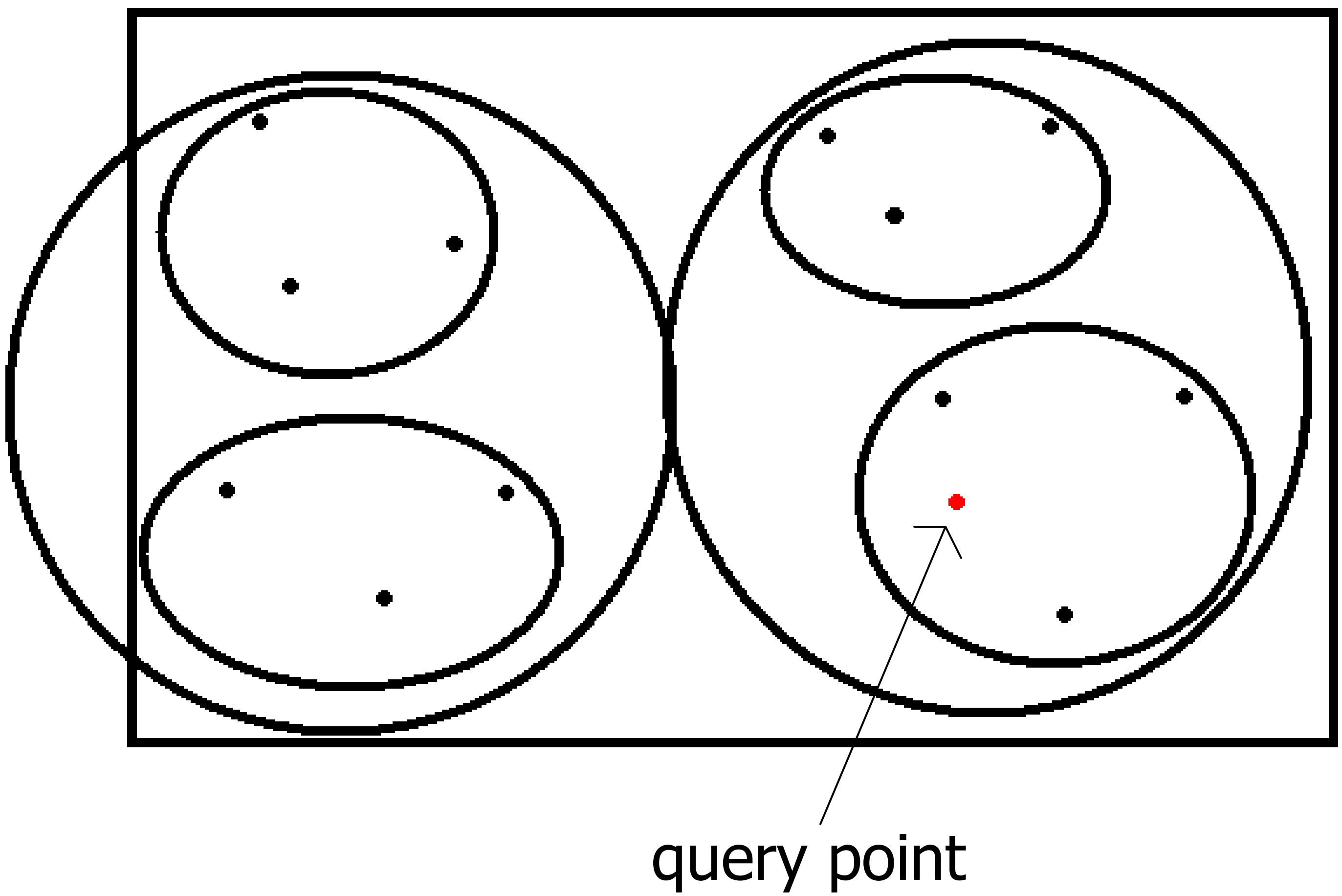


Status of metric tree →



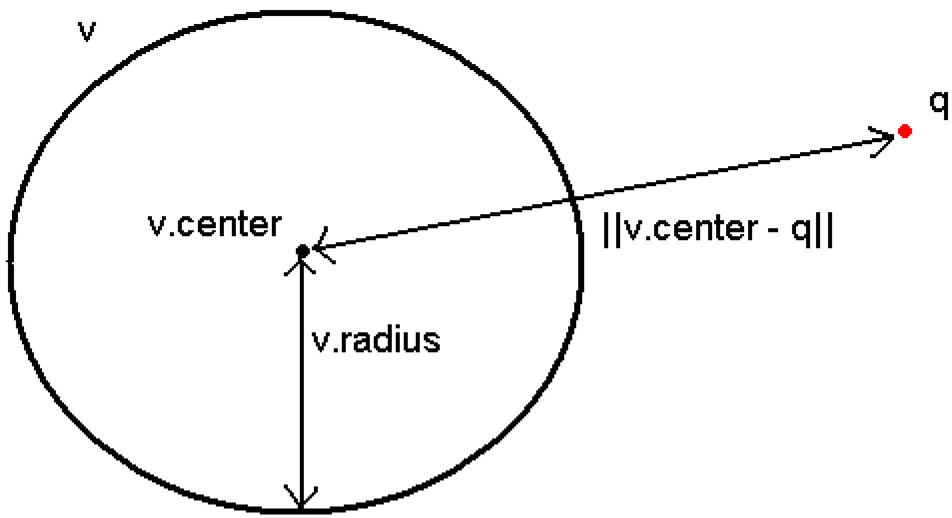
Invariant: $x \in$ sphere → d(center of sphere, x) < radius of sphere

Example of Metric Trees (cont.)



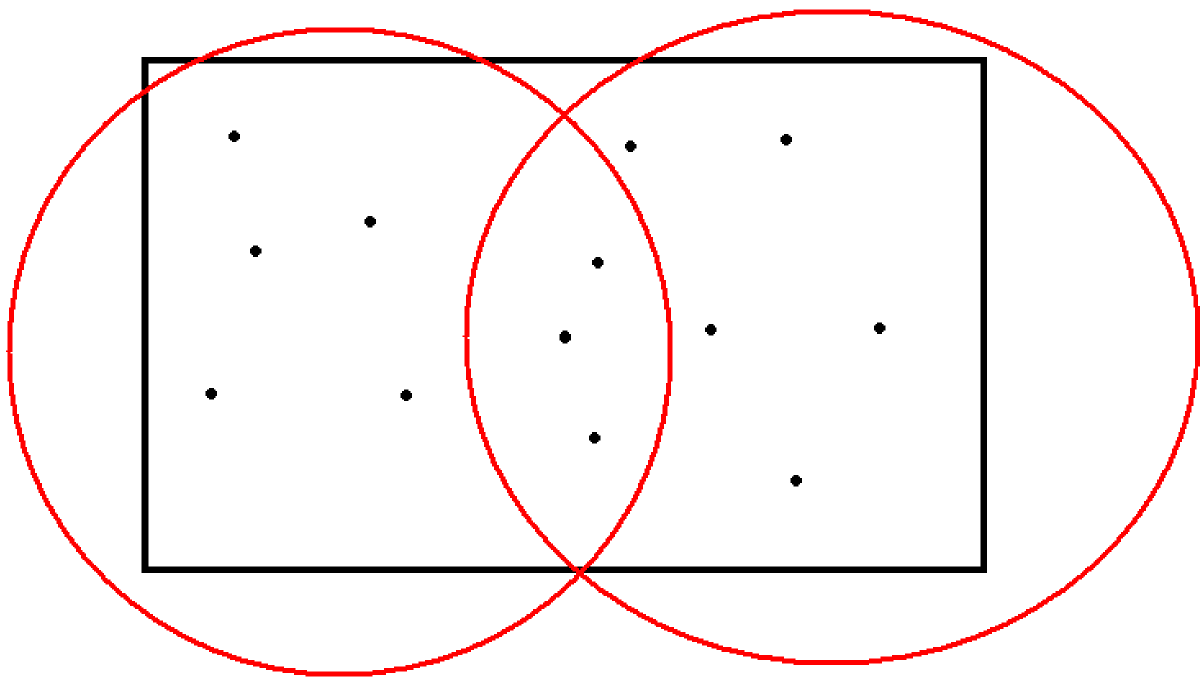
Searching with Metric Trees

- Guided depth first search (DFS) with pruning
- Descend the tree to reach the hypersphere leaf node where the query lies
- Assign a "candidate NN", x, with distance r from the query.
- If DFS is about to visit a node v, but no member of v can be within distance r from the query, prune this node (do not visit it or any of its children)
- This is whenever $\| v.center - q \| - v.radius \geq r$



Spill Trees

- Similar to Metric Trees except that the children of a single node can share data points.



