EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
SCIENCE
Computer Graphics

Massively Parallel Computing

- N-body Problems -

SoSe 2018

Today

- N-Body Problem Introduction

- Nearest Neighbor Search in High Dimensions

- Excerpts from a Tutorial
  - Fast N-body Algorithms for Massive Datasets
  - by Alexander Gray
  - presented at the 2008 SIAM Conference on Data Mining

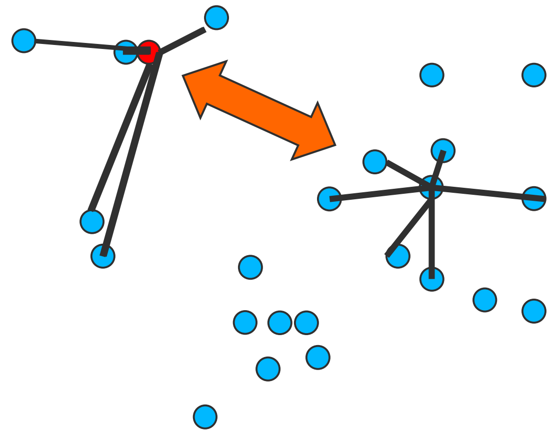
Introduction
N-body Problems

N-Body Problems

- Ubiquitous N-Body Problems
  - Astrophysics
  - Molecular Dynamics
  - Particle discretizations for PDEs
  - Data Mining
  - Irregular Sampling in Graphics

- Simple Observations
  - Points have no intrinsic topology
  - Metric/Kernel relations matter K(x,y)
  - N^2 interactions for all to all

- Typical Problems
  - Nearest neighbors
  - Weighted interpolation
  - Partition of unity
  - Kernel density, Multipole



Two canonical problems

- Nearest-neighbor search

$$NN(x_q) = \arg\min_r \left\| x_q - x_r \right\|$$

- Kernel density estimation

$$\hat{f}(x_q) = \frac{1}{N} \sum_{r \neq q}^{N} K_h(\left\| x_q - x_r \right\|)$$

Ideas

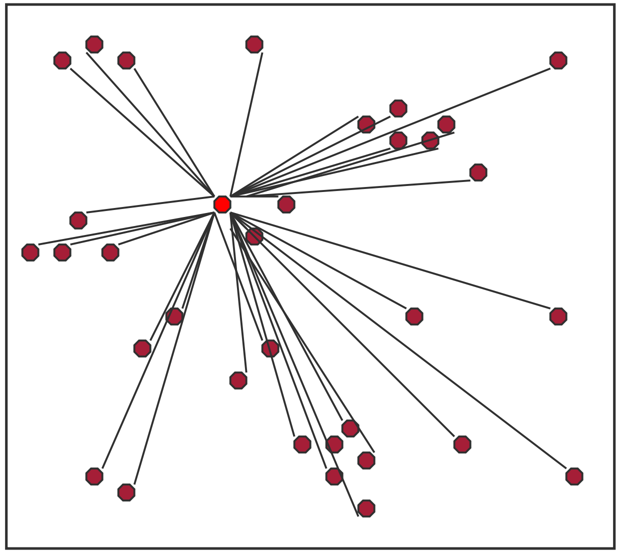
1. Data structures and how to use them
2. Monte Carlo
3. Series expansions
4. Problem/solution abstractions

Nearest Neighbor - Naïve Approach

- Given a query point X.

- Scan through each point Y:
  - Calculate the distance d(X,Y)
  - If d(X,Y) < best_seen then Y is the new nearest neighbor.

- Takes O(N) time for each query!



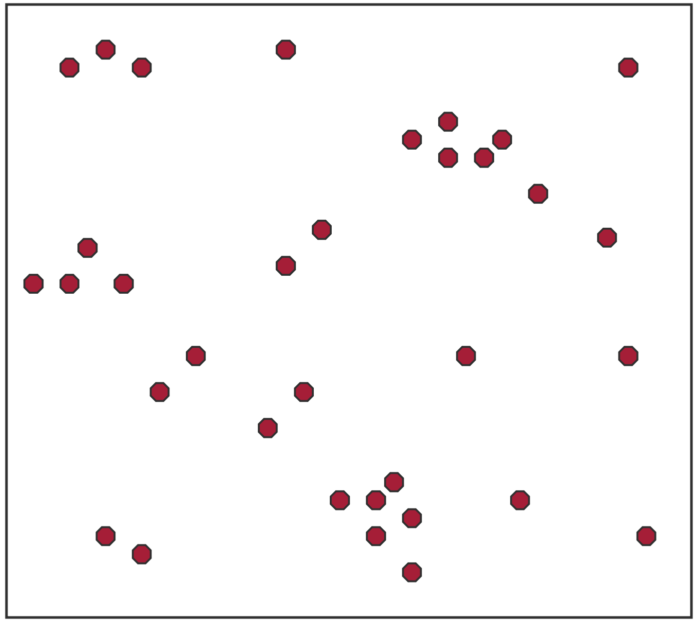
33 Distance Computations

Speeding Up Nearest Neighbor

- We can speed up the search for the nearest neighbor:
  - Examine nearby points first.
  - Ignore any points that are further then the nearest point found so far.

- Do this using a KD-tree:
  - Tree based data structure
  - Recursively partitions points into axis aligned boxes.

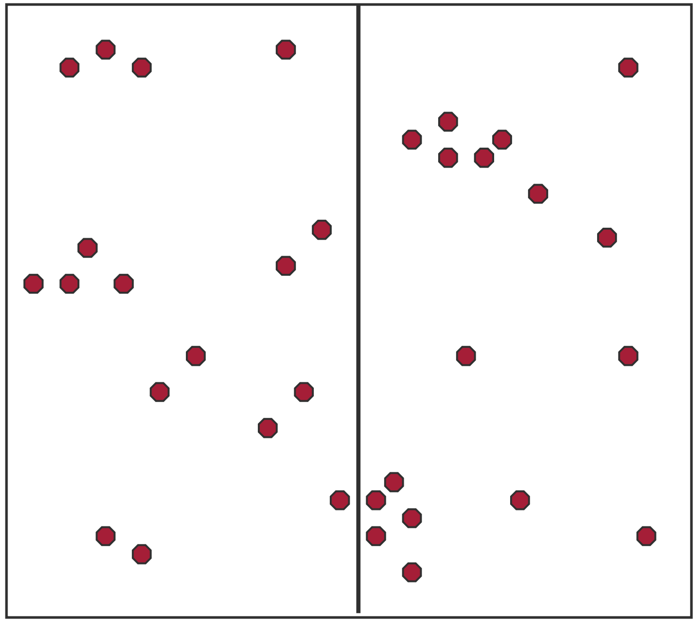
KD-Tree Construction

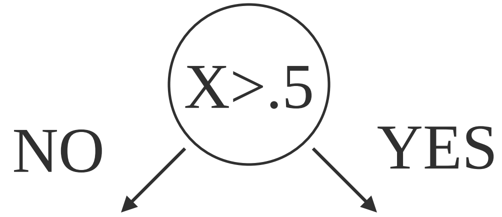


Pt	X	Y
1	0.00	0.00
2	1.00	4.31
3	0.13	2.85
…	…	…

We start with a list of n-dimensional points.

KD-Tree Construction



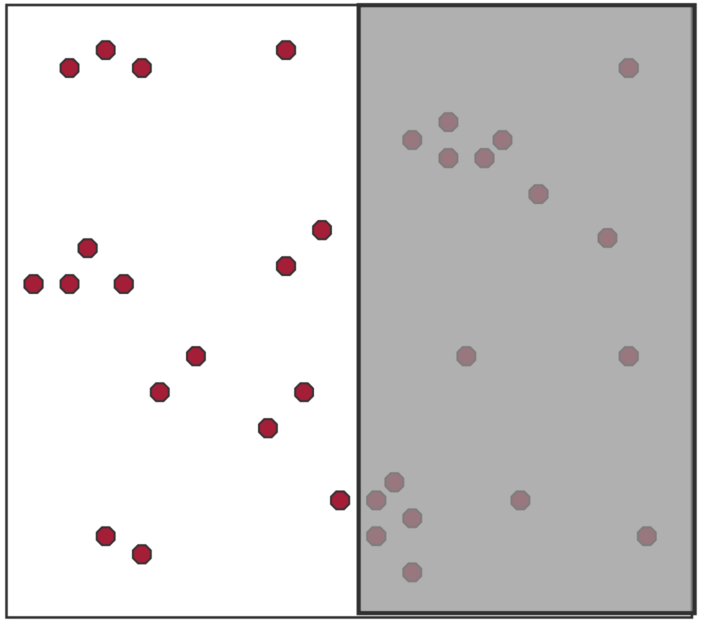


Pt	X	Y
1	0.00	0.00
3	0.13	2.85
…	…	…

Pt	X	Y
2	1.00	4.31
…	…	…

We can split the points into 2 groups by choosing a dimension X and value V and separating the points into X > V and X <= V.

KD-Tree Construction



X>.5

NO

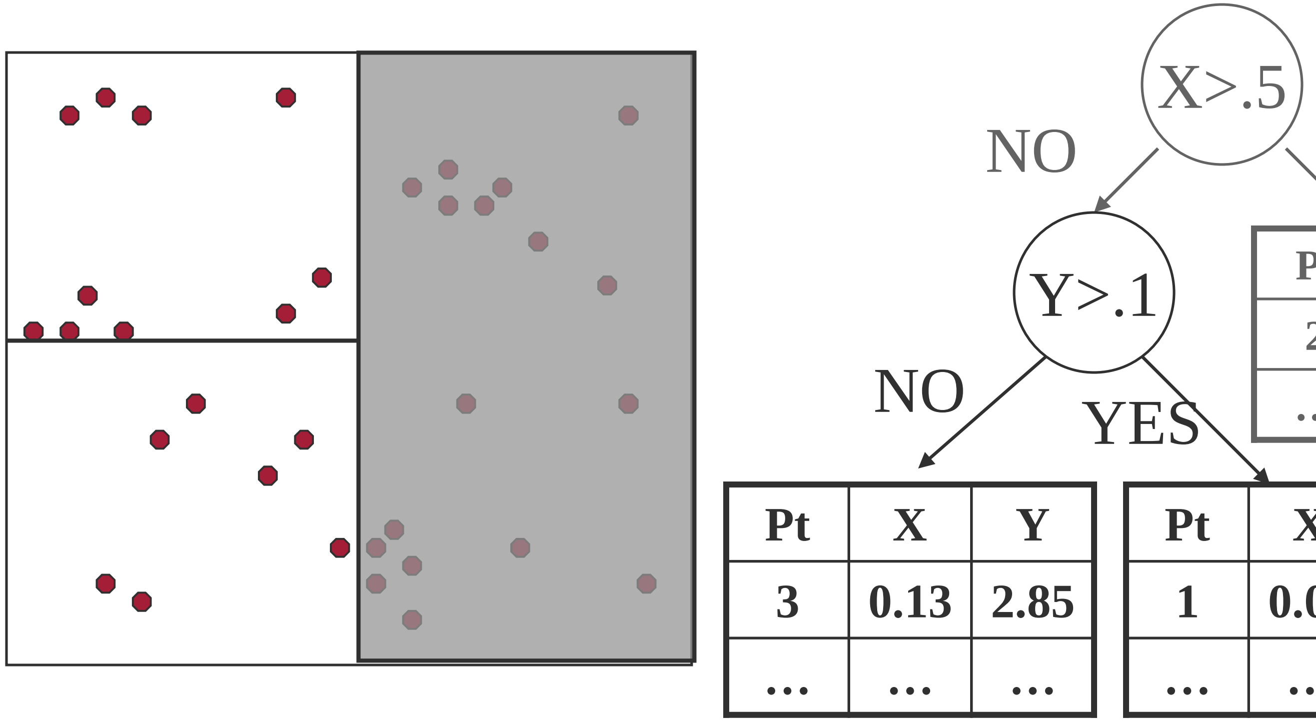
YES

Pt	X	Y
1	0.00	0.00
3	0.13	2.85
…	…	…

Pt	X	Y
2	1.00	4.31
…	…	…

We can then consider each group separately and possibly split again (along same/different dimension).

KD-Tree Construction



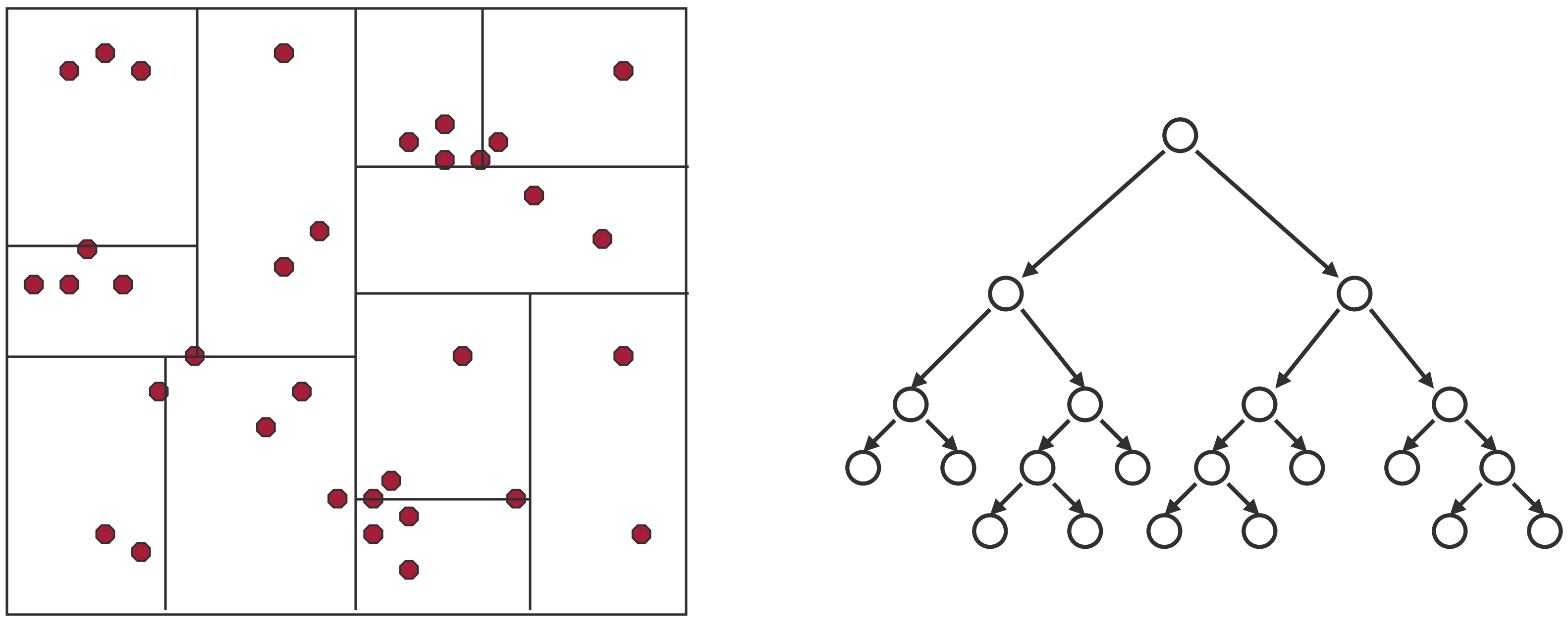
Pt	X	Y
2	1.00	4.31
…	…	…

Pt	X	Y
3	0.13	2.85
…	…	…

Pt	X	Y
1	0.00	0.00
…	…	…

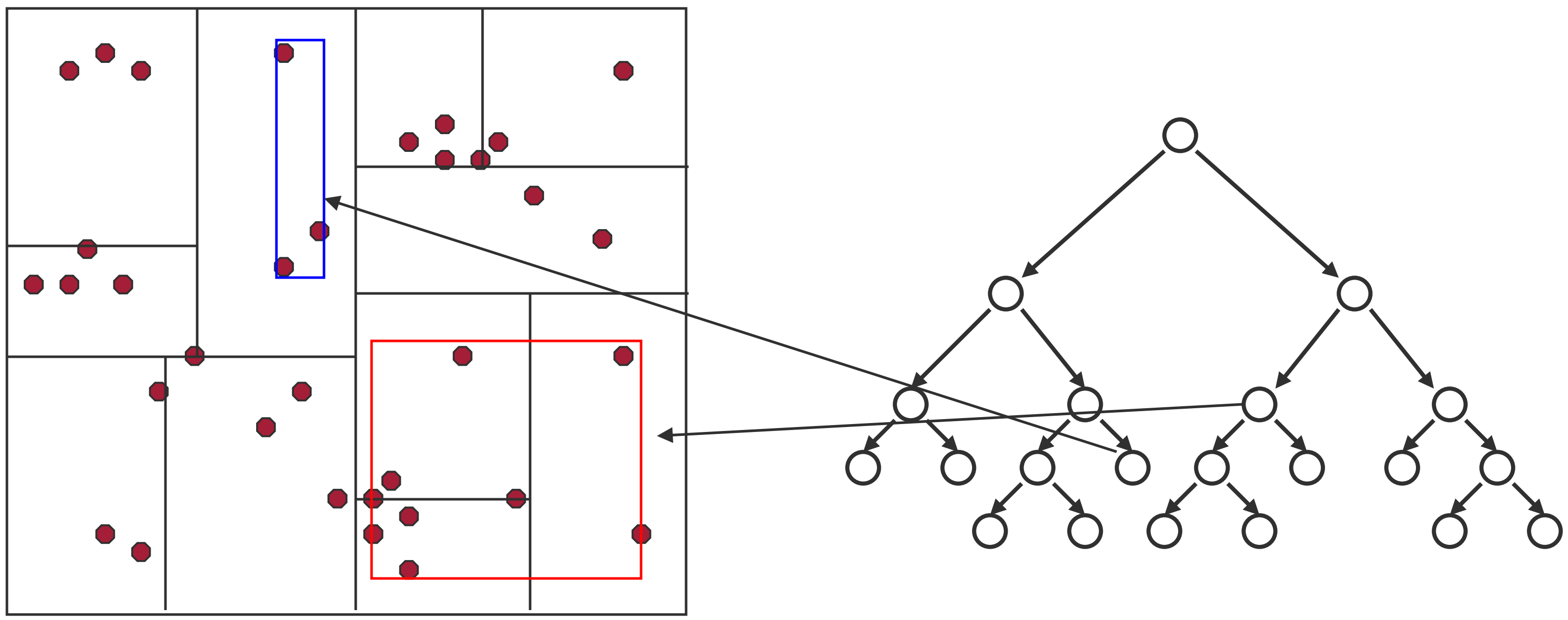
We can then consider each group separately and possibly split again (along same/different dimension).

KD-Tree Construction



We can keep splitting the points in each set to create a tree structure. Each node with no children (leaf node) contains a list of points.

KD-Tree Construction



We will keep around one additional piece of information at each node. The (tight) bounds of the points at or below this node.

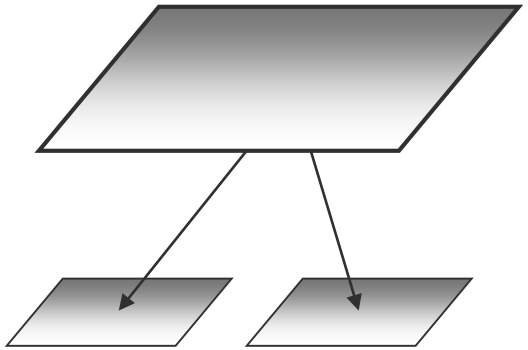
KD-Tree Construction

Use heuristics to make splitting decisions:

- Which dimension do we split along?
  - Widest
- Which value do we split at?
  - Median of value of that split dimension for the points.
- When do we stop?
  - When there are fewer then m points left OR the box has hit some minimum width.

Exclusion and inclusion

using point-node *kd*-tree bounds.



O(D) bounds on distance minima/maxima:

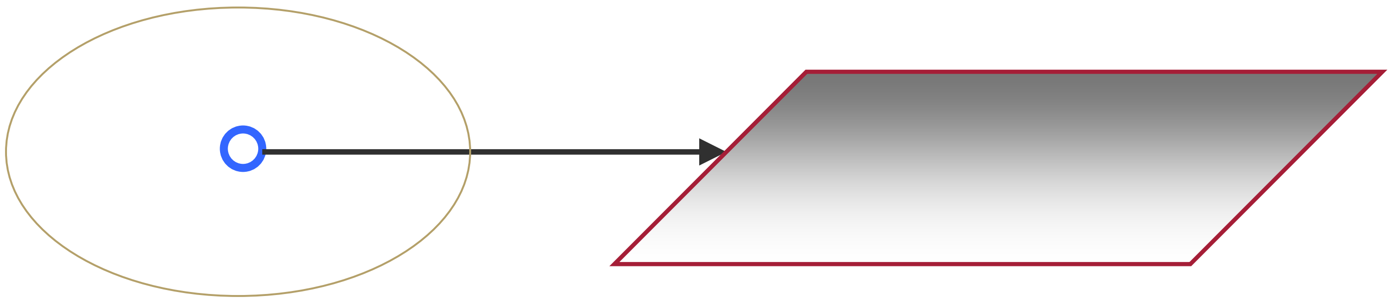
$$\min_i \|x - x_i\| \geq \sum_d^D \left[\max\left\{(l_d - x_d)^2, 0\right\} + \max\left\{(x_d - u_d)^2, 0\right\}\right]$$

$$\max_i \|x - x_i\| \leq \sum_d^D \max\left\{(u_d - x_d)^2, (x_d - l_d)^2\right\}$$

Exclusion and inclusion

using <u>point-node</u> *kd*-tree bounds.

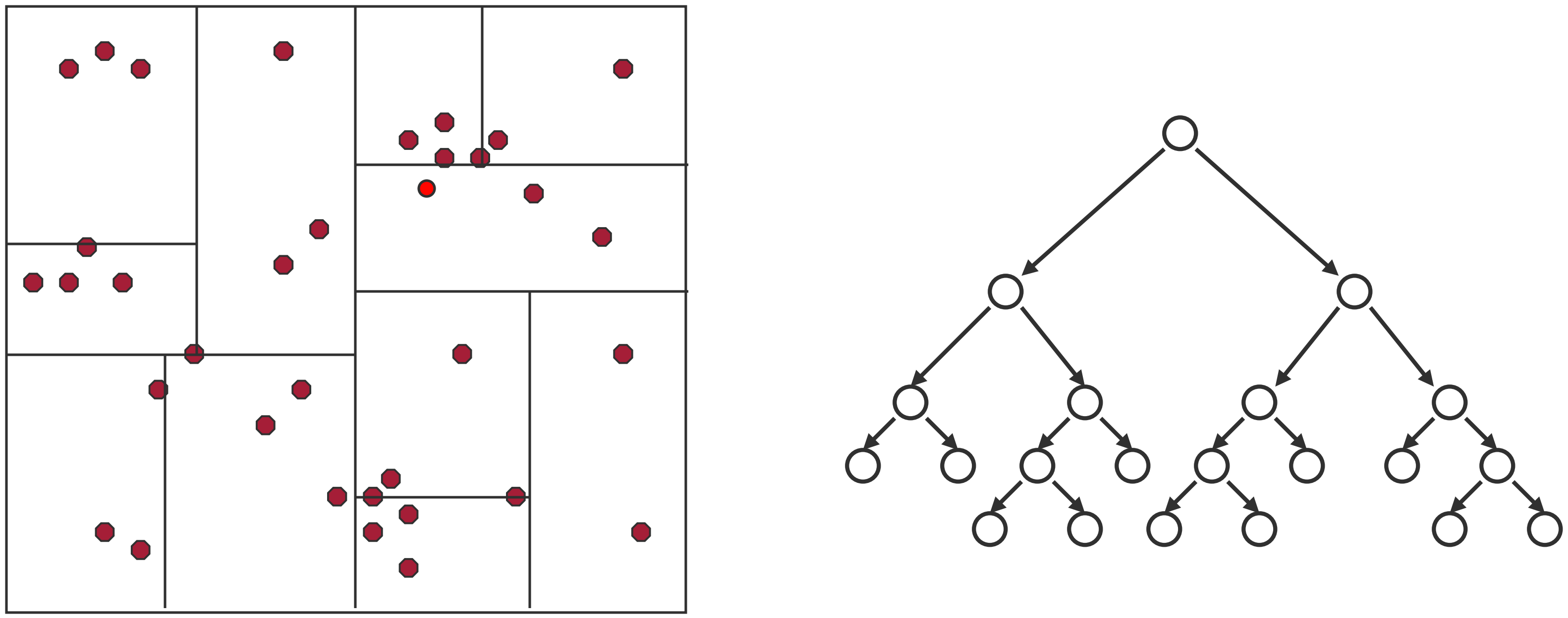
O(D) bounds on distance minima/maxima:



$$\min_i \|x - x_i\| \geq \sum_d^D \left[\max\left\{ (l_d - x_d)^2, 0 \right\} + \max\left\{ (x_d - u_d)^2, 0 \right\} \right]$$

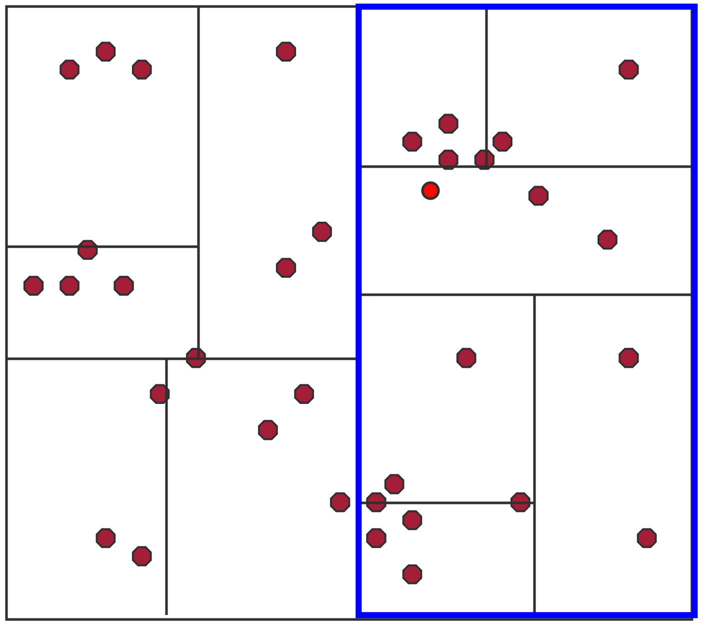
$$\max_i \|x - x_i\| \leq \sum_d^D \max\left\{ (u_d - x_d)^2, (x_d - l_d)^2 \right\}$$

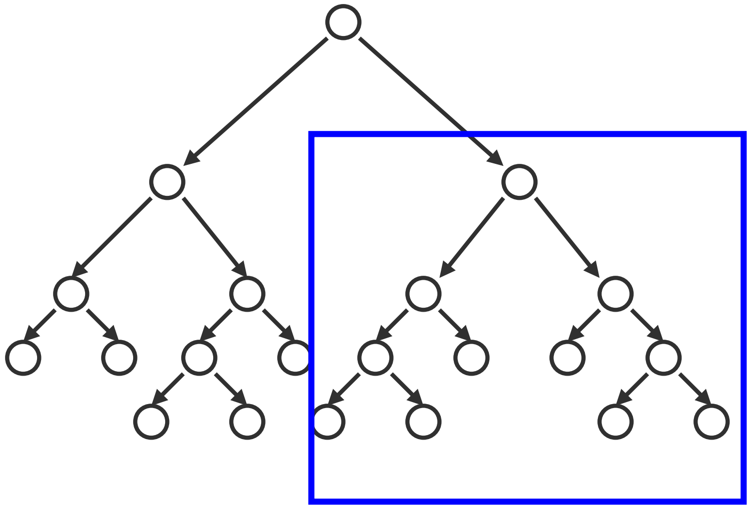
Nearest Neighbor with KD Trees



We traverse the tree looking for the nearest neighbor of the query point.

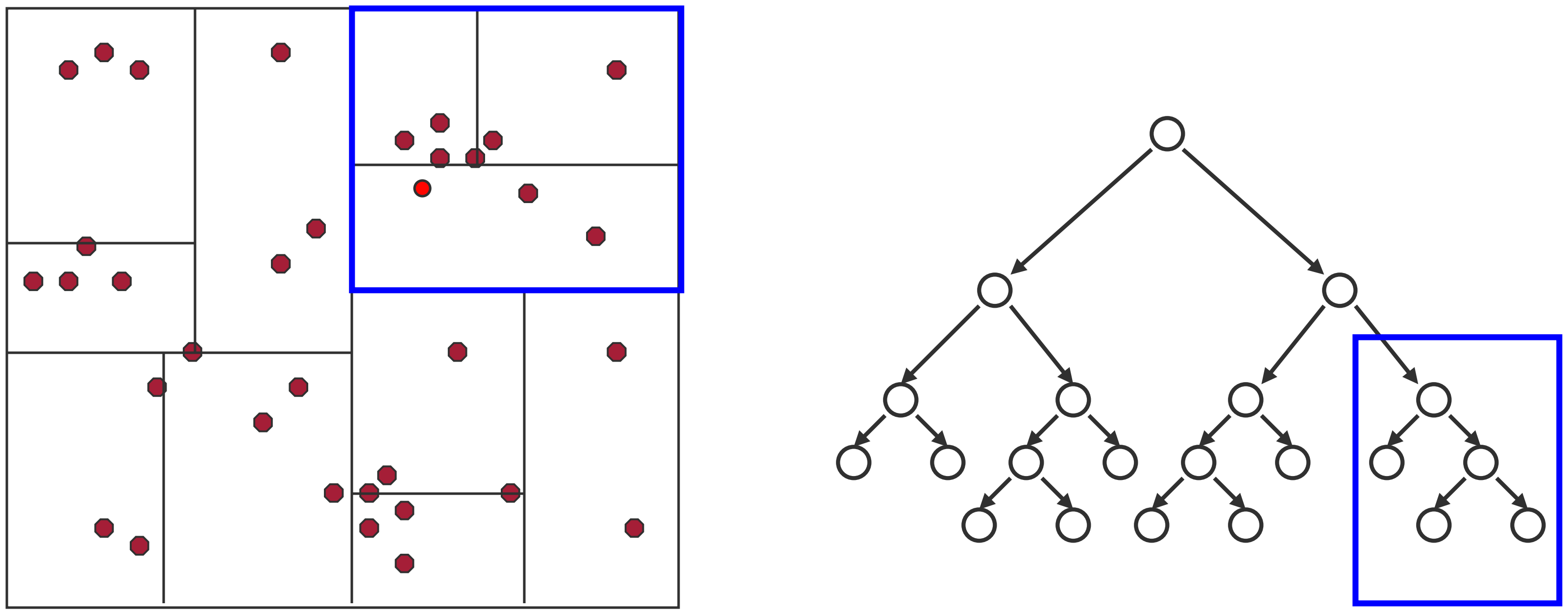
Nearest Neighbor with KD Trees





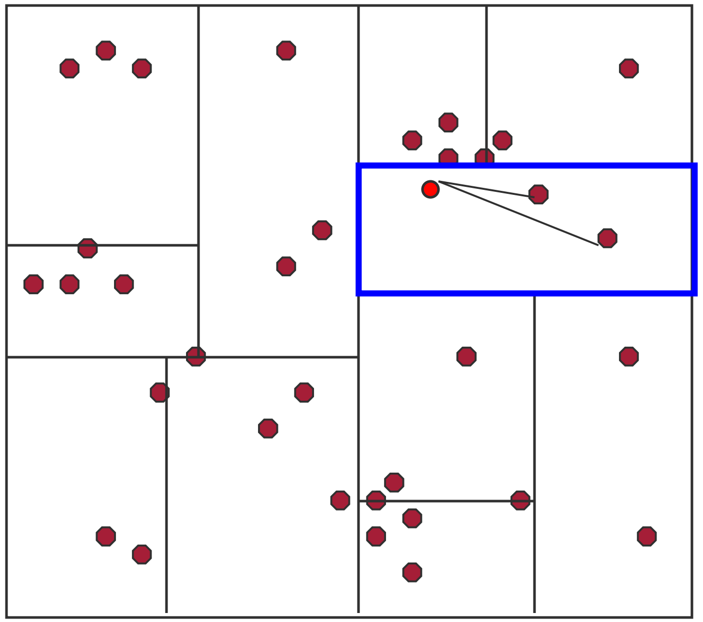
Examine nearby points first: Explore the branch of the tree that is closest to the query point first.

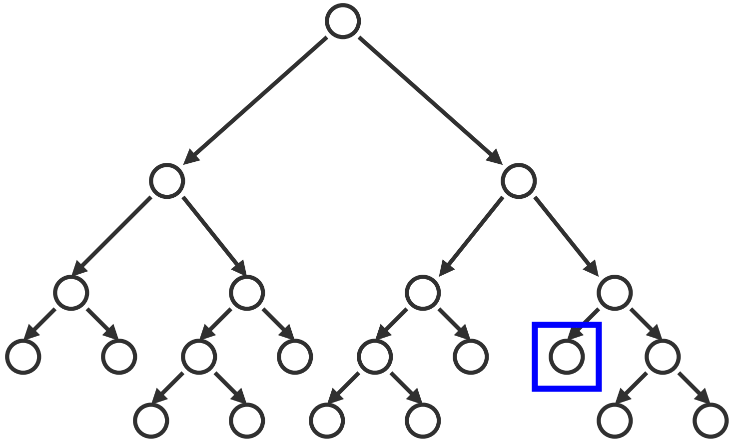
Nearest Neighbor with KD Trees



Examine nearby points first: Explore the branch of the tree that is closest to the query point first.

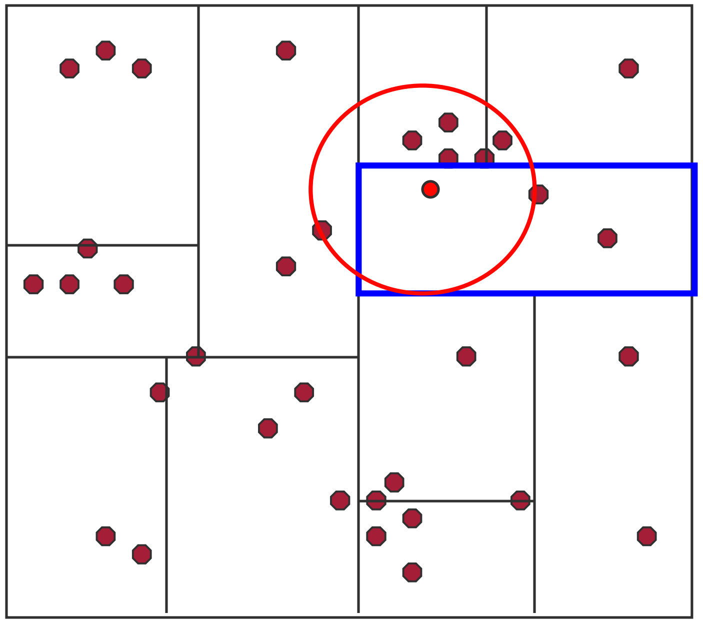
Nearest Neighbor with KD Trees

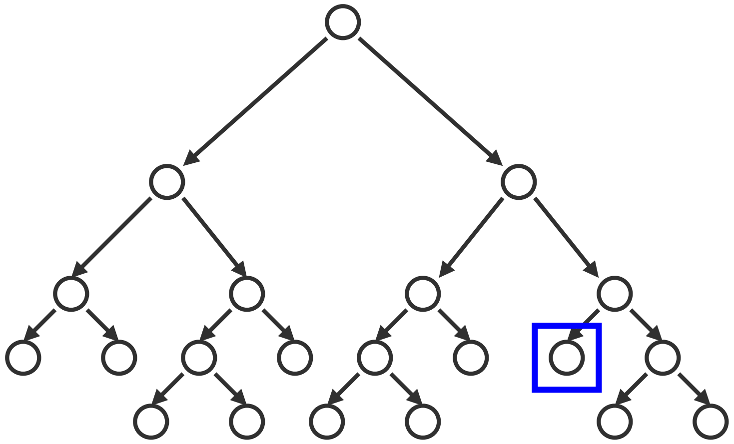




When we reach a leaf node: compute the distance to each point in the node.

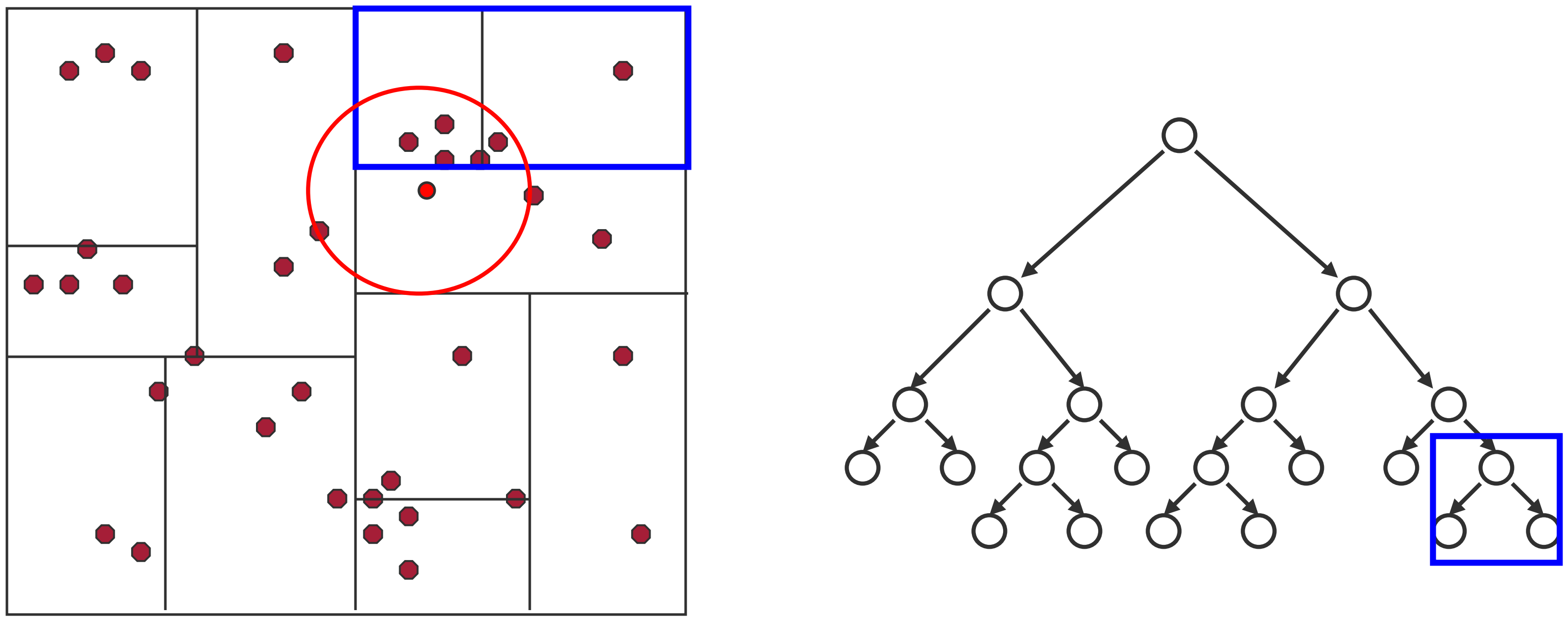
Nearest Neighbor with KD Trees





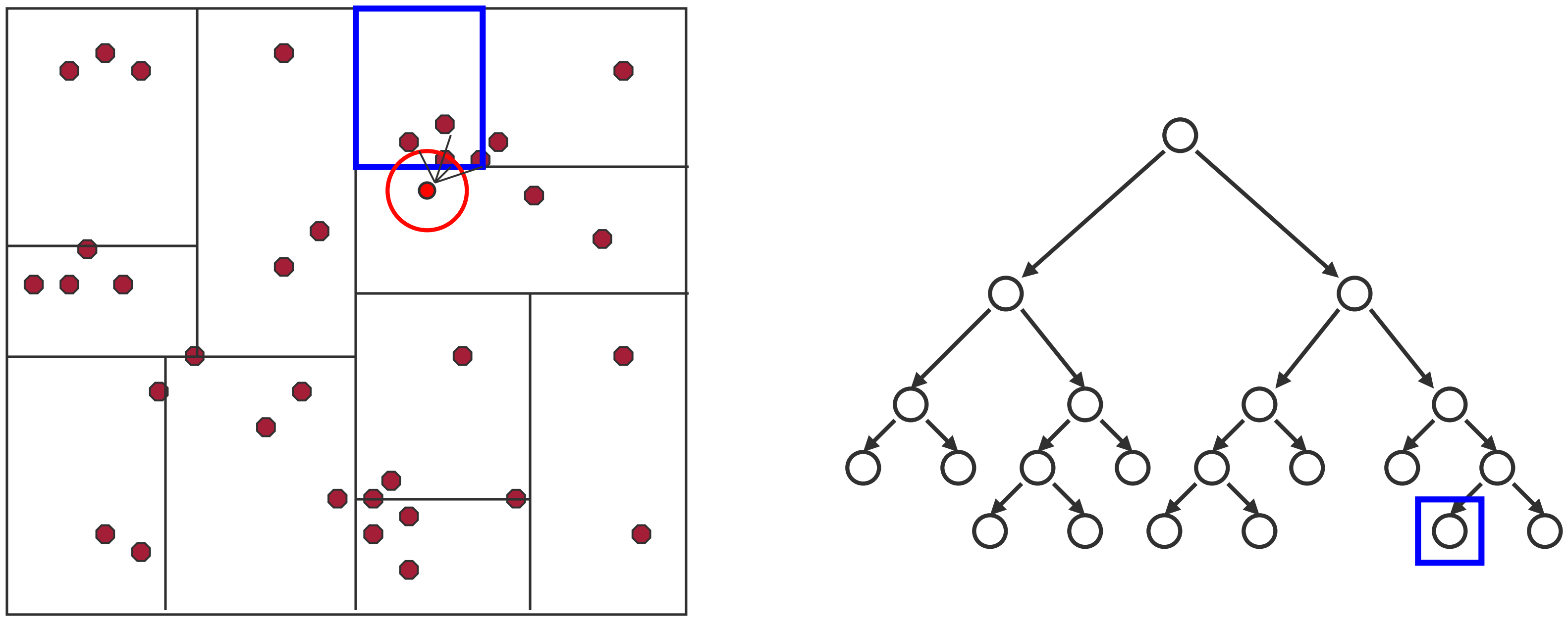
When we reach a leaf node: compute the distance to each point in the node.

Nearest Neighbor with KD Trees



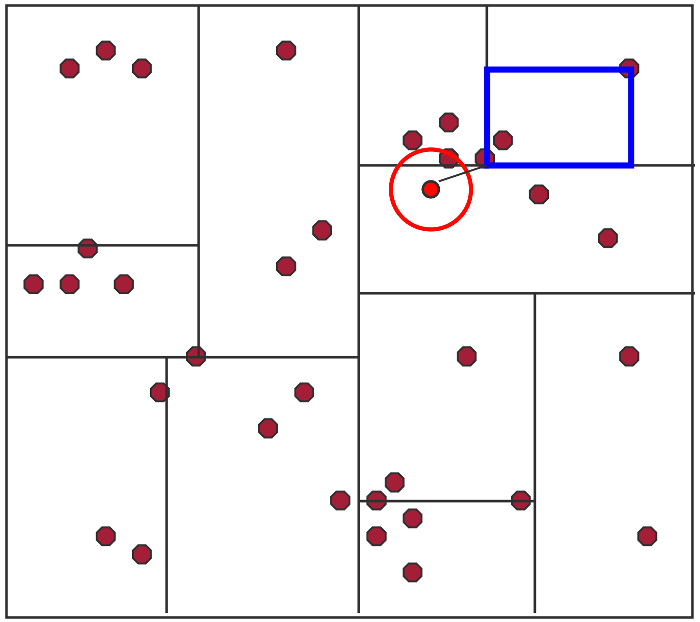
Then we can backtrack and try the other branch at each node visited.

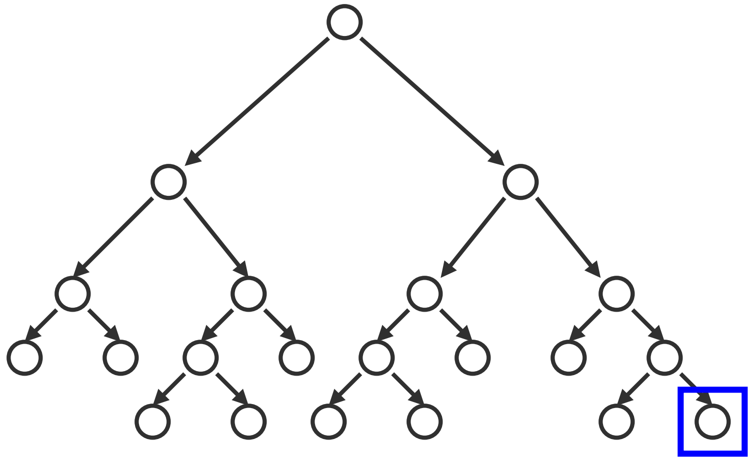
Nearest Neighbor with KD Trees



Each time a new closest node is found, we can update the distance bounds.

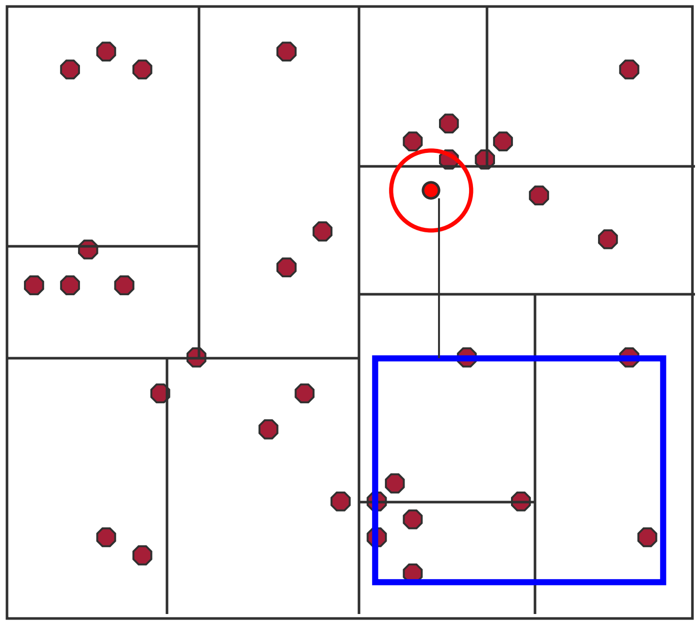
Nearest Neighbor with KD Trees

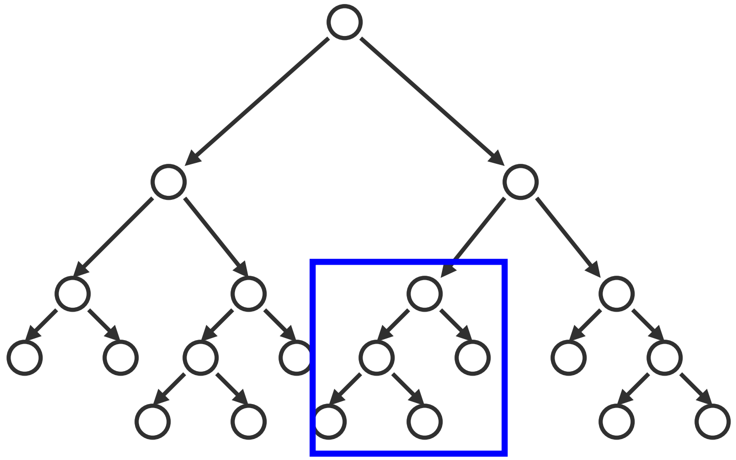




Using the distance bounds and the bounds of the data below each node, we can prune parts of the tree that could NOT include the nearest neighbor.

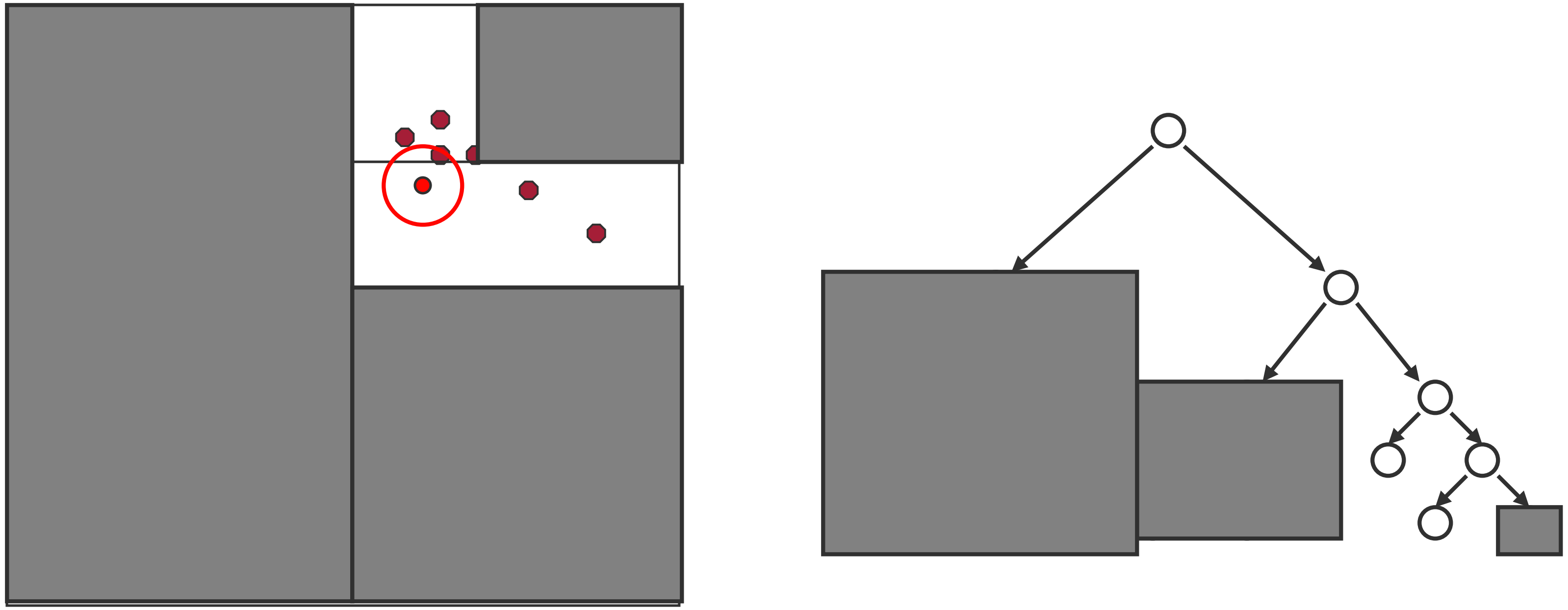
Nearest Neighbor with KD Trees





Using the distance bounds and the bounds of the data below each node, we can prune parts of the tree that could NOT include the nearest neighbor.

Nearest Neighbor with KD Trees



Using the distance bounds and the bounds of the data below each node, we can prune parts of the tree that could NOT include the nearest neighbor.