Metric vs. Spill

- Let N(v) denote the set of points represented by node v
- Let v.lc and v.rc denote the left and right children of v
- In Metric Trees:

$$N(v) = N(v.lc) \cup N(v.rc)$$
$$\varnothing = N(v.lc) \cap N(v.rc)$$

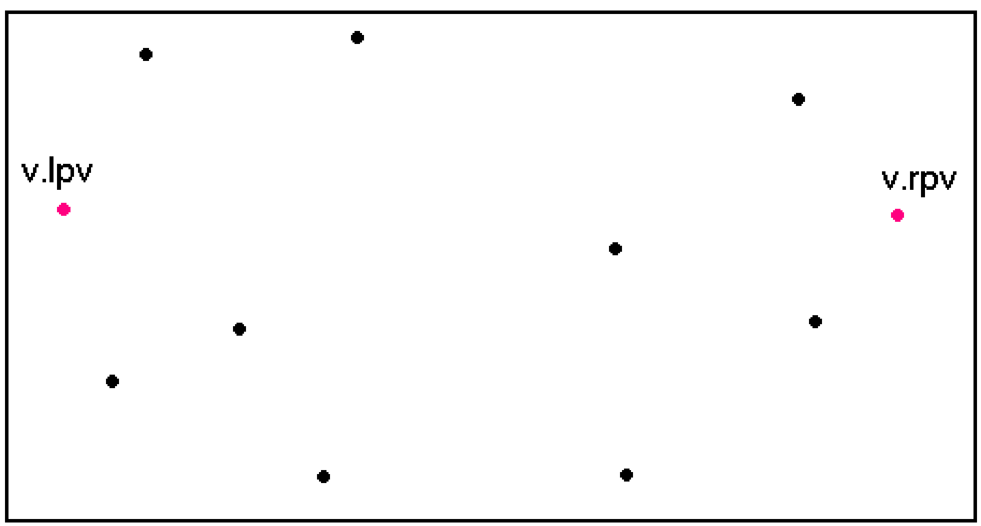
- In Spill Trees:

$$N(v) = N(v.lc) \cup N(v.rc)$$
$$\varnothing \leq N(v.lc) \cap N(v.rc)$$

Constructing a Spill Tree

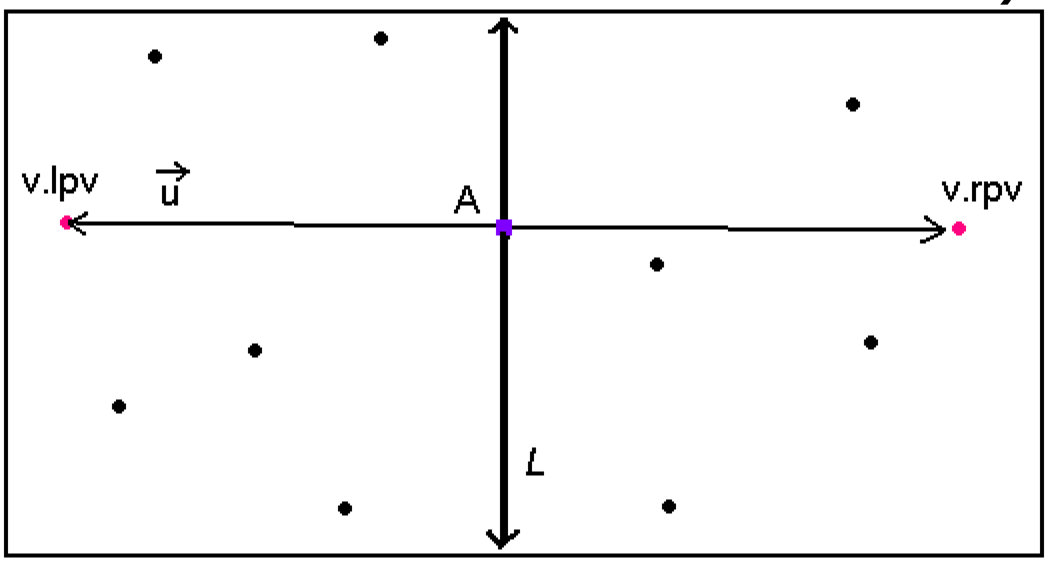
- Given a node v, we choose two pivot points v.lpv $\in$ N(v) and v.rpv $\in$ N(v), ideally such that they are maximally separated.
- Specifically,