Simple Recursive Algorithm

- (k=1 case)

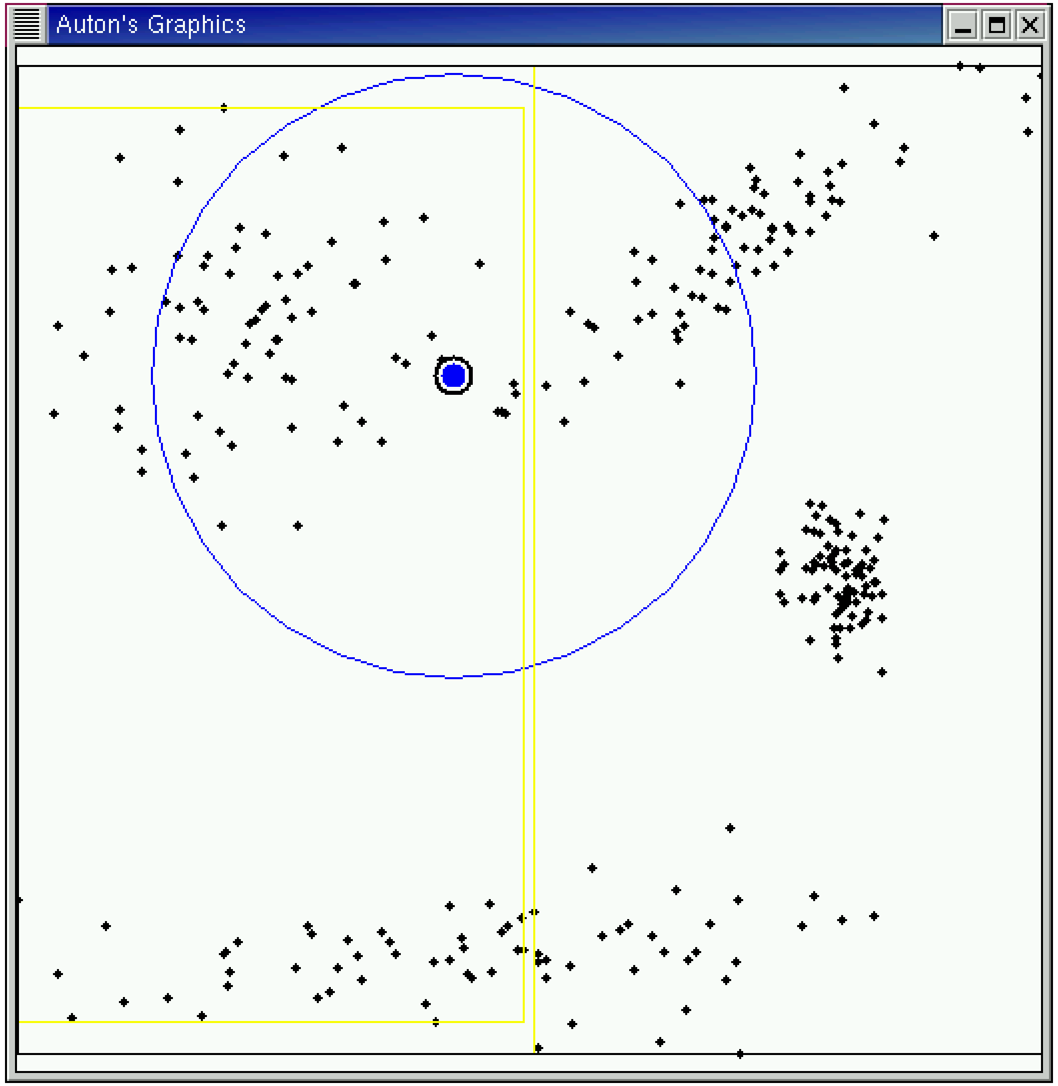
NN(x_q,R,d_{lo},x_{sofar},d_{sofar})
{
  if $d_{lo} > d_{sofar}$, return.

  if leaf(R), [x_{sofar},d_{sofar}]=**NNBase**(x_q,R,d_{sofar}).
  else,
    [R1,d1,R2,d2]=orderByDist(x_q,R.l,R.r).
    NN(x_q,R1,d1,x_{sofar},d_{sofar}).
    NN(x_q,R2,d2,x_{sofar},d_{sofar}).
}

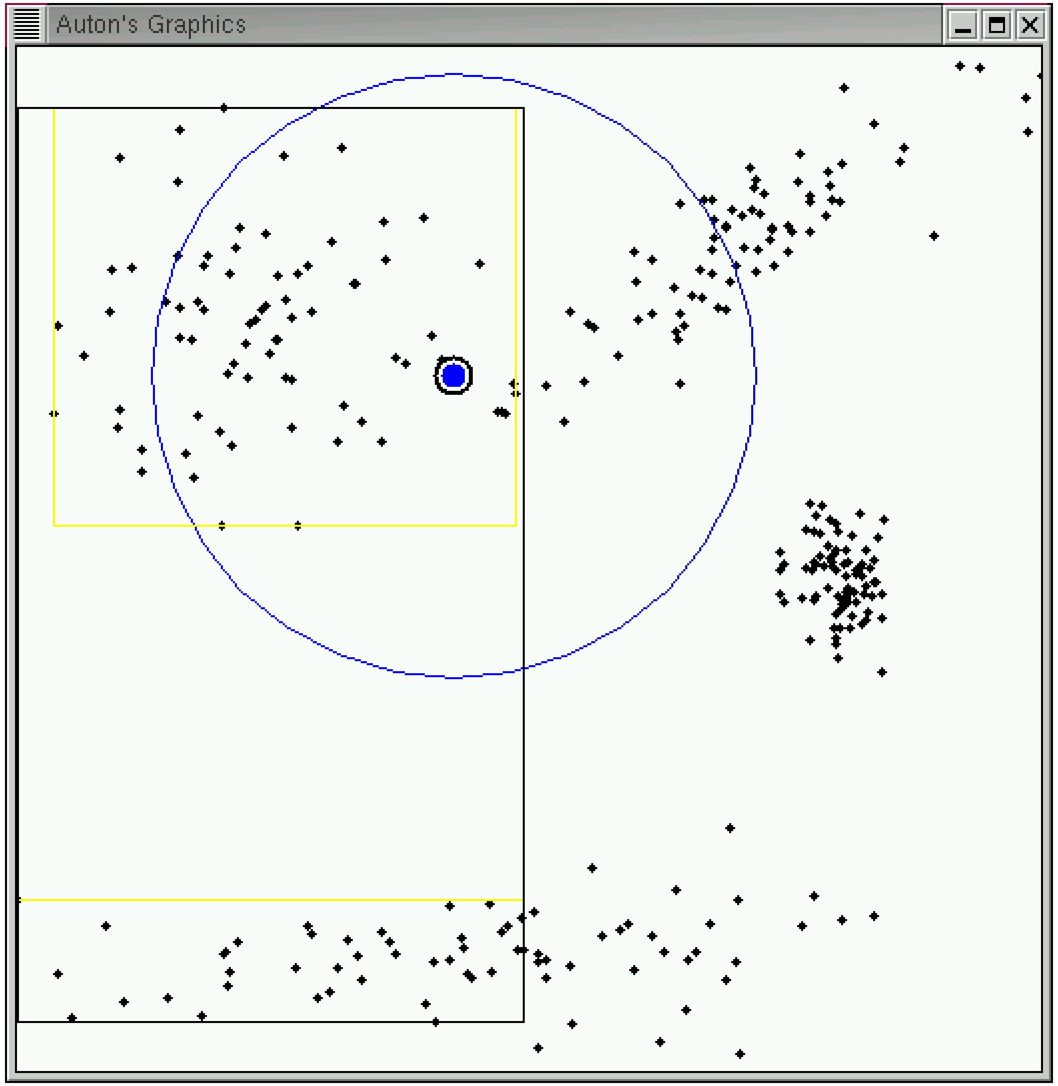
Range Queries

(all Points within Radius)

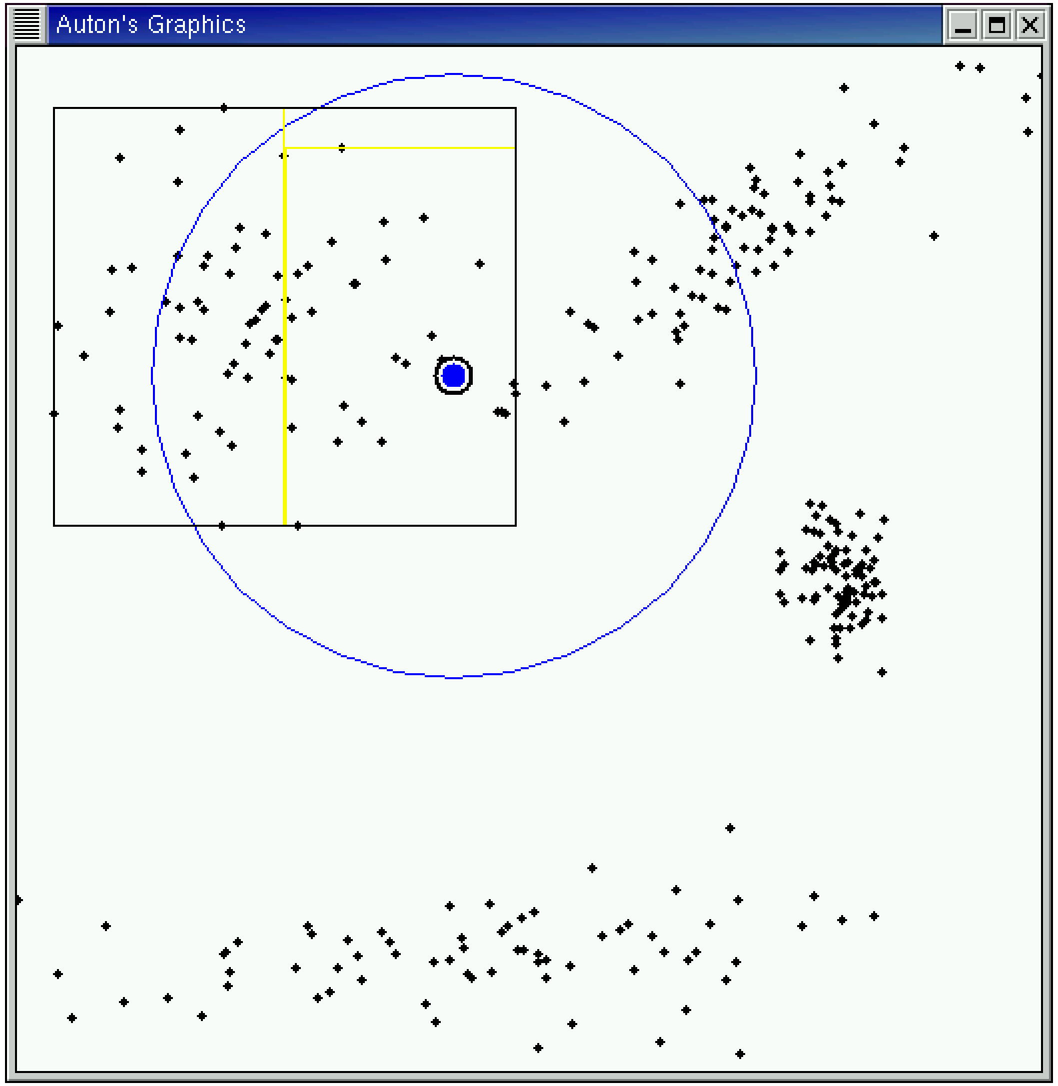
Range-count Example



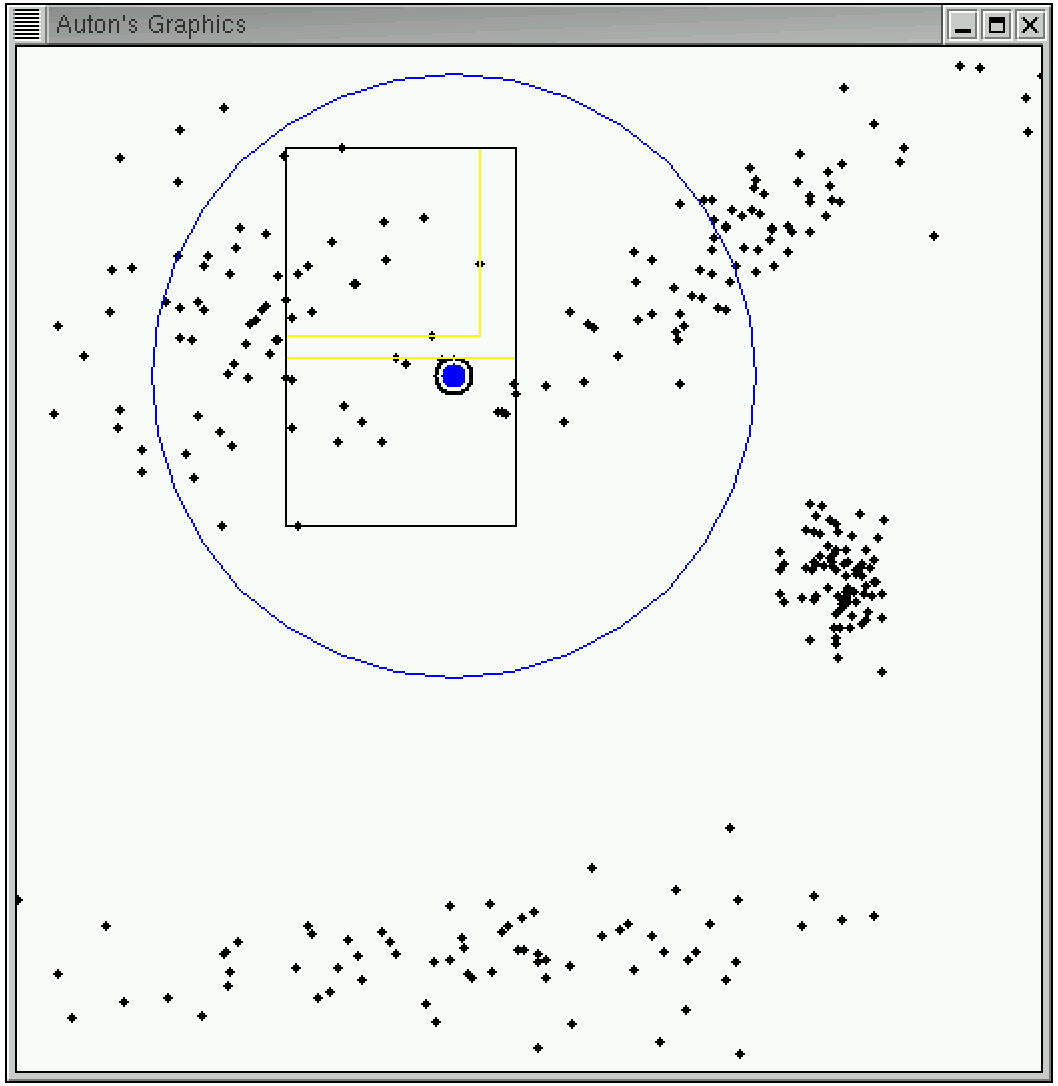
Range-count example



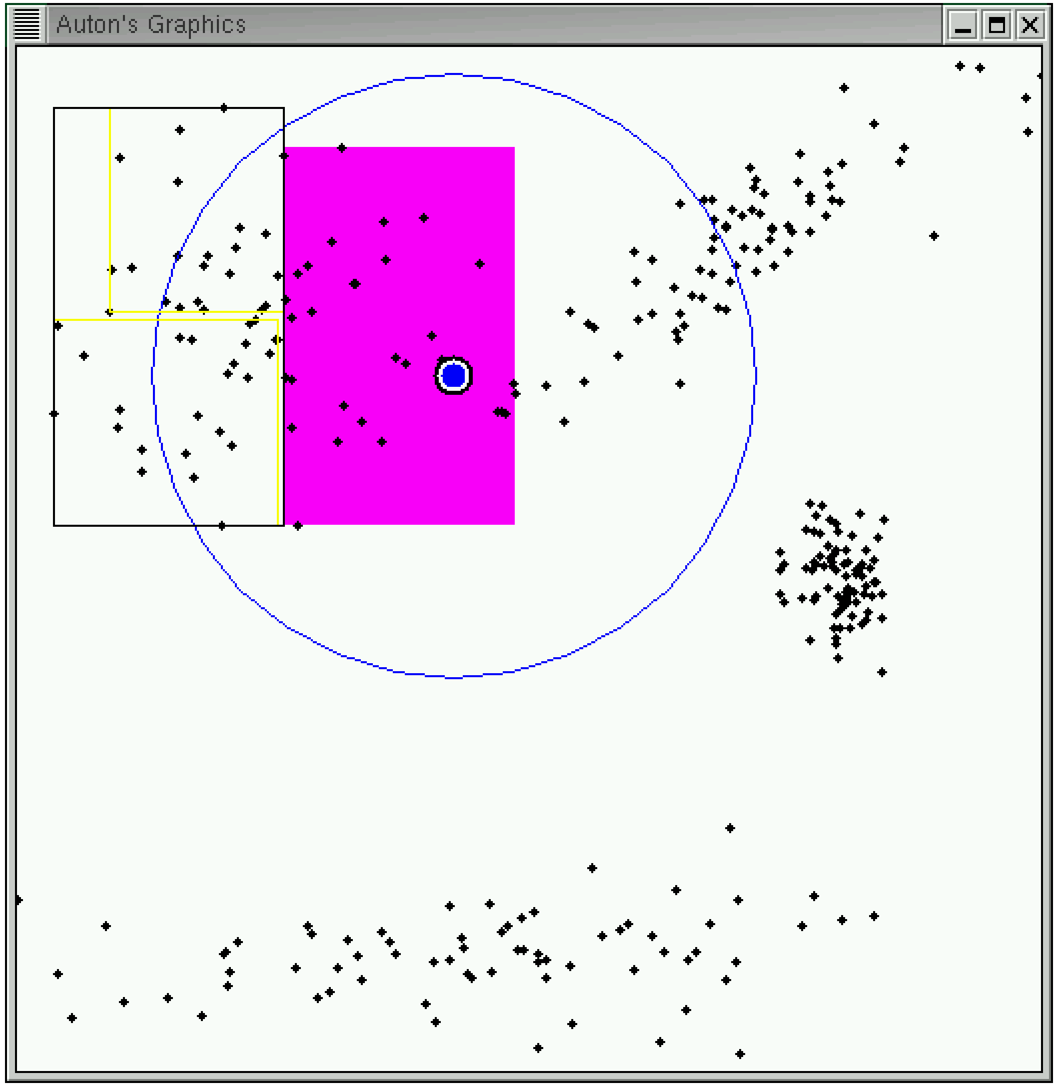
Range-count example



Range-count example

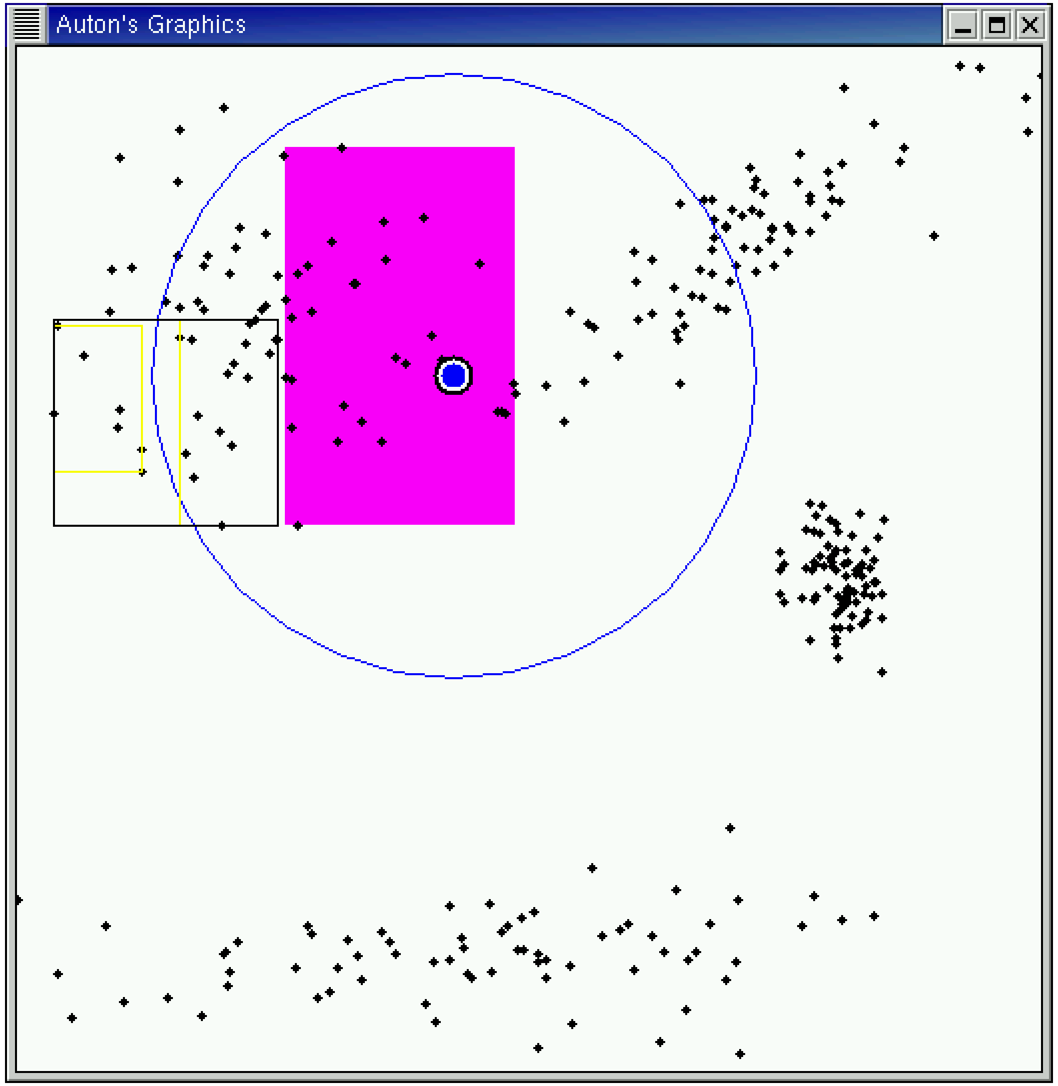


Range-count example

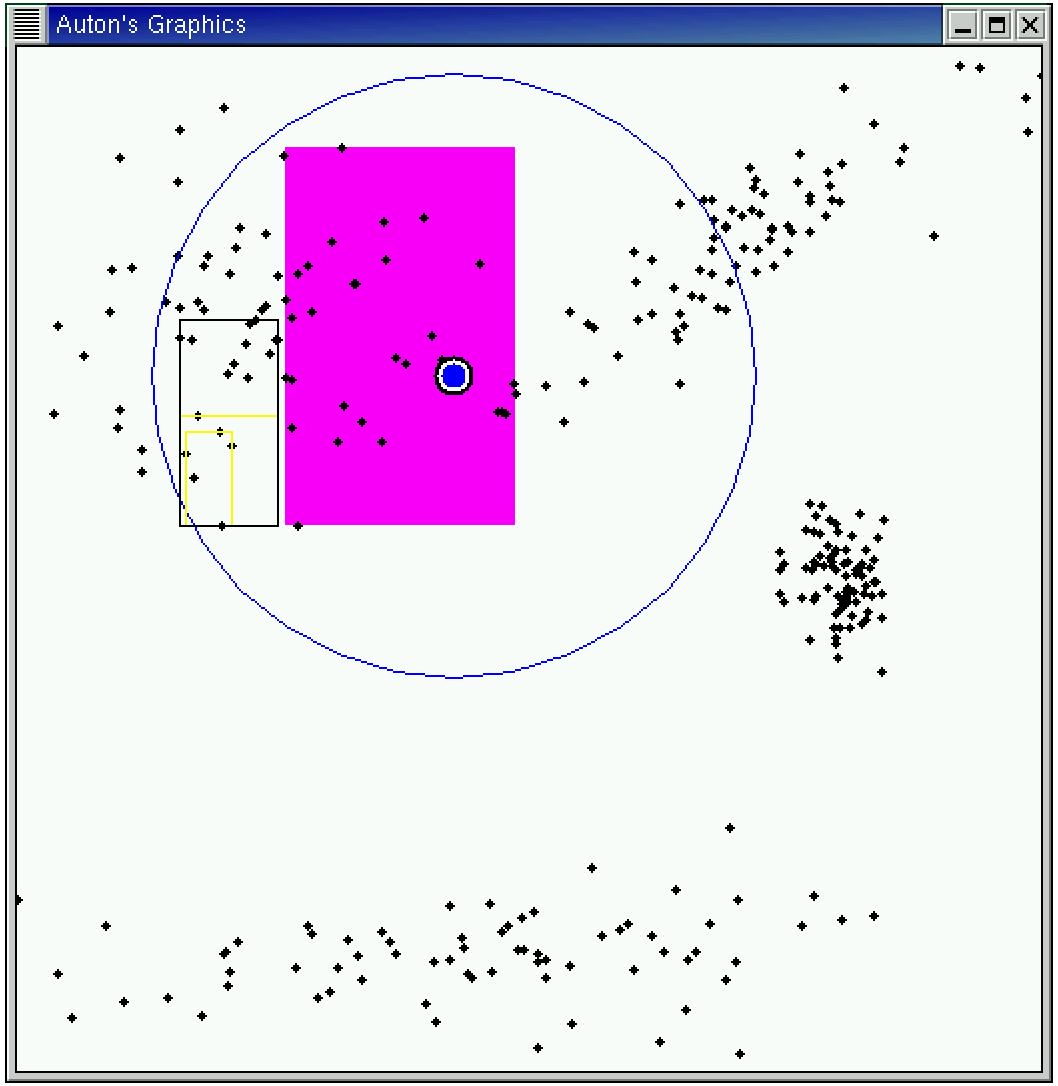


Pruned!
(inclusion)

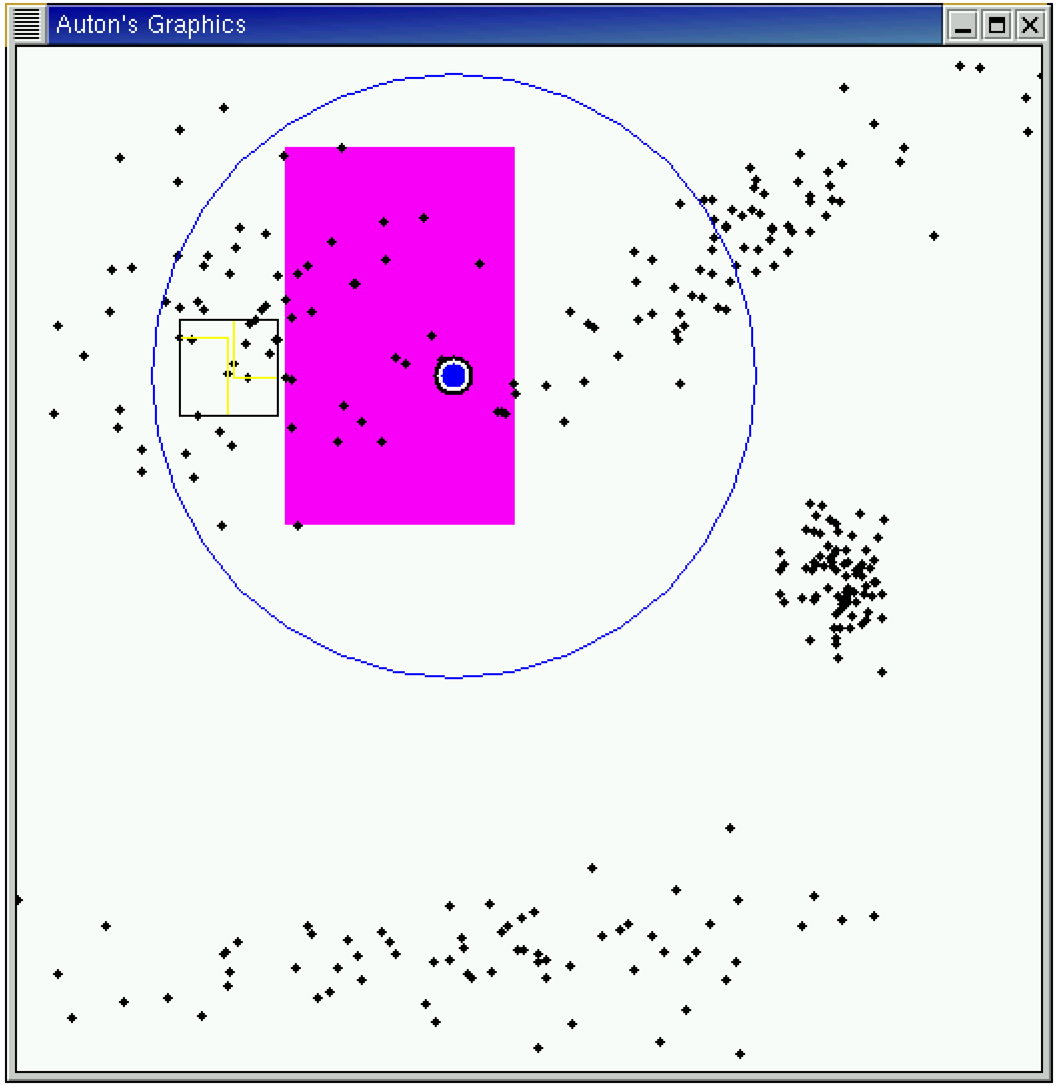
Range-count example



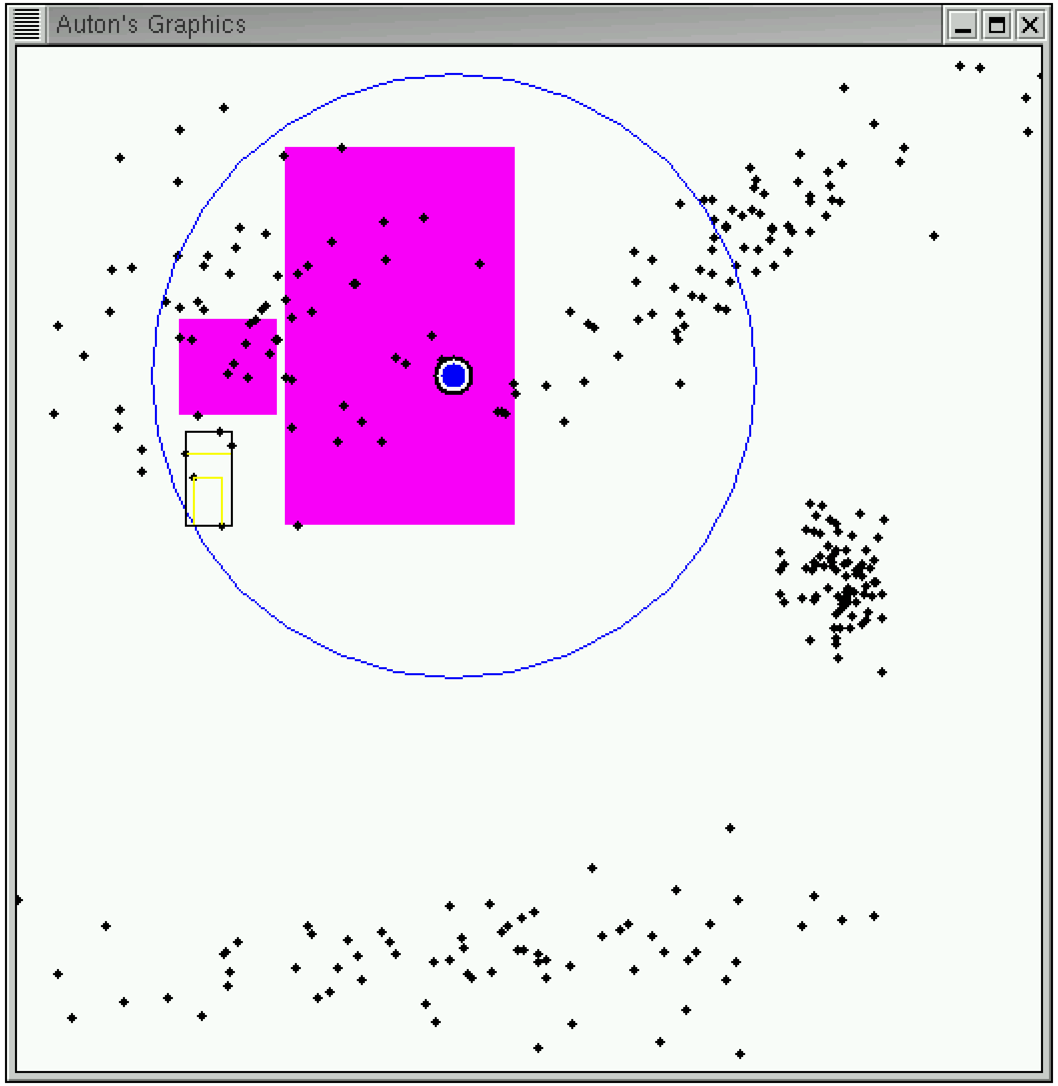
Range-count example



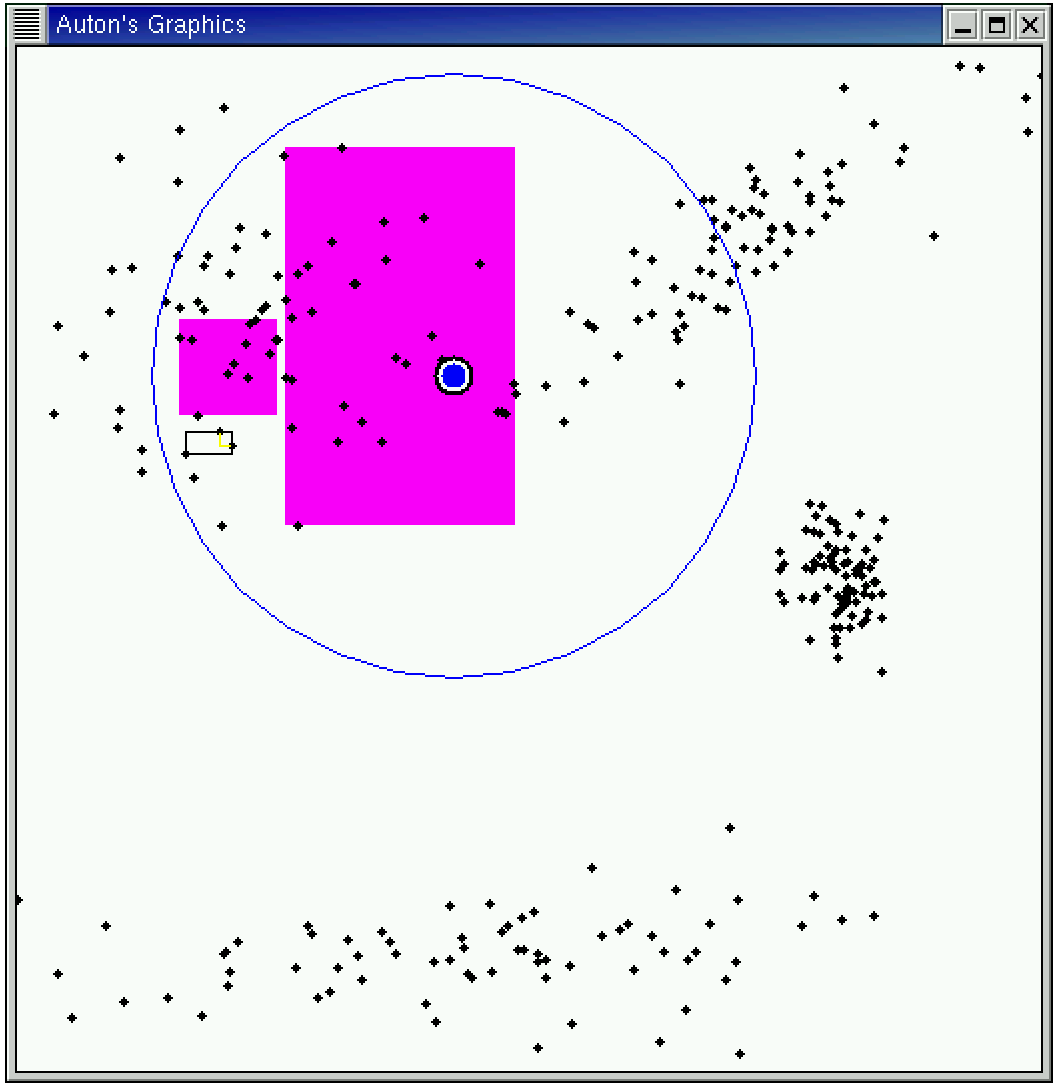
Range-count example



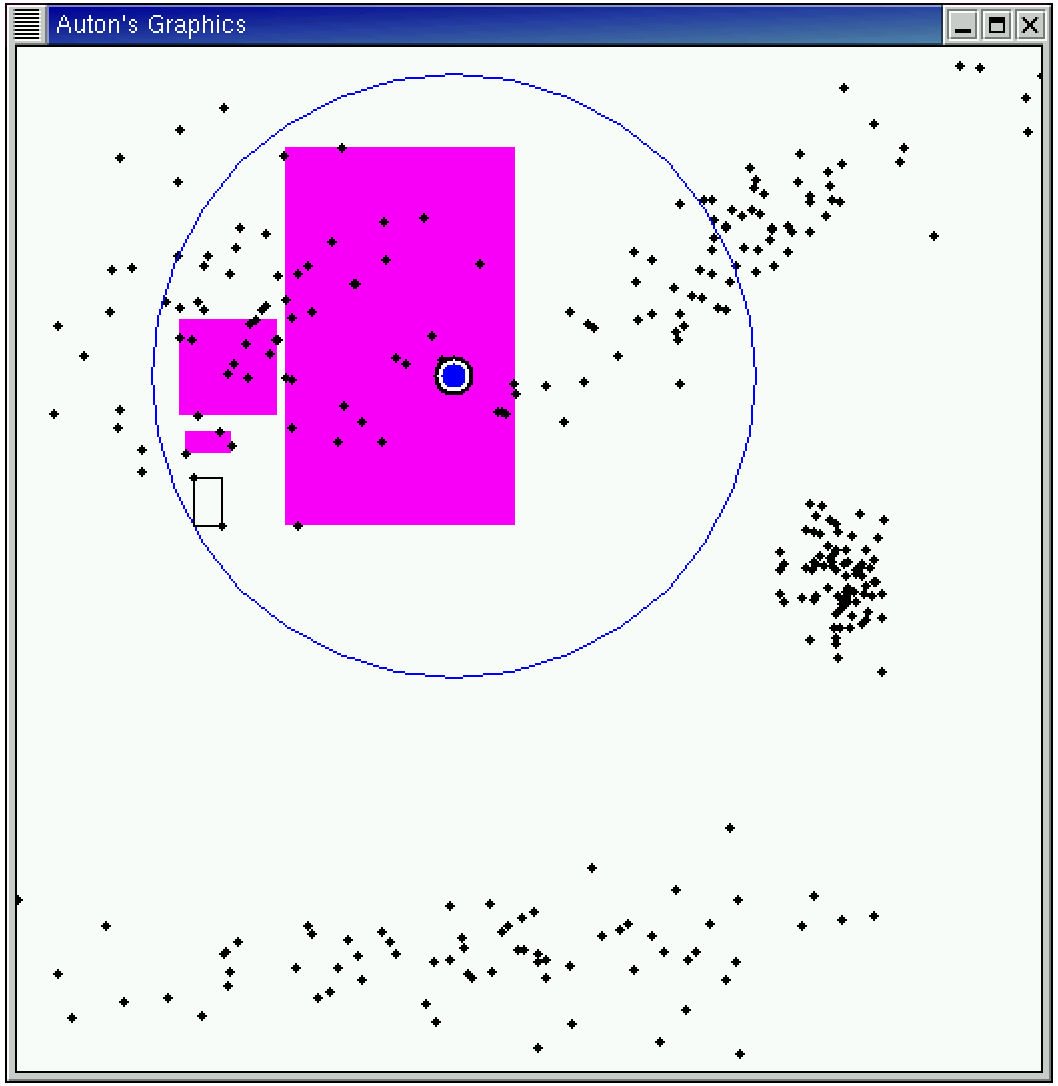
Range-count example



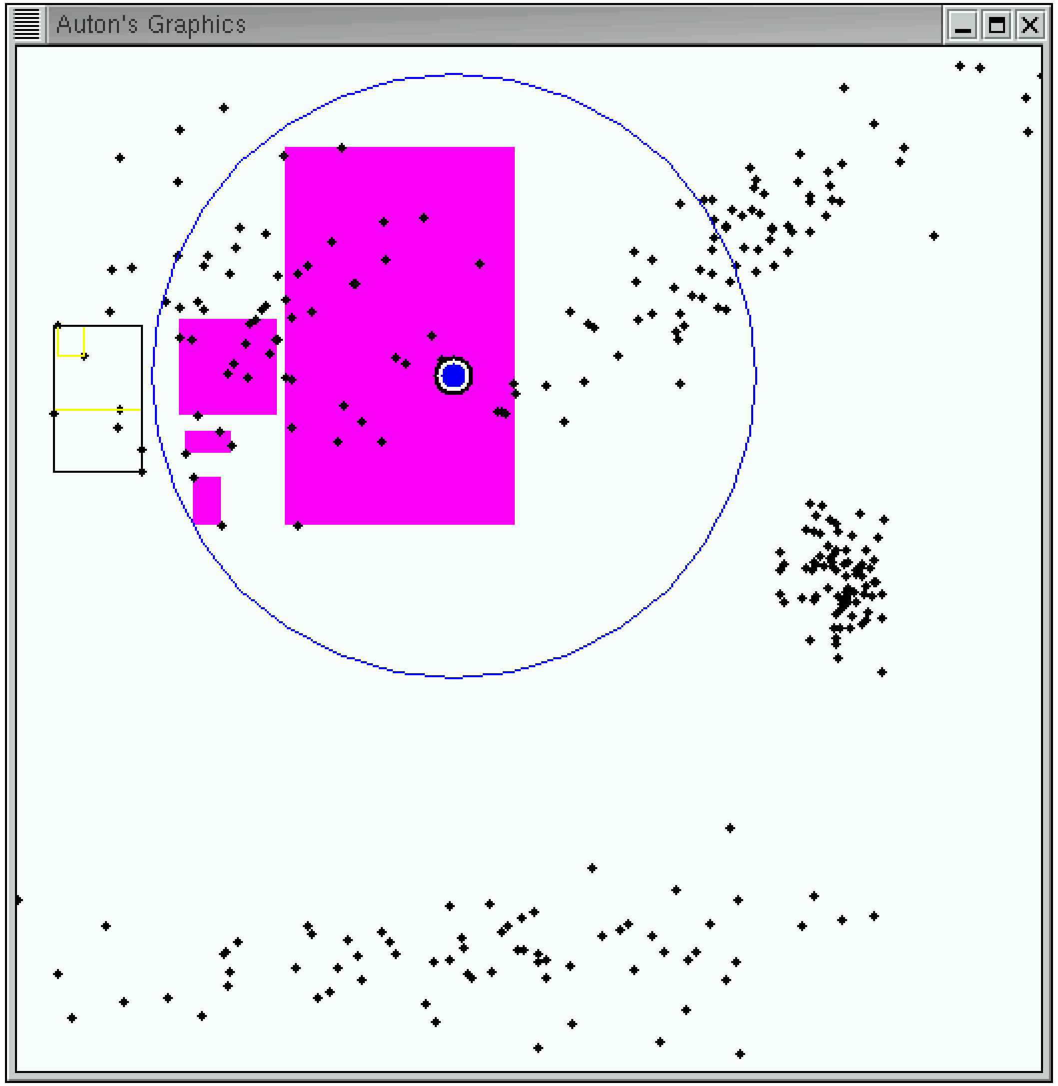
Range-count example



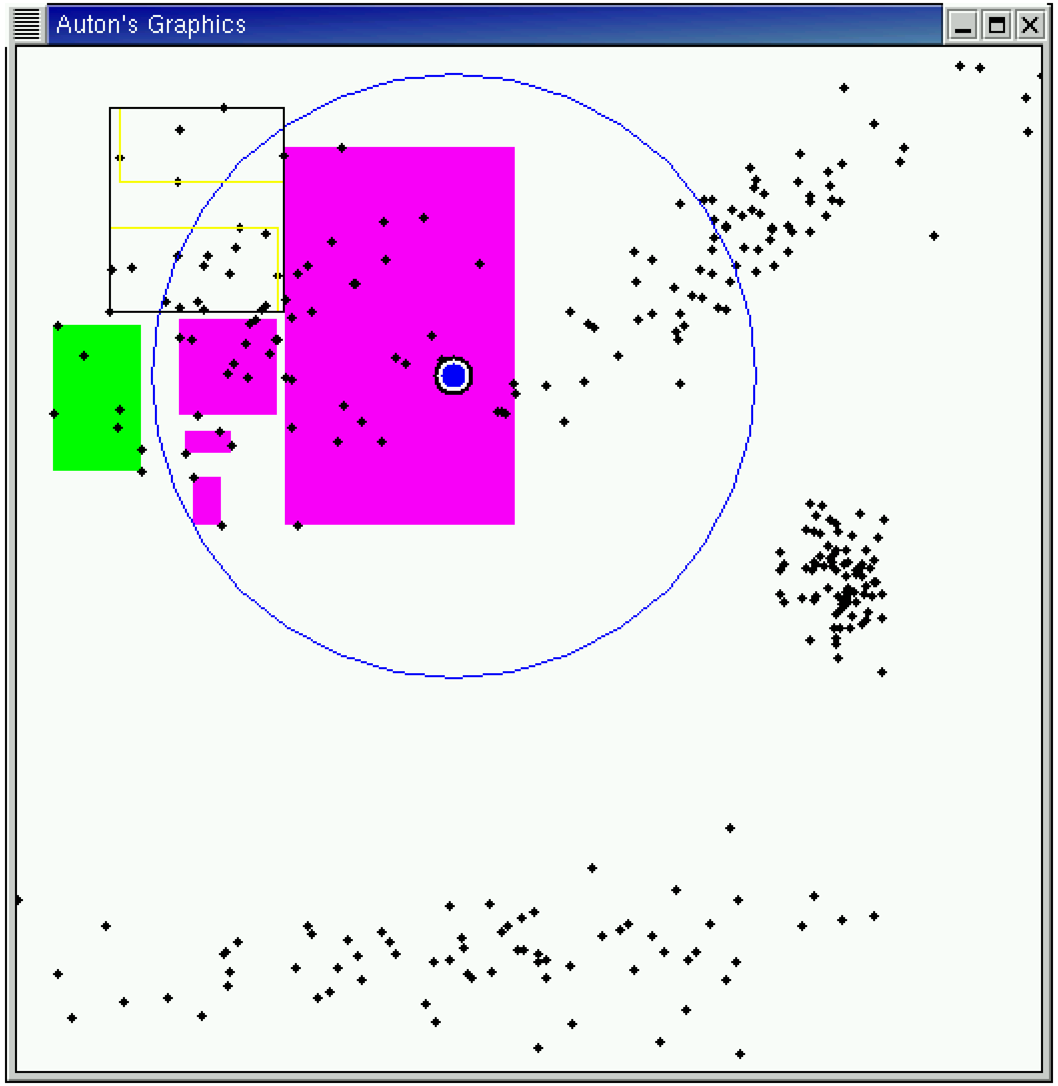
Range-count example



Range-count example

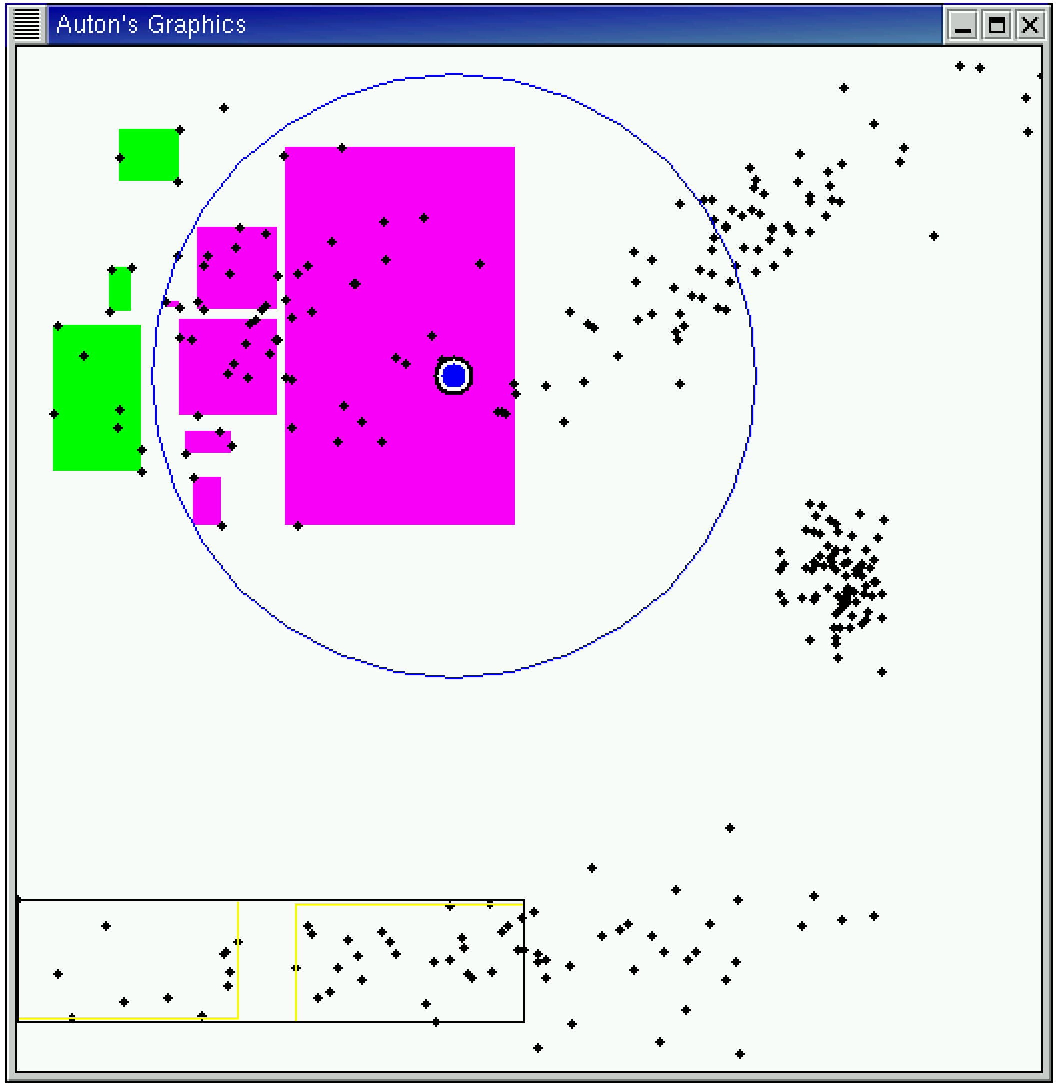


Range-count example

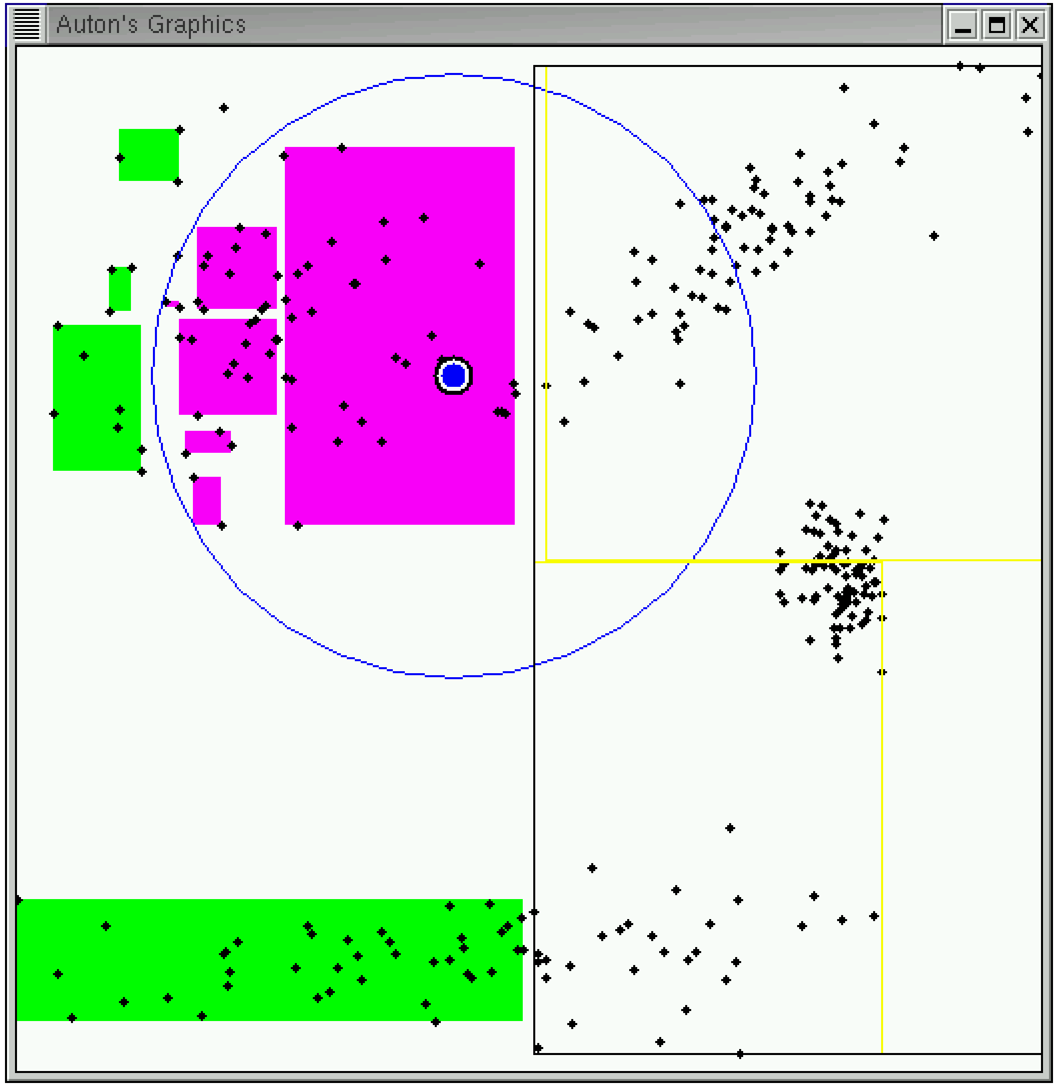


Pruned!
(exclusion)