$$\| v.lpv - v.rpv \| = \max_{p1, p2 \in N(v)} \| p1 - p2 \|$$



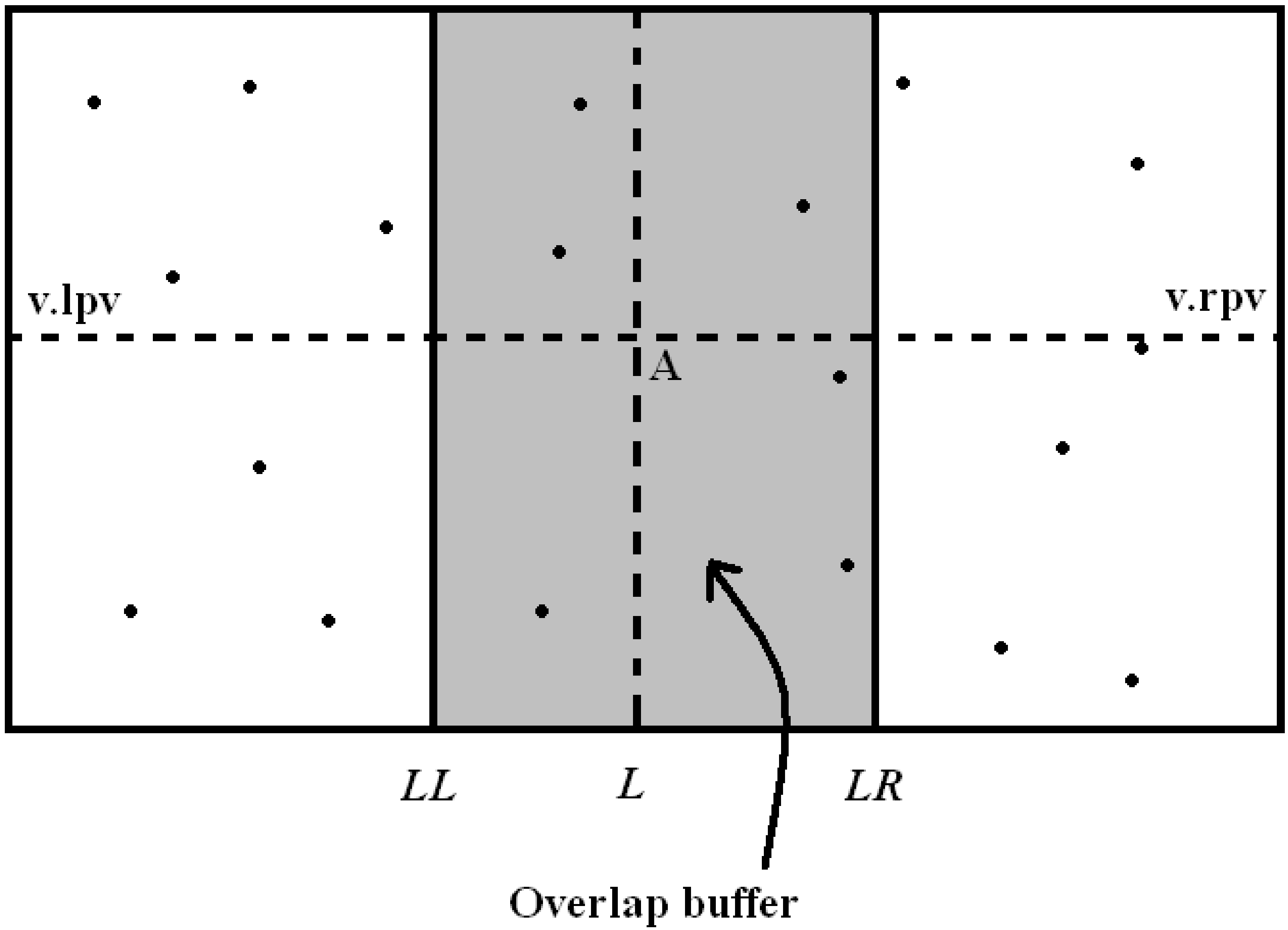
Constructing a Spill Tree (cont.)