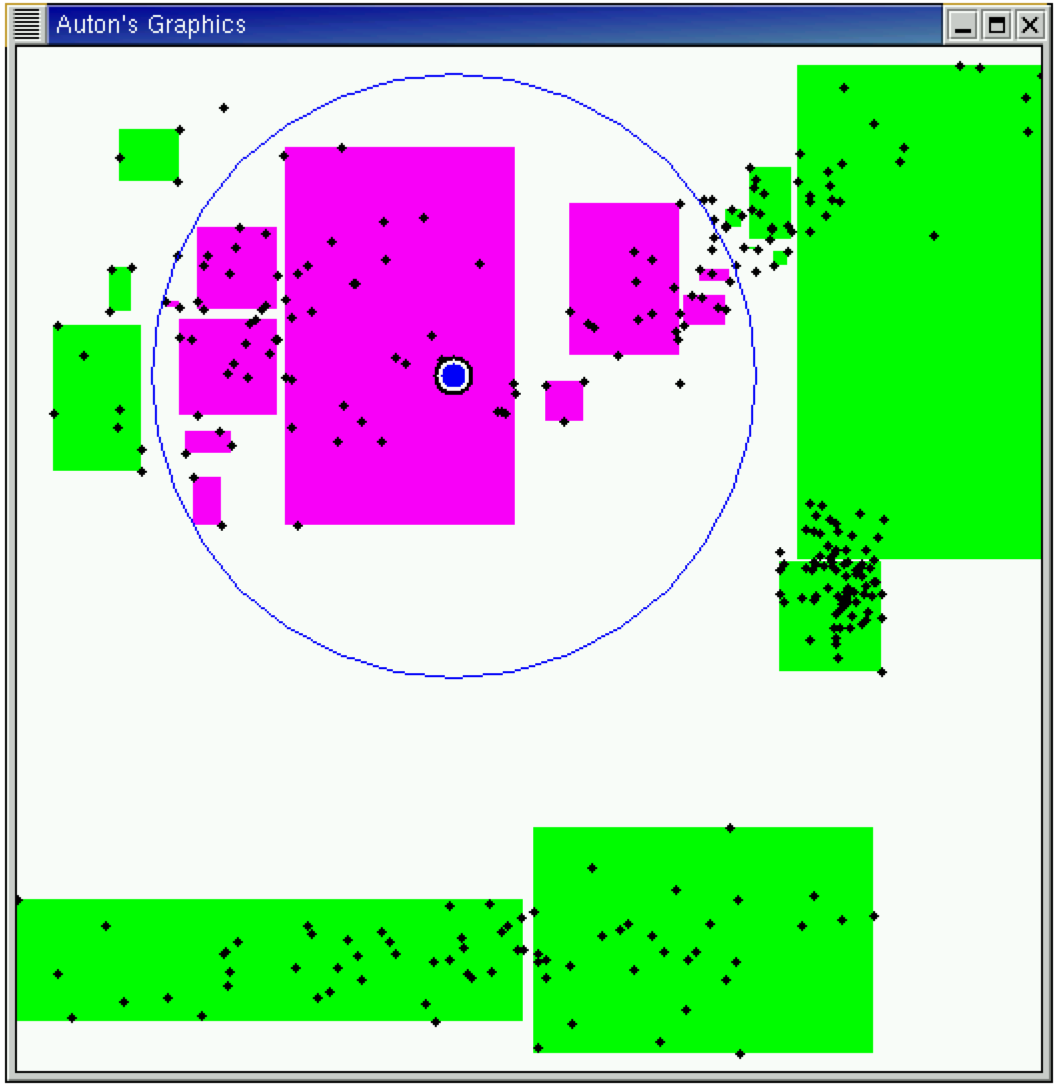
Range-count example



Range-count example



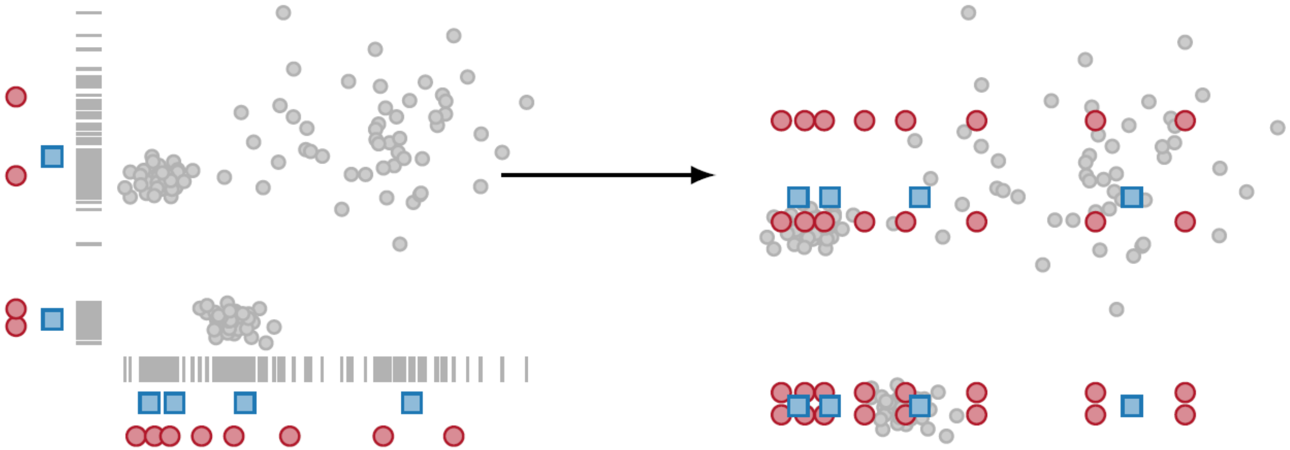
Range-count example



Some questions

- Asymptotic runtime analysis?
  - In a rough sense, O(logN)
  - But only under some regularity conditions

- How high in dimension can we go?
  - Roughly exponential in intrinsic dimension
  - In practice, in less than 100 dimensions, still big speedups



Product Quantization Trees for Approximate Nearest Neighbor Search

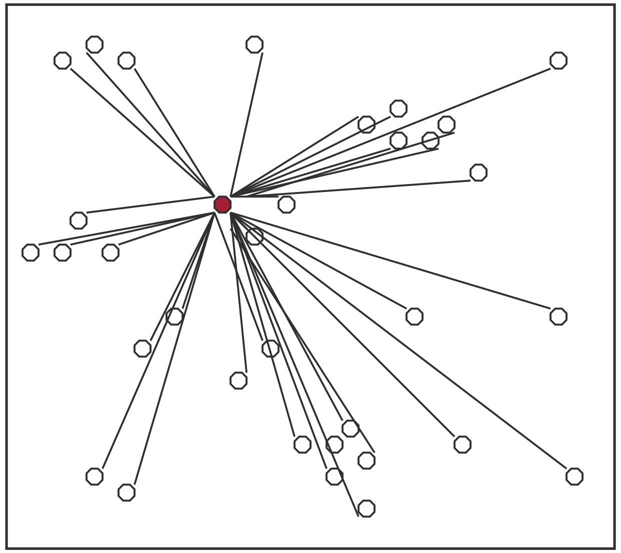
[Wieschollek, Wang, Sorkine-Hornung, Lensch – CVPR16]

High-Dimensional kNN – k Nearest Neighbor Search

- Simple kd-tree good for small dimensions
  D < 20

- For large dimensions **D > 20** brute force often as good as simple trees

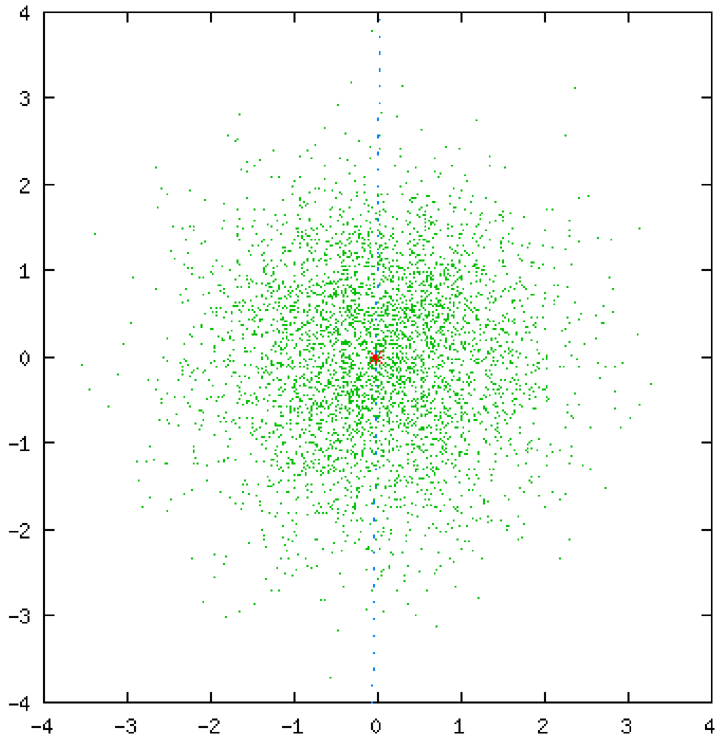
- Existing fast approximations
  - randomized kd-forests
  - local sensitive hashing (LSH)
  - k-means trees
  - product quantization



33 Distance Computations

Vector Quantization (k-means)

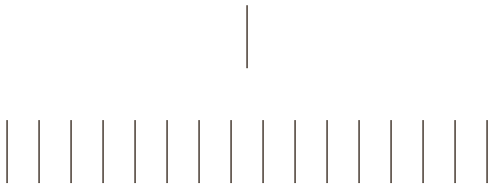
- Represent all data vectors by the nearest cluster representative
- K-means clustering



[http://www.data-compression.com/vqanim.shtml]

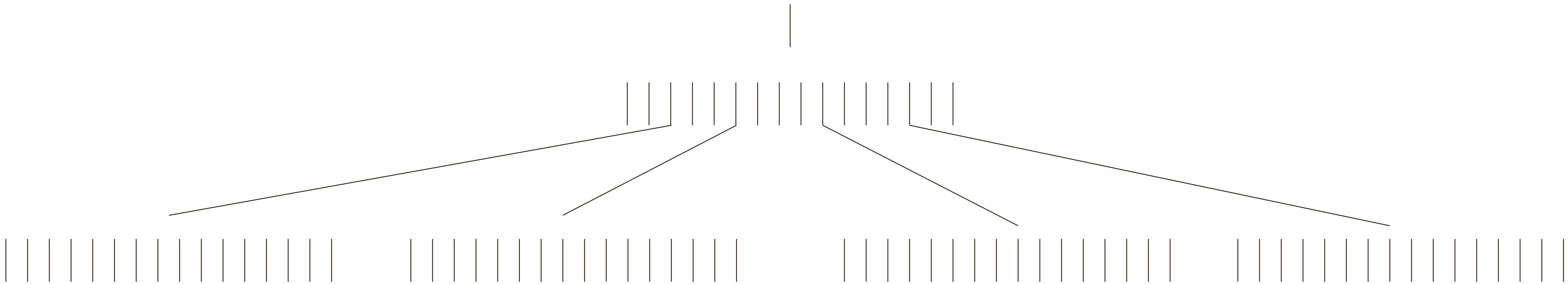
K-Means-Trees

- Huge branching-factor in each node of the tree



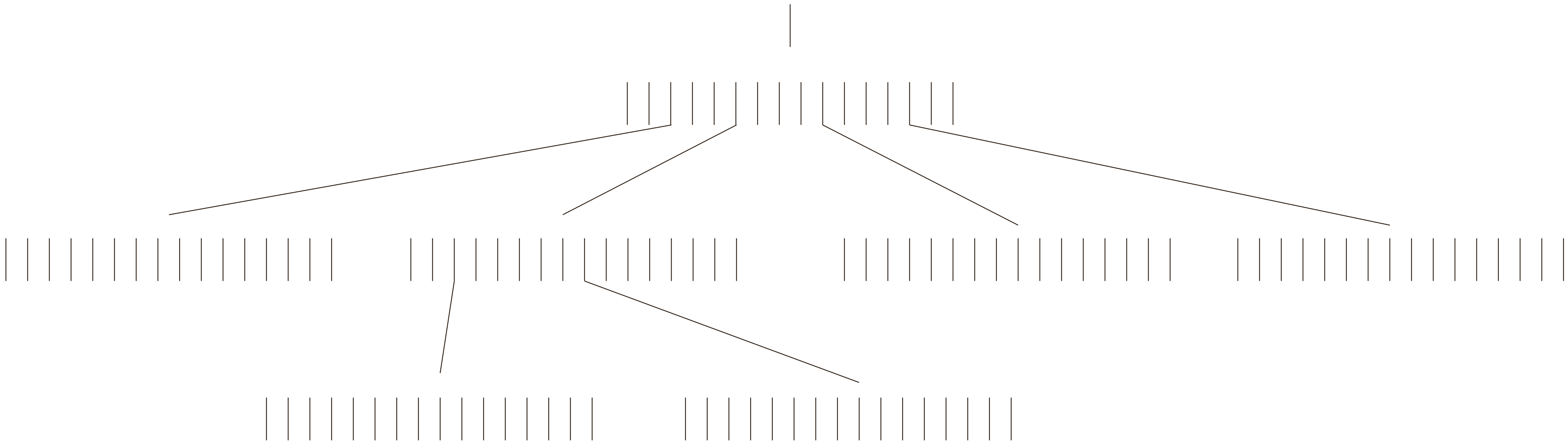
K-Means-Trees

- Huge branching-factor in each node of the tree



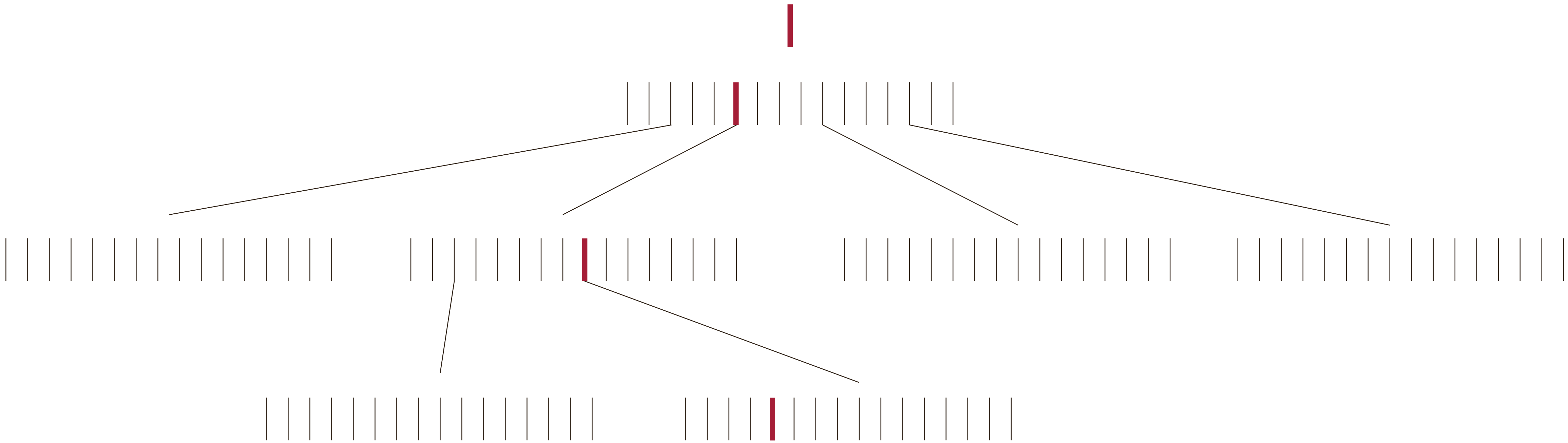
K-Means-Trees

- Huge branching-factor in each node of the tree



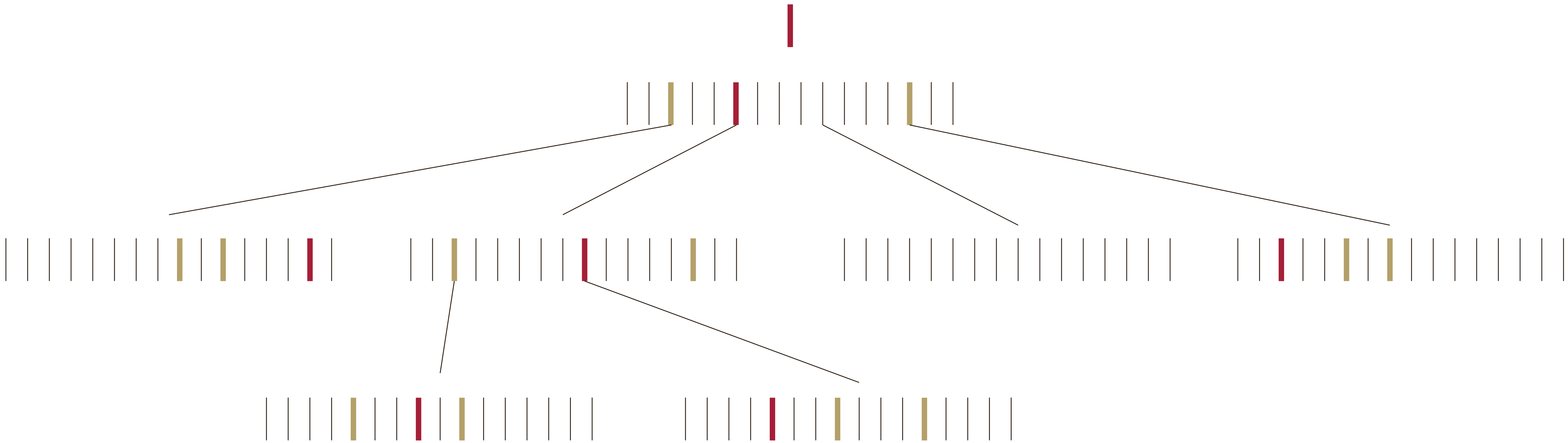
K-Means-Trees - Lookup

- Visit nearest bin (calc distances to all center clusters)
- And potentially near-by bins



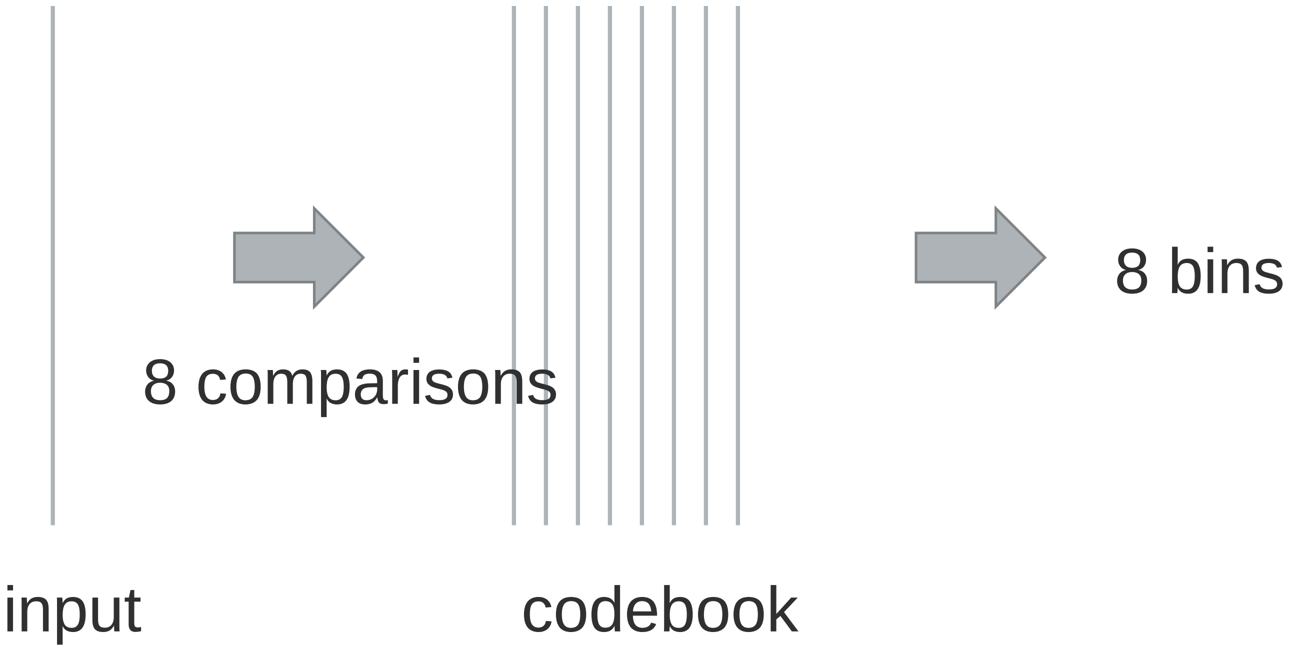
K-Means-Trees - Lookup

- Visit nearest bin (calc distances to all center clusters)
- And potentially near-by bins



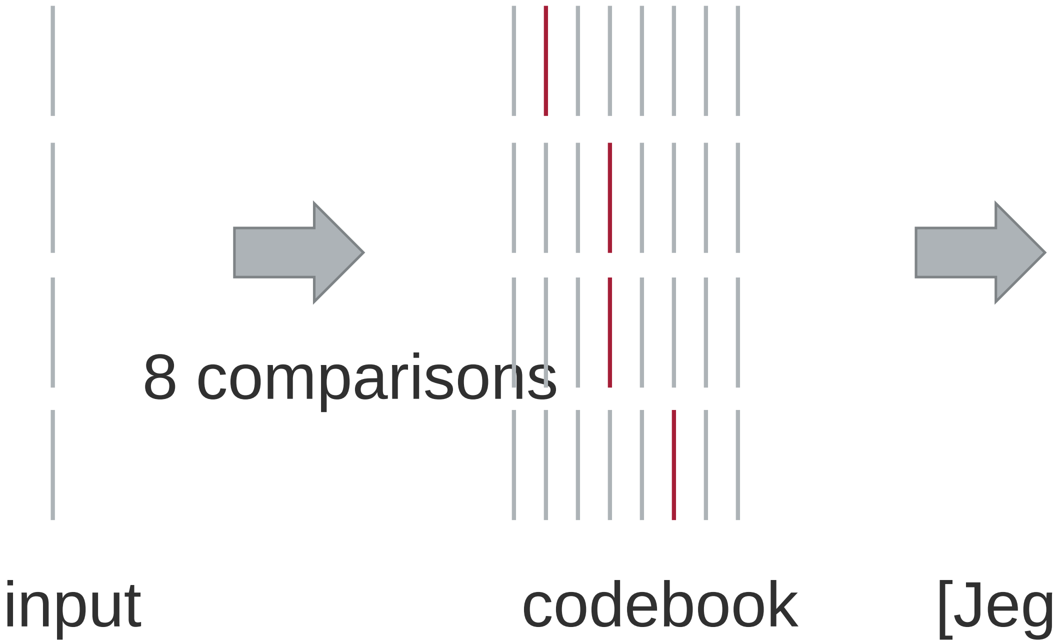
Vector Quantization

- VQ – number of bins = number of cluster = number of comparisons



Product Quantization

- VQ – number of bins = number of cluster = number of comparisons
- Segment vector
- VQ on each vector part