- Project all the data points down to the vector
$$\vec{u} = \overrightarrow{v.rpv} - \overrightarrow{v.lpv}$$
- Find the midpoint A along $\vec{u}$
- *L* denotes the d-1 dimensional plane orthogonal to $\vec{u}$, which goes through A.
- *L* is known as the *decision boundary*



Constructing a Spill Tree (cont.)

- We define two separating planes *LL* and *LR*, both parallel to and at distance τ from *L*
- *LL* and *LR* define a stripe, also known as the *overlap buffer*
- Metric Trees have empty stripes
- All data points to the right of *LL* belong in v.rc
- All data points to the left of *LR* belong in v.lc
- All data points in the stripe are shared by v.lc and v.rc



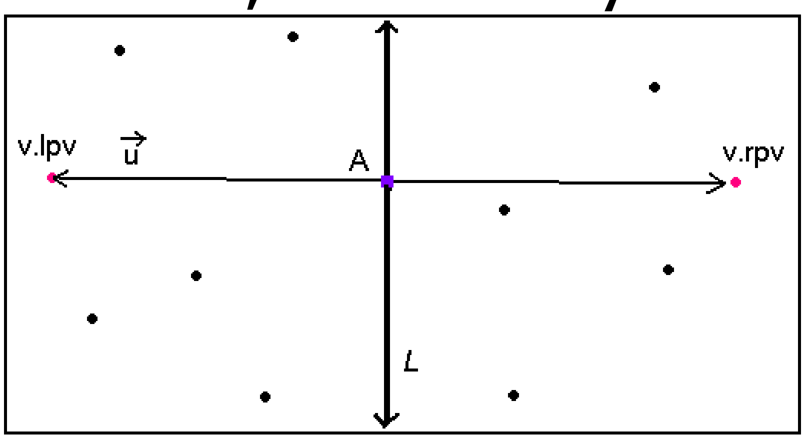
v.lpv
v.rpv
A
LL
L
LR
Overlap buffer

Spill Tree NN Search

- Use *defeatist search*, which descends the tree according the the decision boundary *L* at each node, without backtracking, outputting the point x in the first leaf node visited.
- Not guaranteed to find the correct NN
- Wider stripe means slower search, but more accurate

Drawbacks of Spill Trees

- The depth of Spill Trees varies considerably depending on τ (where 2τ is the overlap buffer size)
- If τ =0, the Spill Tree acts as a Metric Tree
- If $\tau \geq \| v.rpv - v.lpv \| / 2$, then $N(v.lc) = N(v.rc) = N(v)$ and construction of a Spill Tree does not even terminate, giving it a depth of ∞
- To address this, we use Hybrid Spill Trees

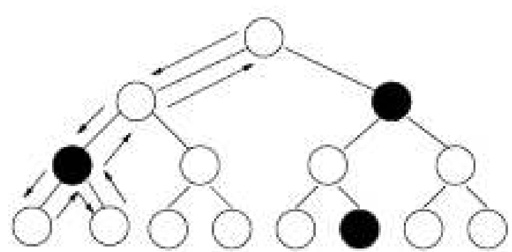


Hybrid Spill Trees

- Define a *balance threshold* $\rho < 1$ usually set to 70%
- For each node v, we first split the data points using the overlapping buffer
- If either of its children contains more than ρ fraction of the total data points in v, we undo the overlapping splitting, instead use a conventional metric-tree partition, and mark v as a *non-overlapping node*
- This ensures that each split reduces the number of data points of a node by a constant factor, maintaining logarithmic depth of the tree

Hybrid Spill Tree Search

- Hybrid of Metric Tree DFS and defeatist search
- Only do defeatist search on overlapping nodes
- For non-overlapping nodes, we still do backtracking as Metric Tree DFS



The Drawbacks

- All of these algorithms were designed to run on a single machine
- In our case, our data cannot all fit on a single machine, and disk access is too slow
- Noise affects distance metric
- Curse of dimensionality

→ Authors will address these drawbacks using a new variant of spill trees

Outline

- Background
  - Brute-force nearest neighbor search
  - *k*-D trees
  - Metric Trees
  - Spill Trees
  - Hybrid Spill Trees
- Image preprocessing
- Parallel computing framework and data partition
- MapReduce
- Using MapReduce for parallel version of Hybrid Spill Trees
- Results

Image Preprocessing

- Normalize each image
- Scale the image to a fixed size of 64x64 pixels (each pixel is 3 bytes)
- Convert image to Haar wavelet domain
  - All but the largest 60 magnitude coefficients are set to 0, and the remaining ones are quantized to +/- 1
- So far, the feature vector is 64x64x3, which is still fairly large

Image Preprocessing (cont.)