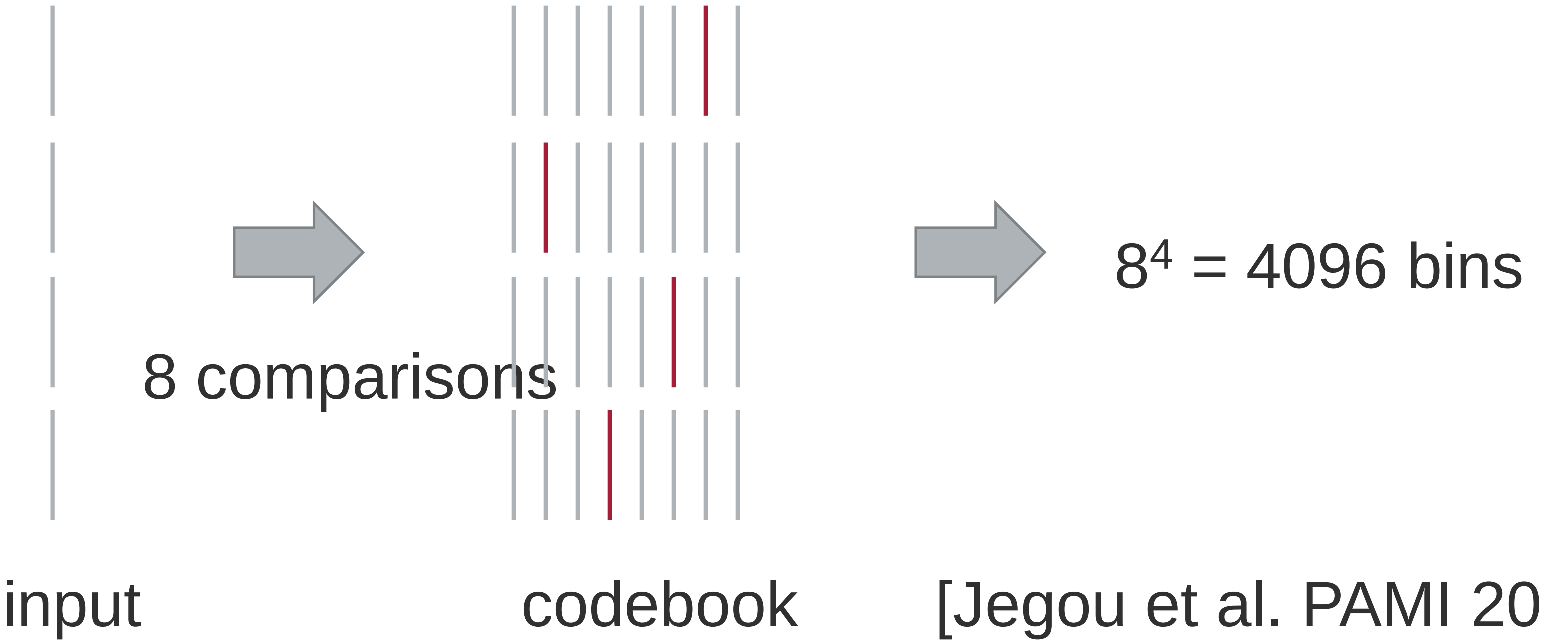


$8^4 = 4096$ bins

[Jegou et al. PAMI 2011]

Product Quantization

- VQ – number of bins = number of cluster = number of comparisons
- Segment vector
- VQ on each vector part

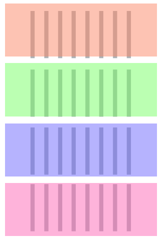


[Jegou et al. PAMI 2011]

Product Quantization Trees

- E.g. Build a two-layer tree

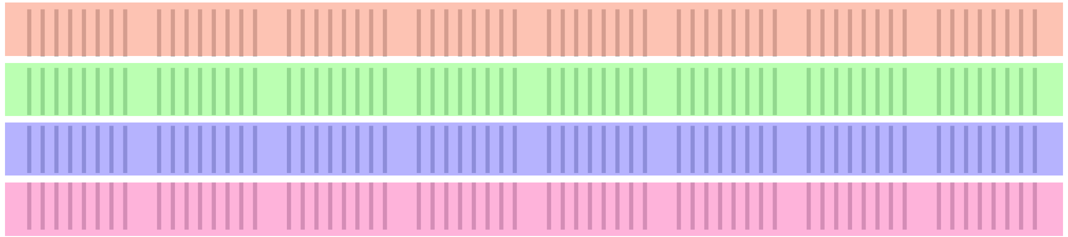
1st level



8^4 bins

- Each first-level-bin gets another codebook of 8 vectors

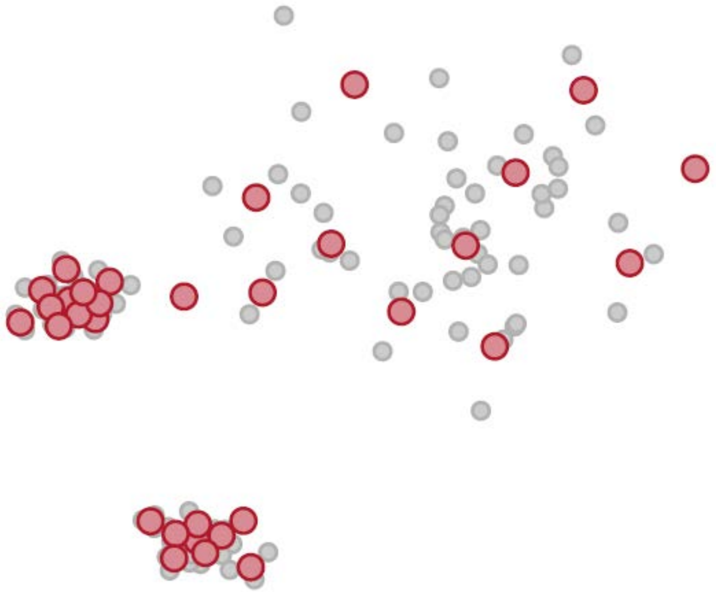
2nd level



$8^4 * 8^4 = 16,777,216$ bins

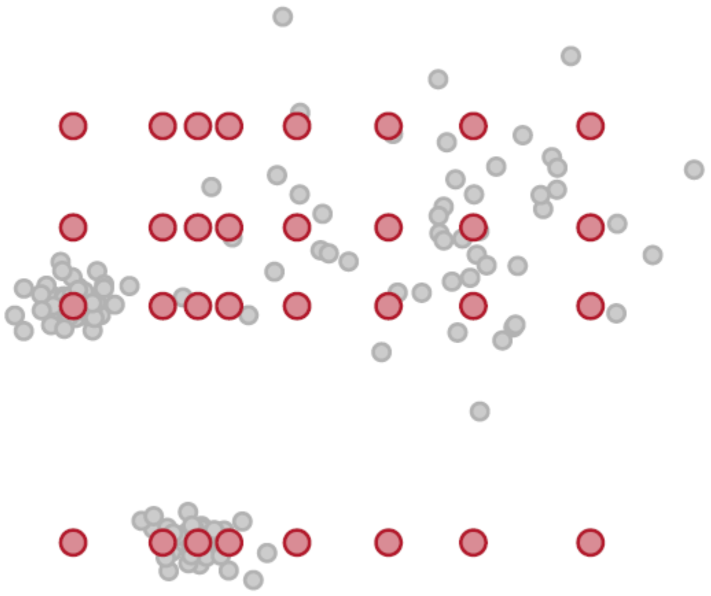
Vector Quantization

- #bins = #comparisons



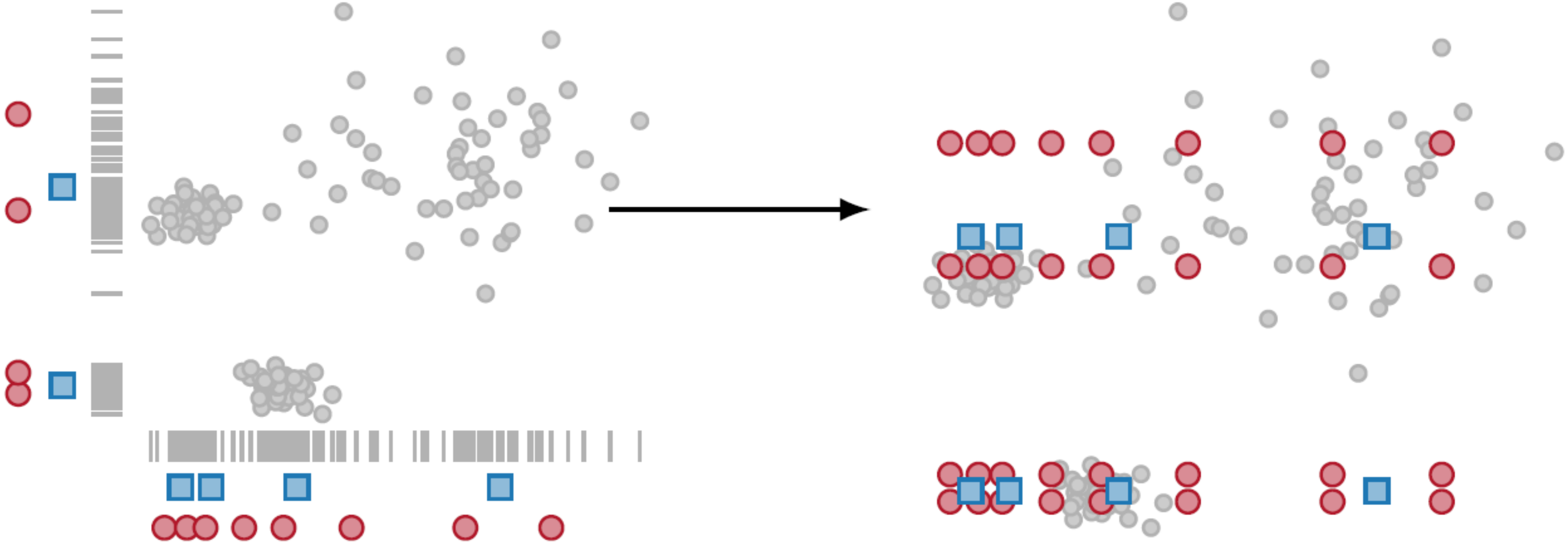
Product Quantization

- VQ for each part / group of dimensions
- Cartesian product of VQ
- #bins = #comparisons $^{\#parts}$



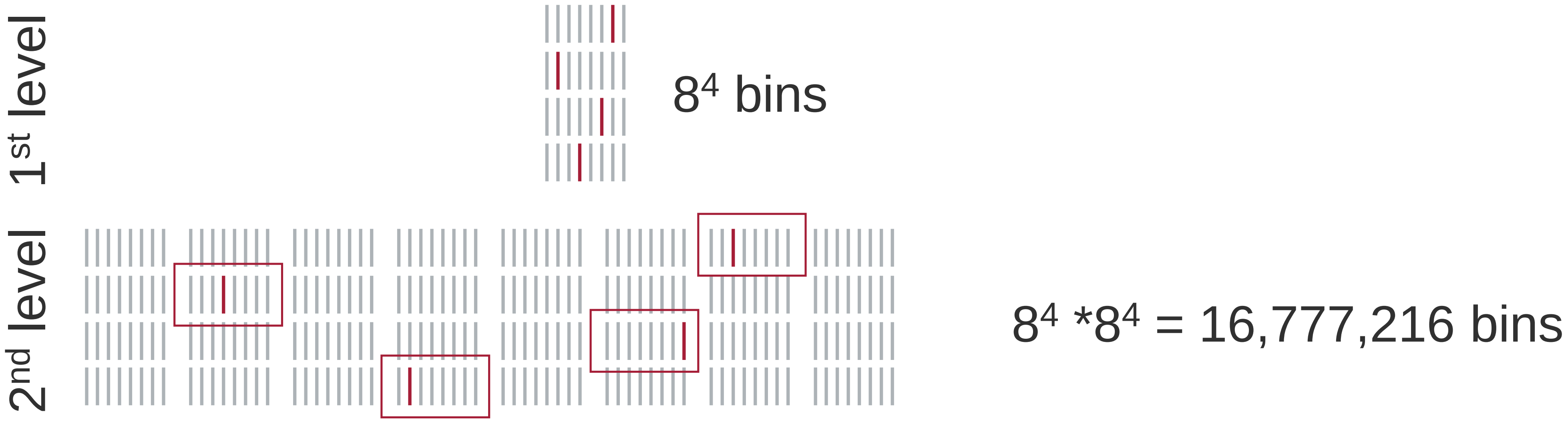
Product Quantization Tree

- #bins = $(\#\text{centroids}_{L1} * \#\text{centroids}_{L2})^{\#\text{parts}}$
- #comparisons = $\#\text{centroids}_{L1} + w * \#\text{centroids}_{L2}$



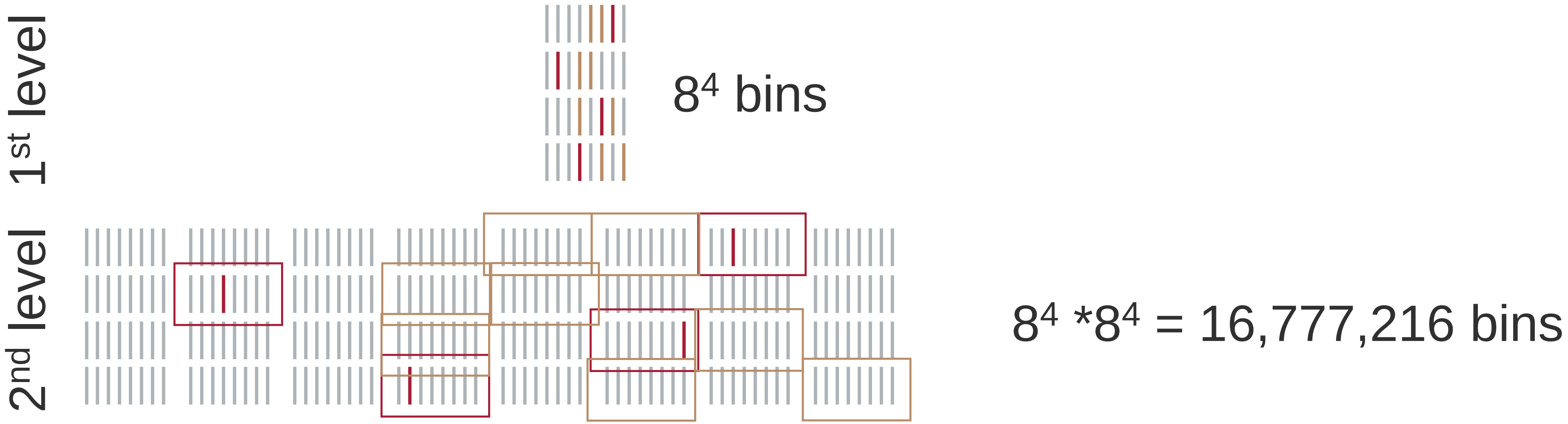
hierarchical subspace clustering Product Quantization Tree

Compact Product Quantization Trees



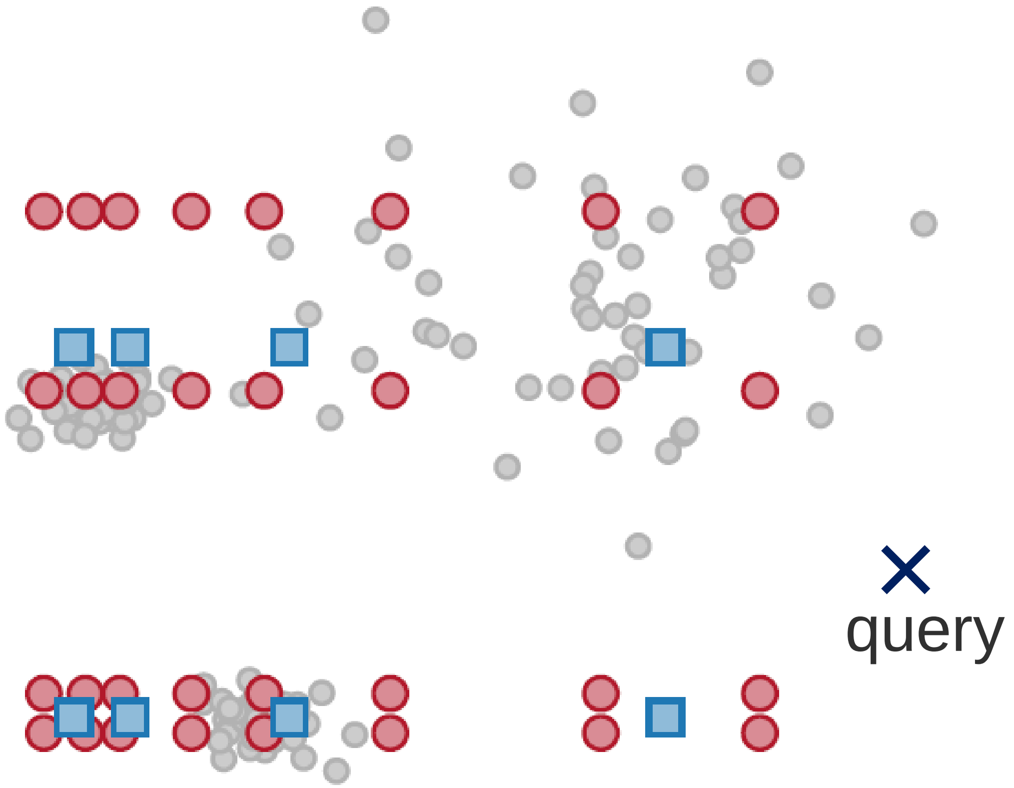
- Best centroids in 1st level identify group of centroids in 2nd level
- Basically, VQ tree on each part
  - requires only 8+8*8 vectors to represent tree
  - millions of addressable bins
  - number of compared vectors 2 * 8

Product Quantization Trees – K-Best Lookup



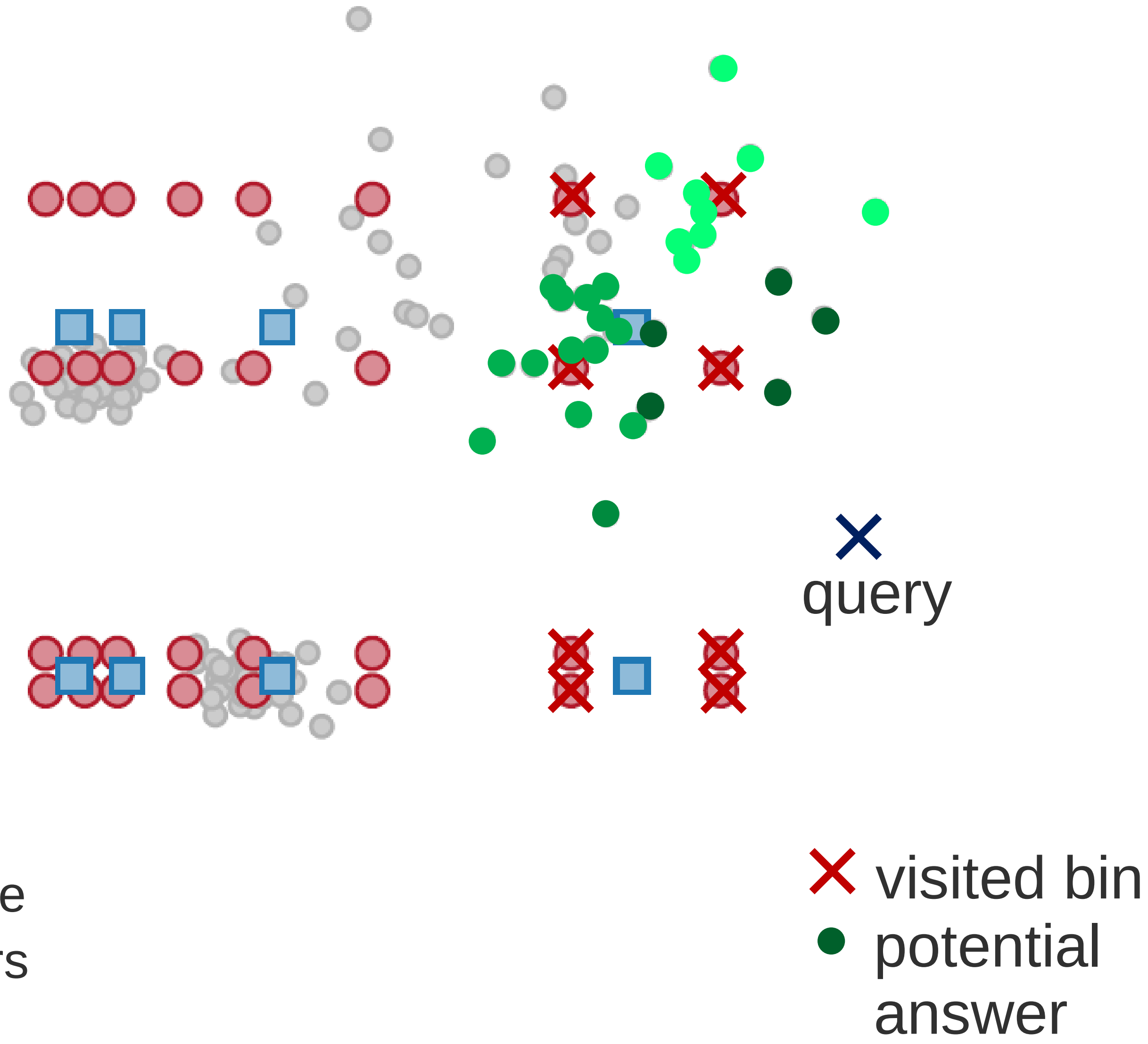
- Visit the k-best neighbors of the first bin
  - expand each subtree
  - number of compared vectors 3 * 2 * 8
  - sort by 2nd-level distances

Complete Query



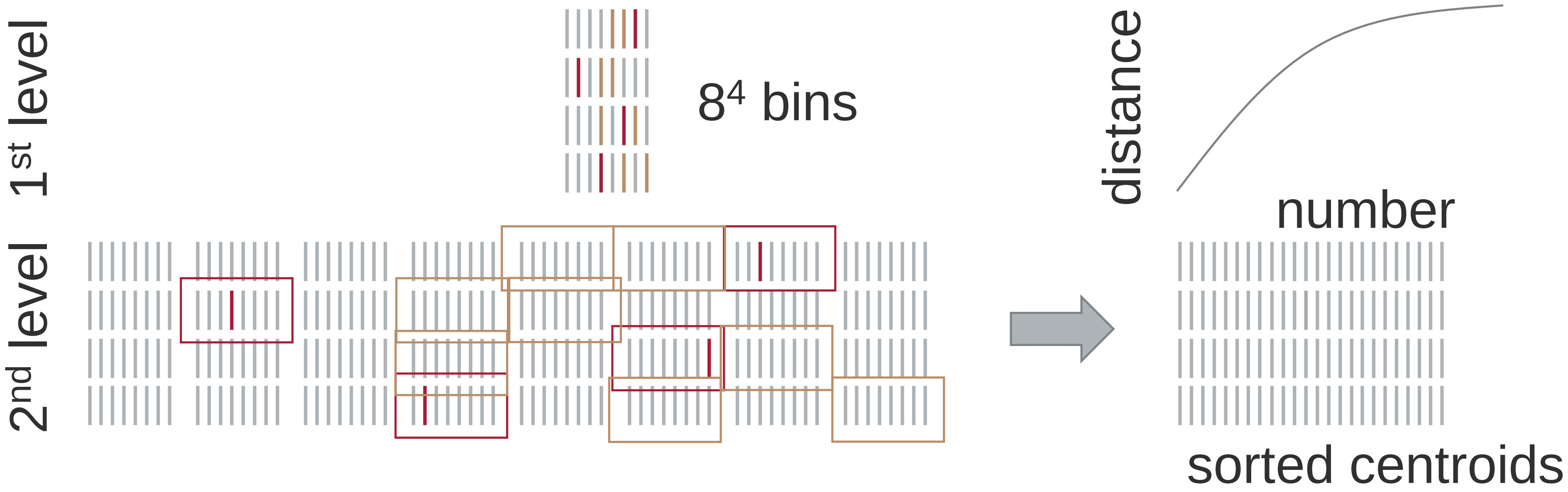
- Find closest bin
- Visit all neighbor bins, one by one
- Report and sort contained vectors

Complete Query



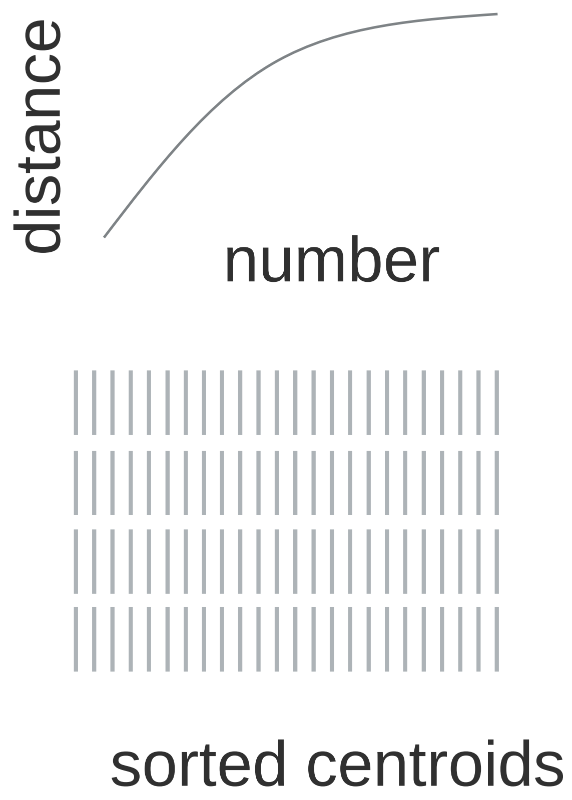
- Find closest bin
- Visit all neighbor bins, one by one
- Report and sort contained vectors

Product Quantization Trees - Lookup



- Visit the k-best neighbors of the first bin
  - expand each subtree
  - number of compared vectors 3 * 2 * 8
  - sort 2nd level distances

Enumerating Neighbor Bins - Heuristic



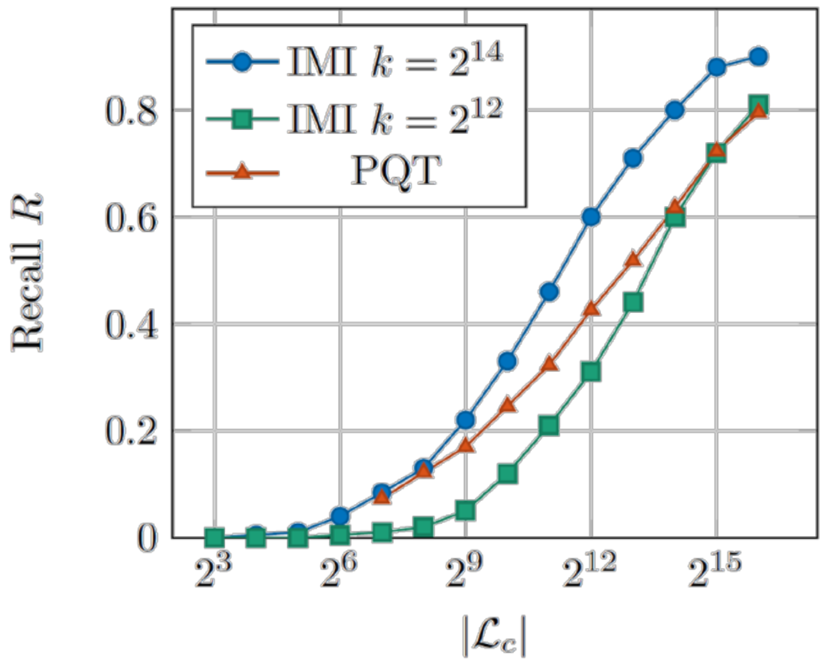
- How to visit all neighbor bins?
- Priority-queue vs. heuristic

- Generate all possible combintations using P-partitions for {0,1,….p * N}, e.g. for 2 parts
  {(0,0),
  (1,0),(0,1),
  (2,0),(1,1),(0,2)
  (3,0),(2,1),(1,2),(0,3),
  … }

- Sort all proposed bins

Results

- Precision for reported vector list length
- 1,000,000,000 SIFT vectors, $(32*16)^4$ bins
- 0.067ms / query (speedup 1000x)



reported list length

Results

- 1,000,000 SIFT vectors (128 D)

Method	Recall @ 100	Time (ms)
FLANN	0.97	5.32
LOPQ	0.97	51.1
IVFADC	0.93	11
PQT(CPU)	1.00	5.74
PQT(GPU)	0.92	**0.02**

Conclusion

- Product Quantization Trees
  - 100x to 1000x speedup
  - GPU friendly approach
  - Many empty bins

- Enabling methods for future research
  - Dynamic changes in DB possible
  - Video matching (FLANN: 1.5 days, PPQT: 20min)

‘All’-type problems

- **Nearest-neighbor search**

$$\boxed{NN(x_q) = \arg\min_r \|x_q - x_r\|}$$

All-nearest neighbor (bichromatic):

$$\boxed{\forall x_q : NN(x_q) = \arg\min_r \|x_q - x_r\|}$$

- **Kernel density estimation**

$$\boxed{\hat{f}(x_q) = \frac{1}{N} \sum_{r \neq q}^{N} K_h(\|x_q - x_r\|)}$$

‘All’ version (bichromatic):

$$\boxed{\forall x_q : \hat{f}(x_q) = \frac{1}{N} \sum_{r \neq q}^{N} K_h(\|x_q - x_r\|)}$$

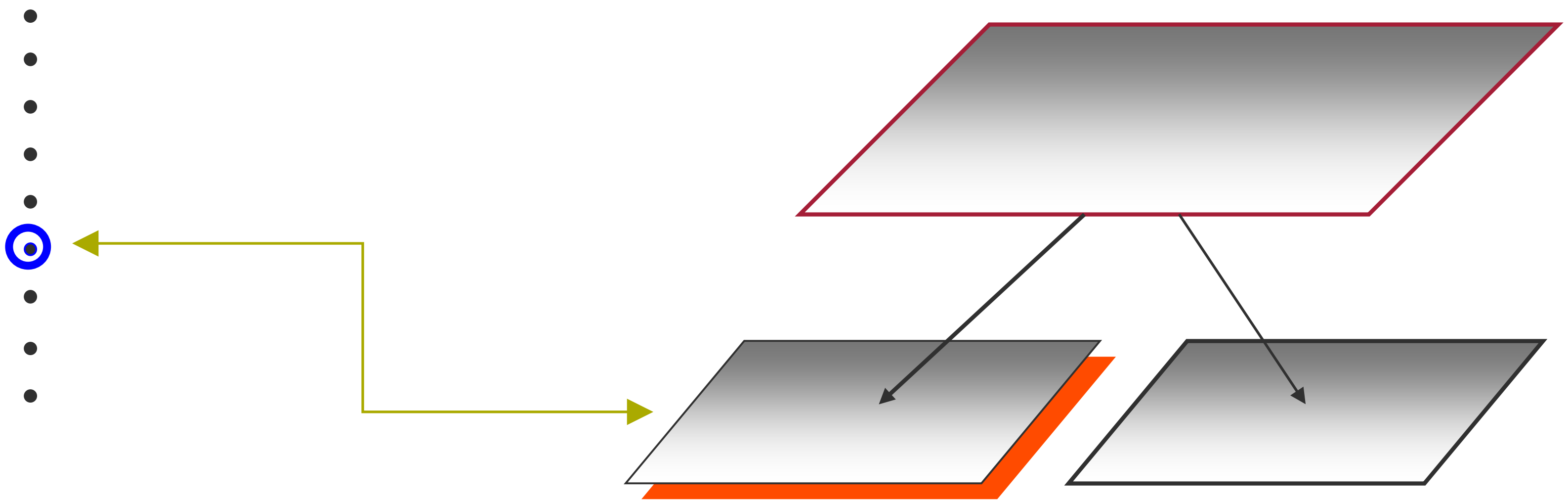
Dual-tree idea

If all the queries are available simultaneously, then it is faster to:

1. Build a tree on the queries as well
2. Effectively process the queries in chunks rather than individually
   → *work is shared between similar query points*

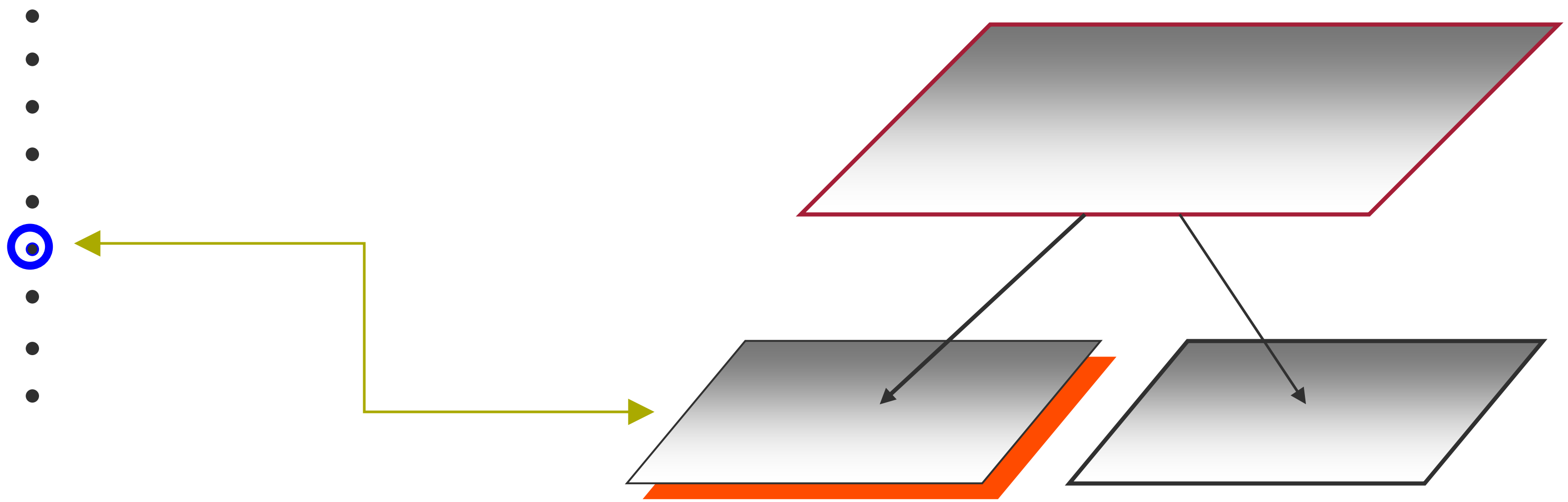
Single vs. Dual Tree

- Single Tree

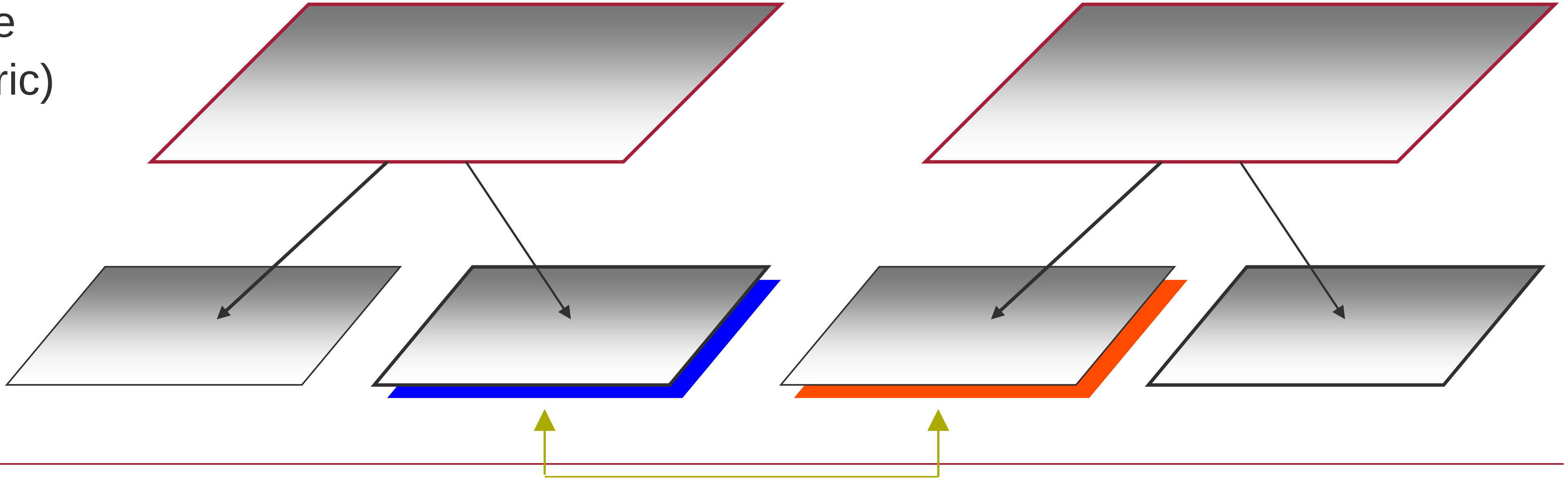


Single vs. Dual Tree

- Single Tree



- Dual Tree (symmetric)



Exclusion and Inclusion

using point-node *kd*-tree bounds.

O(D) bounds on distance minima/maxima:

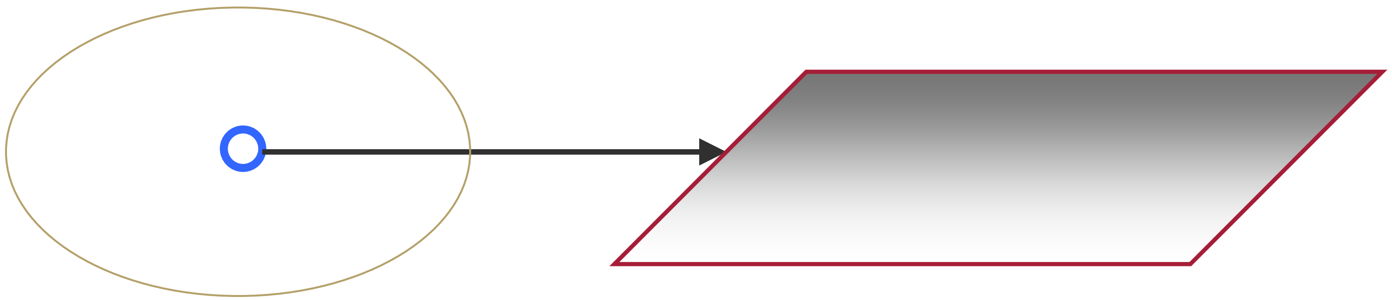
$$\min_i \|x - x_i\| \geq \sum_d^D \left[\max\left\{ (l_d - x_d)^2, 0 \right\} + \max\left\{ (x_d - u_d)^2, 0 \right\} \right]$$

$$\max_i \|x - x_i\| \leq \sum_d^D \max\left\{ (u_d - x_d)^2, (x_d - l_d)^2 \right\}$$

Exclusion and inclusion

using point-node *kd*-tree bounds.

O(D) bounds on distance minima/maxima:



$$\min_i \|x - x_i\| \geq \sum_d^D \left[\max\left\{(l_d - x_d)^2, 0\right\} + \max\left\{(x_d - u_d)^2, 0\right\}\right]$$

$$\max_i \|x - x_i\| \leq \sum_d^D \max\left\{(u_d - x_d)^2, (x_d - l_d)^2\right\}$$

Exclusion and Inclusion

using <u>node-node</u> *kd*-tree bounds.

O(D) bounds on distance minima/maxima:

(Analogous to point-node bounds.)

Requires nodewise bounds

Exclusion and Inclusion

using node-node *kd*-tree bounds.

O(D) bounds on distance minima/maxima:

(Analogous to point-node bounds.)

Requires nodewise bounds

Dual-tree: simple recursive algorithm (k=1)

AllNN$(Q,R,d_{lo},\underline{x}_{sofar},\underline{d}_{sofar})$

{

if $d_{lo} > Q.d_{sofar}$, return.

if leaf(Q) & leaf(R),

$[\underline{x}_{sofar},\underline{d}_{sofar}]=$**AllNNBase**$(Q,R,\underline{d}_{sofar})$. $Q.d_{sofar}=\max_Q \underline{d}_{sofar}$.

else if !leaf(Q) & leaf(R), …

else if leaf(Q) & !leaf(R), …

else if !leaf(Q) & !leaf(R),

[R1,d1,R2,d2]=orderByDist(Q.l,R.l,R.r).

AllNN$(Q.l,R1,d1,\underline{x}_{sofar},\underline{d}_{sofar})$.

AllNN$(Q.l,R2,d2,\underline{x}_{sofar},\underline{d}_{sofar})$.

[R1,d1,R2,d2]=orderByDist(Q.r,R.l,R.r).

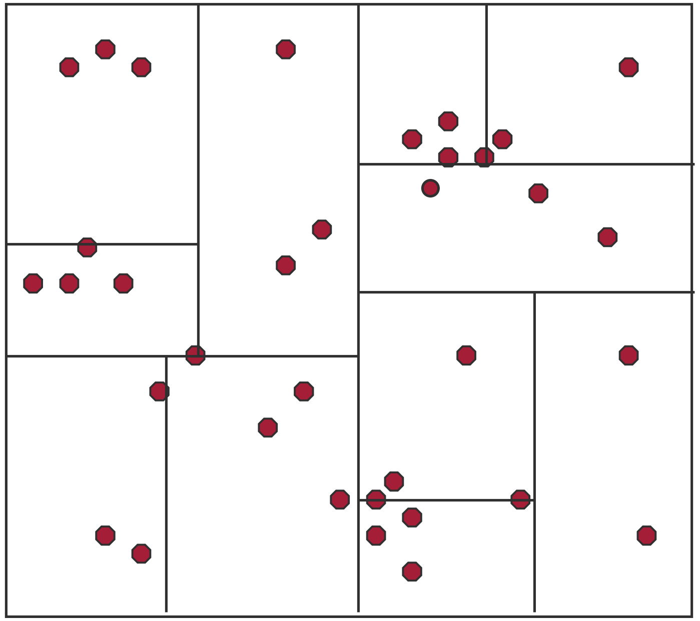
AllNN$(Q.r,R1,d1,\underline{x}_{sofar},\underline{d}_{sofar})$.

AllNN$(Q.r,R2,d2,\underline{x}_{sofar},\underline{d}_{sofar})$.

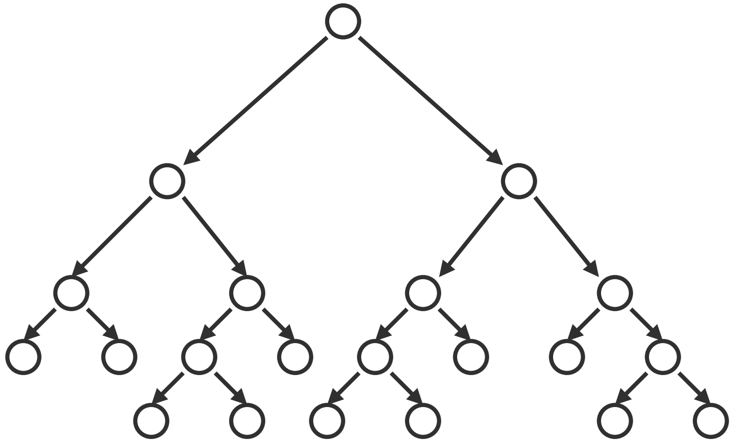
$Q.d_{sofar} = \max(Q.l.d_{sofar},Q.r.d_{sofar})$.

}

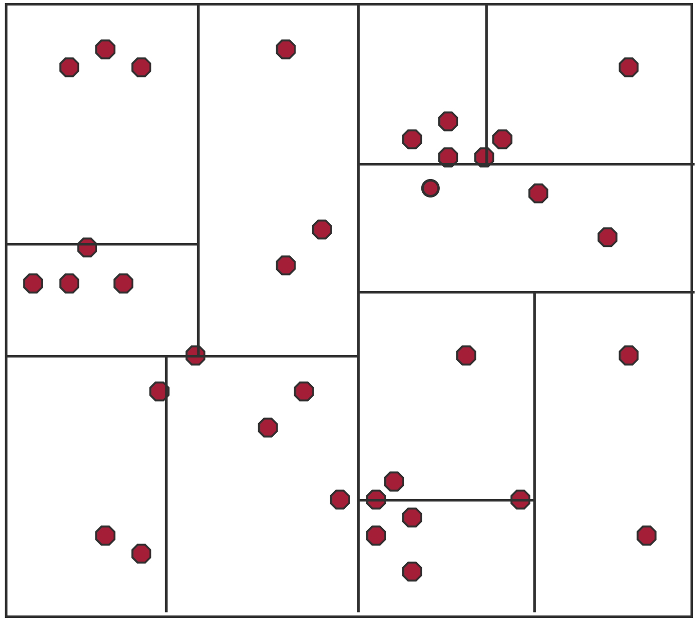
Dual-Tree



tree for
data points

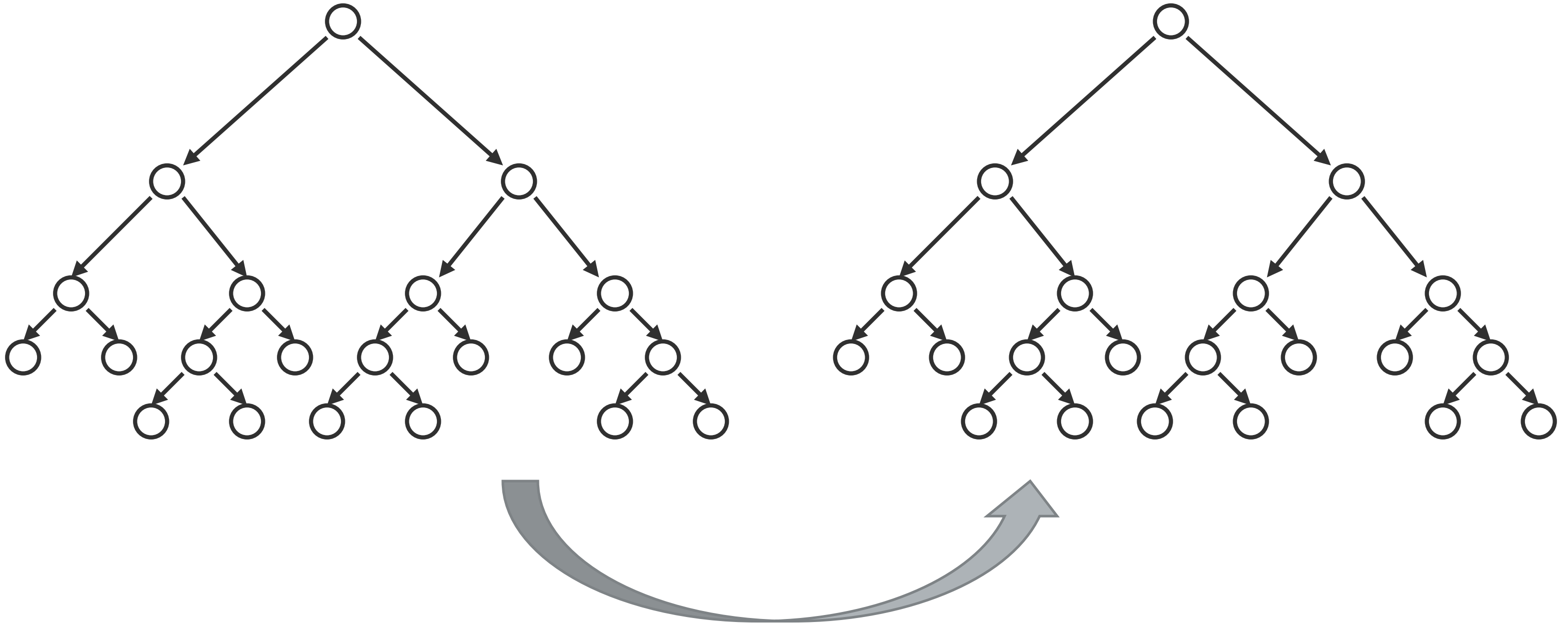


Dual-Tree



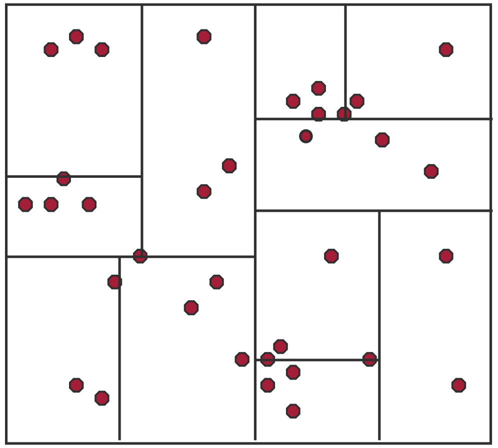
tree for
data points

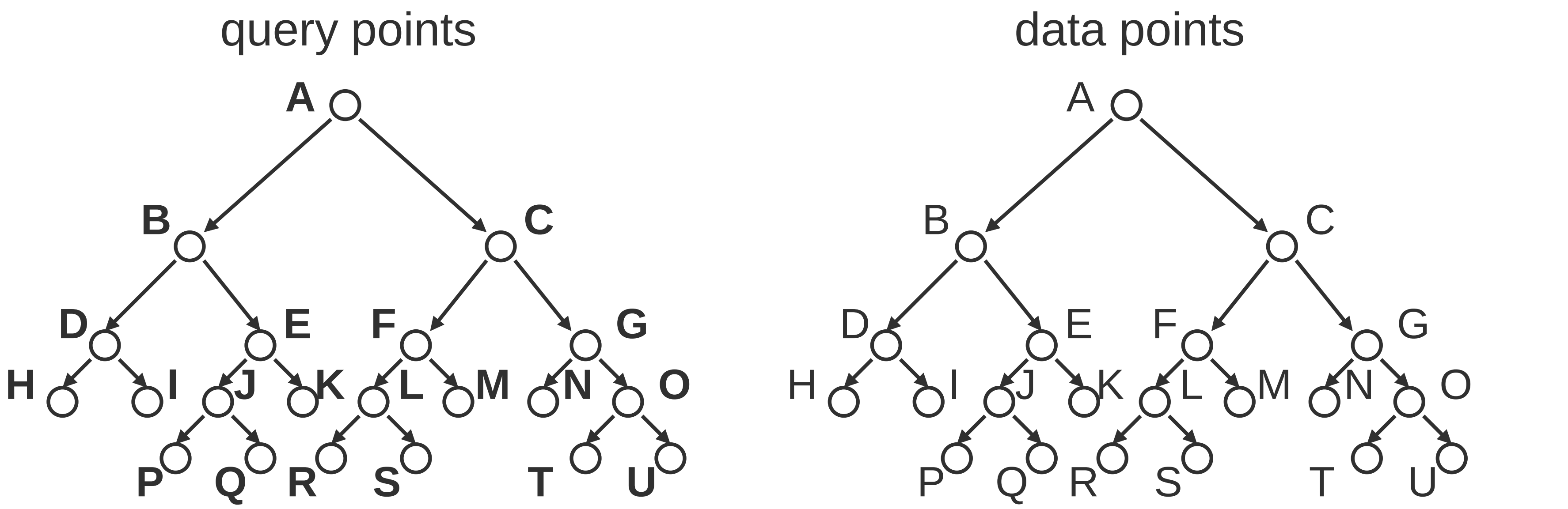
tree for
query points



second instance of the same tree

Dual-Tree Traversal





- Start with (**A**,A)   - test possible NN
- Level 1:   (**B**,B), (**B**,C) , (**C**,B) , (**C**,C)
- Level 2:   (**D**,D), (**D**,E), ~~(**D**,F)~~, ~~(**D**,G)~~, (**E**,D), (**E**,E), (**E**,F), ~~(**E**,G)~~, ~~(**F**,D)~~, (**F**,E), (**F**,F), (**F**,G), ~~(**G**,D)~~, ~~(**G**,E)~~, (**G**,F), (**G**,G)
- Level 3:   (**H**,H), (**H**,I), (**I**,I), ~~(**H**,J)~~, ~~(**H**,K)~~, (**I**,J), ~~(**I**,K)~~, …

Higher-order Dvide-and-conquer

Generalizes divide-and-conquer of a single set to divide-and-conquer of multiple sets.

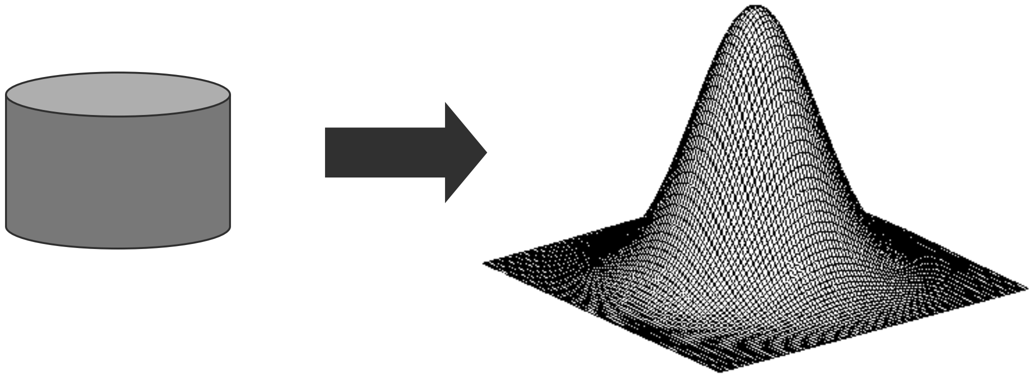
Break each set into pieces.