- Random projection using random unit-length vectors is used to reduce the dimensionality to 100 dimensions
- 4 additional features are added:
  - The average of each color channel
  - The aspect ratio $w/(w+h)$
- Now the feature vectors are of dimensionality 104

Outline

- Background
  - Brute-force nearest neighbor search
  - *k*-D trees
  - Metric Trees
  - Spill Trees
  - Hybrid Spill Trees
- Image preprocessing
- Parallel computing framework and data partition
- MapReduce
- Using MapReduce for parallel version of Hybrid Spill Trees
- Results

Parallel Computing Framework

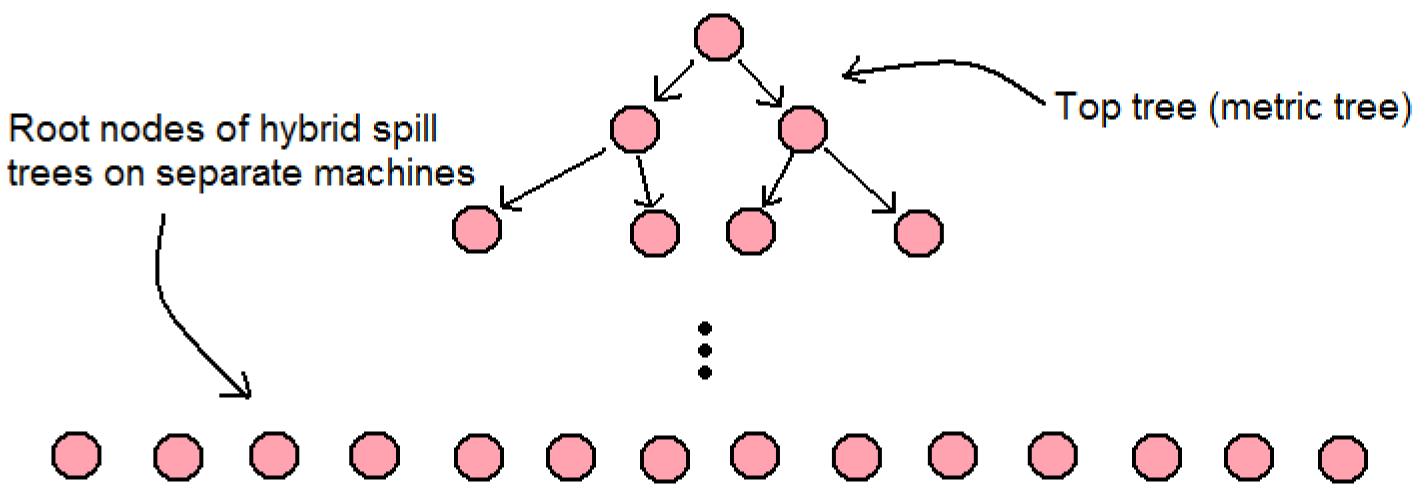
- Main challenge: all feature vectors must be in main memory
- In our case, feature vector = 104 floating point numbers = 416 bytes
- On a machine with 4GB, we could fit 8 million images
- However, we are dealing with 1 billion images, so we would need at least 100 machines

How to Partition the Data?

- One option: random partition, building a separate spill tree for each partition
- More intelligent option: use a metric tree structure
- Why Metric Trees?
  - Non-overlapping children
  - Saves space

Metric Trees to Partition Data

- Take a random sample of all of the data, small enough to fit on one machine (1/M of the data), and build a metric tree for this data
- Each leaf node in this *top tree* defines a partition, for which a spill tree can be built on a separate machine



Building the Top Tree

- Stopping condition for the leaf nodes is an upper bound on the leaf size
- We need each partition to fit on a single machine, so we set the upper limit to roughly this
- There is also a lower bound to prevent partitions from being too small

Outline

- Background
  - Brute-force nearest neighbor search
  - *k*-D trees
  - Metric Trees
  - Spill Trees
  - Hybrid Spill Trees
- Image preprocessing
- Parallel computing framework and data partition
- MapReduce
- Using MapReduce for parallel version of Hybrid Spill Trees
- Results

MapReduce

Map

A user-defined *Map Operation* is performed on each input key-value pair, generating zero or more key-value pairs. This phase works in parallel, with the input pairs being arbitrarily distributed across machines.

MapReduce (cont.)

Shuffle

Each key-value pair generated by the Map phase is distributed to a subset of machines, based on a user defined *Shuffle Operation* of their keys.

Within each machine the key-value pairs are grouped by their keys

MapReduce (cont.)

Reduce

A user-defined *Reduce Operation* is applied to all key-value pairs having the same key, producing zero or more output key-value pairs.

Outline

- Background
  - Brute-force nearest neighbor search
  - *k*-D trees
  - Metric Trees
  - Spill Trees
  - Hybrid Spill Trees
- Image preprocessing
- Parallel computing framework and data partition
- MapReduce
- Using MapReduce for parallel version of Hybrid Spill Trees
- Results

Generating the Sample Data

Map: For each input object, output it with probability 1/M

Shuffle: All objects are taken to a single machine

Reduce: Copy all objects to the output

Building the Top Tree

On one machine, build the top tree using the standard metric tree building procedure, with an upper bound U on the cardinality of the leaf nodes, as well as a lower bound L.

Partitioning of Data and Creation of Leaf Subtrees

Map: For each object, find which leaf subtree it falls into, and output this number as the key along with the object

Shuffle: Each unique key is mapped to a different machine

Reduce: On all objects in each leaf subtree, apply the serial hybrid spill tree algorithm to create the leaf subtree

Efficient Queries of Parallel Hybrid Spill Trees

- On top tree, speculatively send each query object to multiple leaf subtrees when the query is close to a boundary
- This is a runtime version of the overlap buffer
- The benefit is that fewer machines are required to hold the leaf subtrees since there are no duplicates

Finding Neighbors in Each Leaf Subtree

Map: For each input query, descend the top metric tree. At each node in the top tree, the query may be sent to both children if it falls within the pseudo-overlap buffer. Generate one key-value pair for each leaf subtree that is searched.

Shuffle: Each distinct key is mapped to a different machine that holds the appropriate subtree.

Reduce: Standard hybrid spill tree search is used for the objects routed to each of the subtrees, and the k-NN lists for each query are generated.

Combining the k-NN Lists

Map: Copy each query and k-NN list pair to the output

Shuffle: The queries are partitioned randomly (by their numerical value)

Reduce: The k-NN lists for each query are merged, keeping only the k objects closest to the query

Clustering Procedure

1. Compute kNN lists for each image
2. Apply a threshold to drop images that are too far apart
3. Drop singleton images from the 1.5 billion image set, leaving around 200 million images
4. The result is 200 million prototype clusters, which are further combined
5. Union-find algorithm is then applied on one machine

Clustering Procedure (MapReduce)

Map: Input is the kNN list for each image, as well as the distance to each of those images.

1. Apply a threshold to the distances, which shortens the neighbor list.
2. The list is treated as a prototype cluster, and reordered such that the lowest image number is first.
3. Generated output consists of this lowest number as the key, and the value is the whole set.
4. Images with no neighbors within the threshold are dropped.

Clustering Procedure (MapReduce cont.)

Shuffle: The keys (image numbers) are partitioned randomly (by their numerical value)

Reduce: Within a single set of results, the standard union-find algorithm is used to combine the prototype clusters

Outline

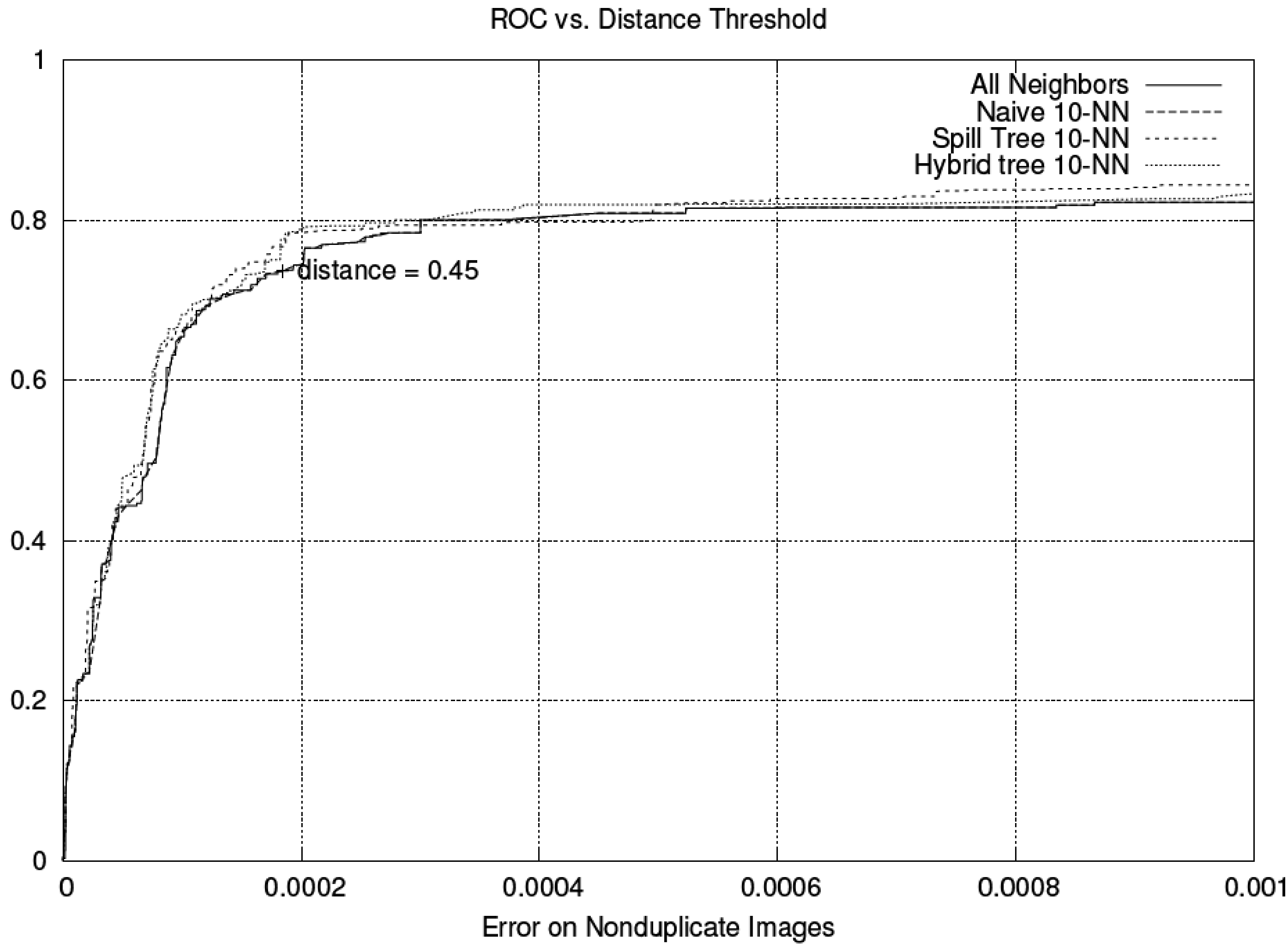
- Background
  - Brute-force nearest neighbor search
  - *k*-D trees
  - Metric Trees
  - Spill Trees
  - Hybrid Spill Trees
- Image preprocessing
- Parallel computing framework and data partition
- MapReduce
- Using MapReduce for parallel version of Hybrid Spill Trees
- Results

Experiments

- Two main datasets used
  - A smaller and labeled dataset that has 3385 images
  - A larger, unlabeled dataset containing around 1.5 billion images

Clustering Results

- On the smaller set, for each pair of images, we compute the distance between their feature vectors
- After varying the distance threshold, we compute clusters by joining all pairs of images which are within the threshold
- Each image pair within each cluster is then checked against manual labeling


ROC vs. Distance Threshold
All Neighbors
Naive 10-NN
Spill Tree 10-NN
Hybrid tree 10-NN
distance = 0.45
Error on Nonduplicate Images
0
0.2
0.4
0.6
0.8
1
0
0.0002
0.0004
0.0006
0.0008
0.001

Clustering Results for Large Set

- Entire processing time for 1.5 billion images was less than 10 hours on 2000 CPUs
- A significant part of the time was spent just on a few machines as the sizes of the subtrees varied considerably
- 50 million clusters found, containing 200 million duplicated images

Clustering Results for Large Set (cont.)

- The most common cluster size is two (because there are often thumbnail and full size image pairs)
- Usually the clusters are accurate but
- Sometimes clusters contain images that are far apart

Visual Results