Solving the sub-parts of the problem and combining these sub-solutions appropriately might be easier than doing this over only one set.

Ideas

- Data structures and how to use them
- Multipole methods
- Problem/solution abstractions

Kernel density estimation

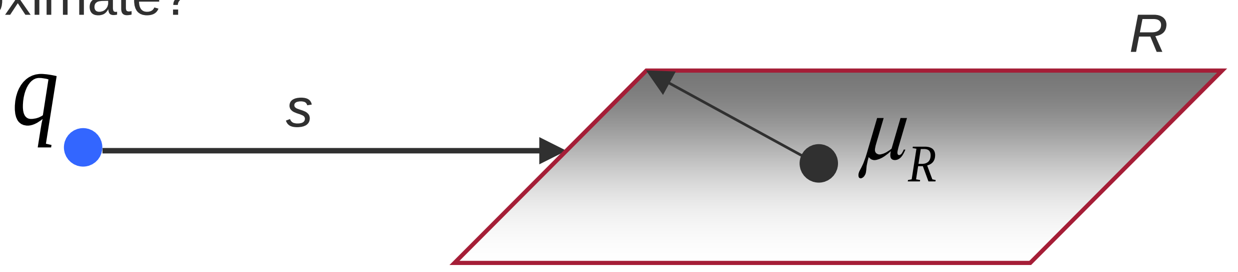


$$\forall x_q, \quad \hat{f}(x_q) = \frac{1}{N} \sum_{r \neq q}^{N} K_h(\|x_q - x_r\|)$$

Multipole Method

- [Barnes and Hut, Science, 1987]

- Point-to-Cell Kernel Estimation
- How to use a tree…
  1. **How** to approximate?
  2. **When** to approximate?



$$\sum_i K(q, x_i) \approx N_R K(q, \mu_R)$$

$$\boxed{\text{if} \quad s > \frac{r}{\theta}}$$

Multipole Method

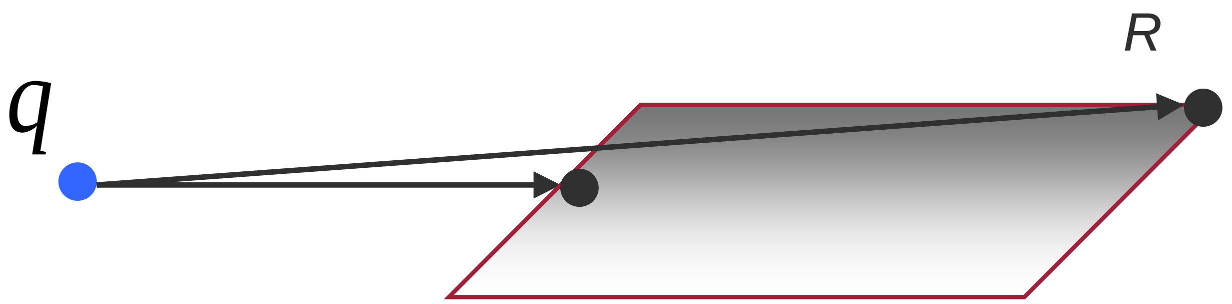
- [Barnes and Hut, Science, 1987]

- Point-to-Cell Kernel Estimation
- How to use a tree…

  3. How to know **potential error**?

Let's maintain bounds on the true kernel sum

$$\Phi(q) \equiv \sum_i K(q, x_i)$$



At the beginning:

$$\Phi^{lo}(q) \leftarrow NK^{lo} \qquad \Phi^{lo}(q) \leftarrow \Phi^{lo}(q) + N_R K(q, \delta_{qR}^{lo}) - N_R K^{lo}$$

$$\Phi^{hi}(q) \leftarrow NK^{hi} \qquad \Phi^{hi}(q) \leftarrow \Phi^{hi}(q) + N_R K(q, \delta_{qR}^{hi}) - N_R K^{hi}$$

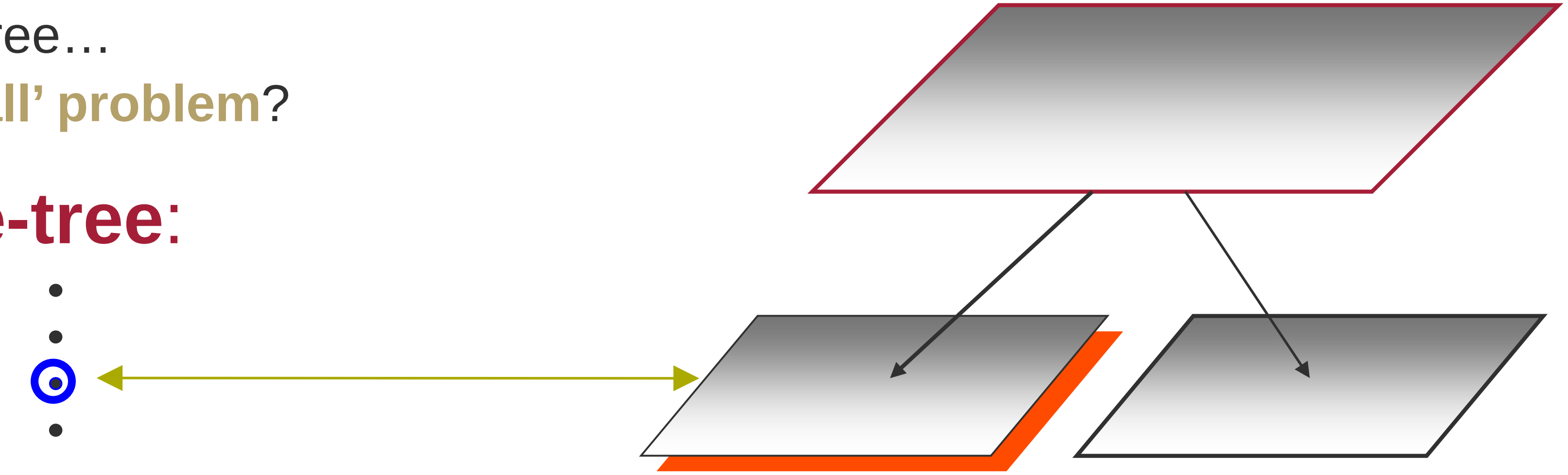
Multipole Method

$$\forall x_q, \quad \hat{f}(x_q) = \frac{1}{N} \sum_{r \neq q}^{N} K_h(\|x_q - x_r\|)$$

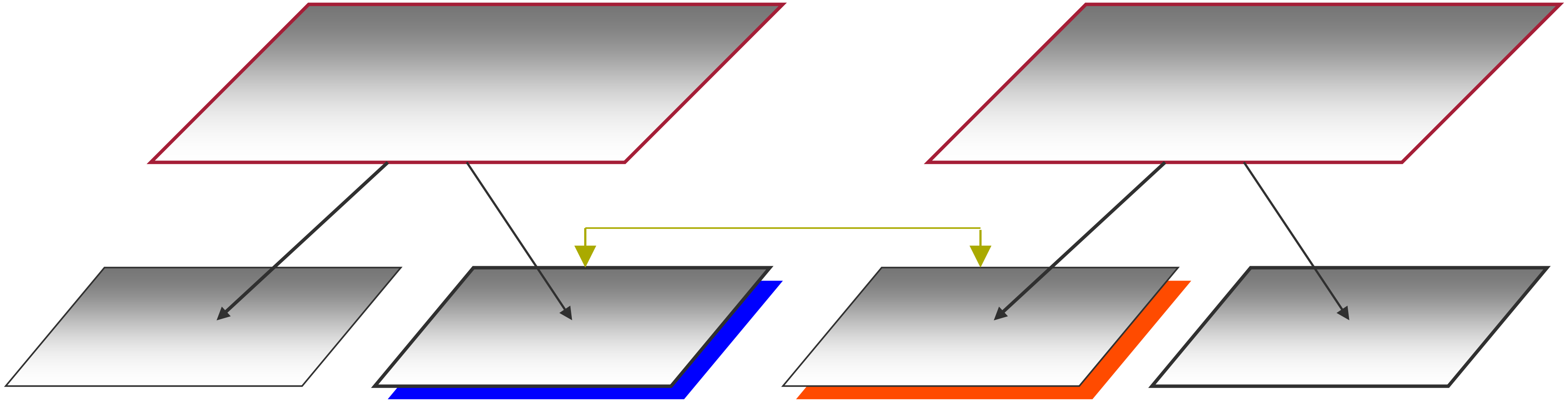
- How to use a tree…

  4. How to do **'all' problem**?

Single-tree:

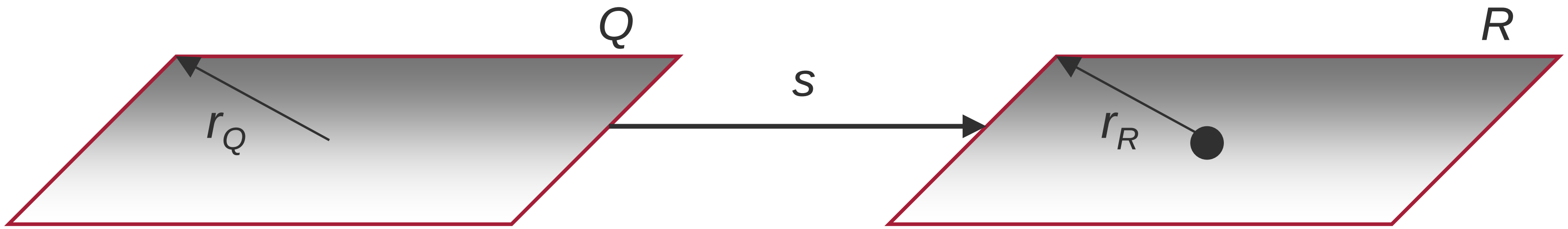


Dual-tree (symmetric): [Gray & Moore 2000]



Multipole Method

- How to use a tree…
  4. How to do ‘all’ problem?
- Generalizes Barnes-Hut to dual-tree



$$\forall q \in Q, \sum_i K(q, x_i) \approx N_R K(q, \mu_R)$$

$$\boxed{\text{if} \quad s > \frac{\max(r_Q, r_R)}{\theta}}$$

BUT:

We have a tweak parameter: θ

Case 1 – alg. gives no error bounds
Case 2 – alg. gives error bounds, but must be rerun
Case 3 – alg. automatically achieves error tolerance

So far we have case 2;
let’s try for case 3

Let’s try to make an automatic stopping rule

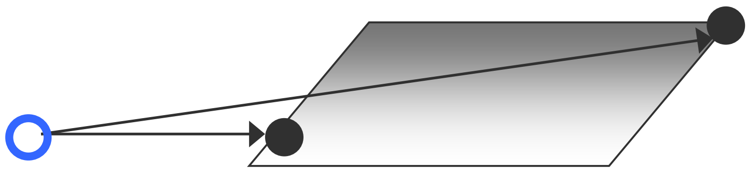
Finite-difference function approximation

Taylor expansion:

$$f(x) \approx f(a) + f'(a)(x-a)$$

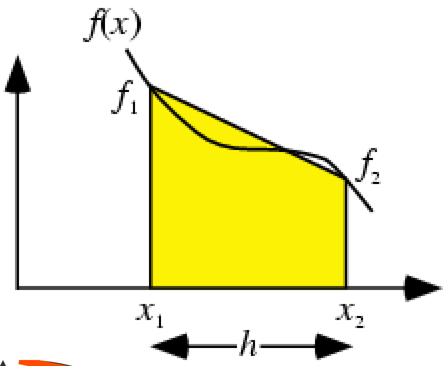
Gregory-Newton finite form:

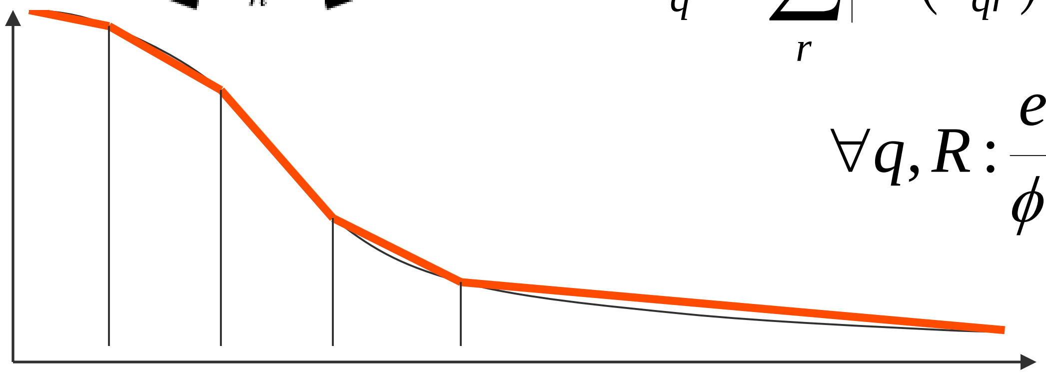
$$f(x) \approx f(x_i) + \frac{1}{2}\left(\frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}\right)(x - x_i)$$



$$K(\delta) \approx K(\delta^{lo}) + \frac{1}{2}\left(\frac{K(\delta^{hi}) - K(\delta^{lo})}{\delta^{hi} - \delta^{lo}}\right)(\delta - \delta^{lo})$$

Finite-difference function approximation





assumes monotonic decreasing kernel

$$\bar{K} = \tfrac{1}{2}\left[K(\delta_{QR}^{lo}) + K(\delta_{QR}^{hi})\right]$$

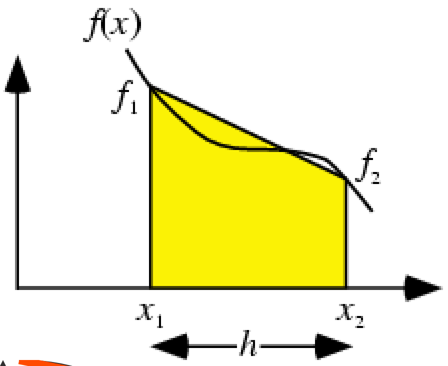
$$err_q = \sum_{r}^{N_R} \left|K(\delta_{qr}) - \bar{K}\right| \le \frac{N_R}{2}\left[K(\delta_{QR}^{lo}) - K(\delta_{QR}^{hi})\right]$$

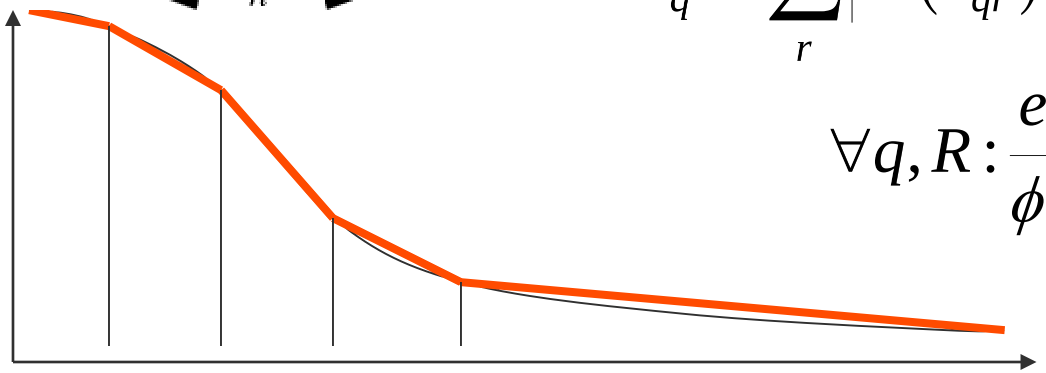
$$\forall q, R : \frac{err_{qR}}{\phi(x_q)} \le \frac{N_R}{N}\varepsilon \Rightarrow \forall q : \frac{err_q}{\phi(x_q)} \le \varepsilon$$

approximate {Q,R} if

$$K(\delta_{lo}) - K(\delta_{hi}) \le \tfrac{2\varepsilon}{N}\Phi_{lo}(Q)$$

Finite-difference function approximation





assumes monotonic decreasing kernel

$$\bar{K} = \tfrac{1}{2}\left[K(\delta_{QR}^{lo}) + K(\delta_{QR}^{hi})\right]$$

$$err_q = \sum_r^{N_R} \left| K(\delta_{qr}) - \bar{K} \right| \le \frac{N_R}{2}\left[K(\delta_{QR}^{lo}) - K(\delta_{QR}^{hi})\right]$$

$$\forall q, R : \frac{err_{qR}}{\phi(x_q)} \le \frac{N_R}{N}\varepsilon \Rightarrow \forall q : \frac{err_q}{\phi(x_q)} \le \varepsilon$$

lower bound

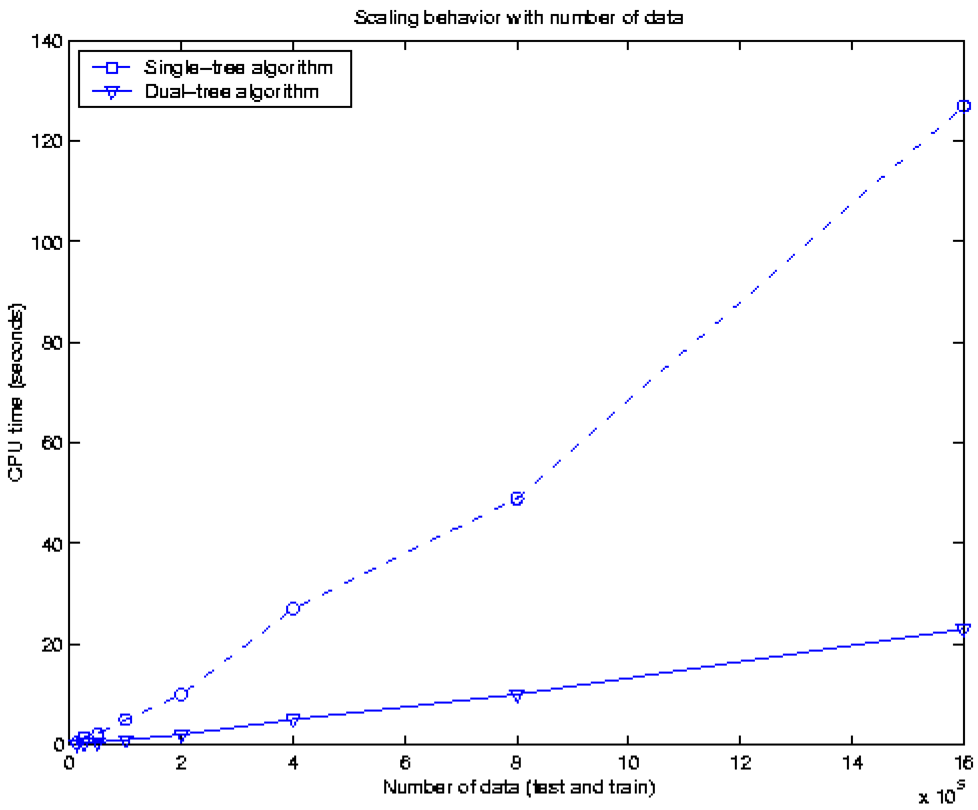
approximate {Q,R} if

$$K(\delta_{lo}) - K(\delta_{hi}) \le \tfrac{2\varepsilon}{N}\Phi_{lo}(Q)$$

Speedup Results

N	naïve	dual-tree
12.5K	7	.12
25K	31	.31
50K	123	.46
100K	494	1.0
200K	1976*	2
400K	7904*	5
800K	31616*	10
1.6M	35 hrs	23

5500x



One order-of-magnitude speedup over single-tree at ~2M points

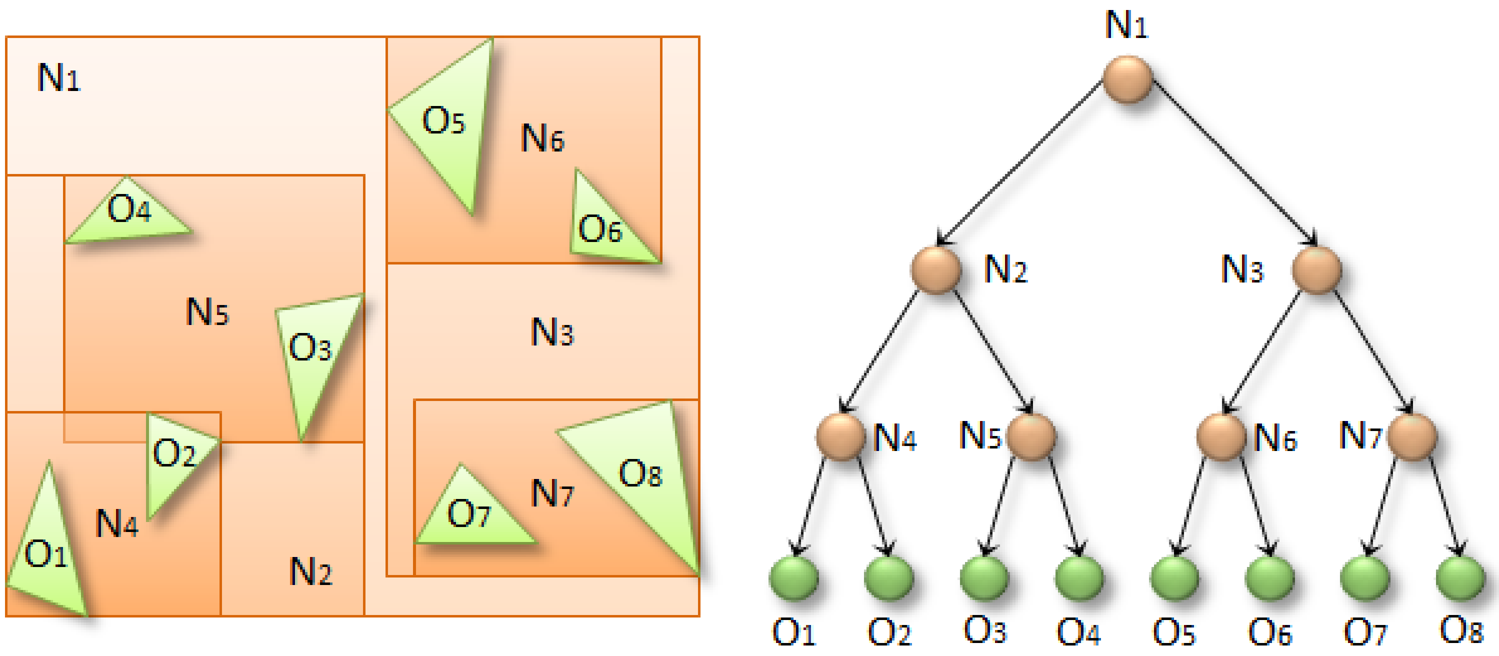
Tree-Traversal in Cuda

Tero Karras:

https://devblogs.nvidia.com/parallelforall/thinking-parallel-part-ii-tree-traversal-gpu/

Bounding Volume Hierarchy

- Goal: fast intersection test for all geometry
- Spatial data structure needed



Naive, independent Traversal

```
void traverseRecursive( CollisionList& list, const BVH&bvh,
  const AABB& queryAABB, int queryObjectIdx, NodePtr node) {
    // Bounding box overlaps the query => process node.
    if (checkOverlap(bvh.getAABB(node), queryAABB)) {
        // Leaf node => report collision.
        if (bvh.isLeaf(node))
            list.add(queryObjectIdx, bvh.getObjectIdx(node));
        // Internal node => recurse to children.
        else {
            NodePtr childL = bvh.getLeftChild(node);
            NodePtr childR = bvh.getRightChild(node);
            traverseRecursive(bvh, list, queryAABB,
                              queryObjectIdx, childL);
            traverseRecursive(bvh, list, queryAABB,
                              queryObjectIdx, childR);
        }
    }
}
```

Naive Traversal (Cuda)

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

```
__device__ void traverseRecursive( CollisionList& list, const BVH&bvh,
  const AABB& queryAABB, int queryObjectIdx, NodePtr node)
{
    // same as before...
}

__global__ void findPotentialCollisions( CollisionList list,
                                          BVH           bvh,
                                          AABB*         objectAABBs,
                                          int           numObjects)
{
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < numObjects)
        traverseRecursive(bvh, list, objectAABBs[idx],
                          idx, bvh.getRoot());
}
```

Naive Traversal – Discussion

- Approx. 3.8ms for 12486 objects
- Recursion introduces divergence
  - left or right branch

- Solution:
  - iterative approach
  - each thread maintains its own recursion stack

Traversal – Iterative

```
__device__ void traverseIterative ( CollisionList& list, const BVH&bvh,
  const AABB& queryAABB, int queryObjectIdx, NodePtr node) {
        // Allocate traversal stack from thread-local memory,
    // and push NULL to indicate that there are no postponed nodes.
    NodePtr stack[64];  NodePtr* stackPtr = stack;
    *stackPtr++ = NULL; // push
    // Traverse nodes starting from the root.
    NodePtr node = bvh.getRoot();
    do
    {
        // Check each child node for overlap.
        NodePtr childL = bvh.getLeftChild(node);
        NodePtr childR = bvh.getRightChild(node);
        bool overlapL = ( checkOverlap(queryAABB,
                                       bvh.getAABB(childL)) );
        bool overlapR = ( checkOverlap(queryAABB,
                                       bvh.getAABB(childR)) );
        ...
```

Traversal – Iterative

```
            // Query overlaps a leaf node => report collision.
            if (overlapL && bvh.isLeaf(childL))
                list.add(queryObjectIdx, bvh.getObjectIdx(childL));
            if (overlapR && bvh.isLeaf(childR))
                list.add(queryObjectIdx, bvh.getObjectIdx(childR));
            // Query overlaps an internal node => traverse.
            bool traverseL = (overlapL && !bvh.isLeaf(childL));
            bool traverseR = (overlapR && !bvh.isLeaf(childR));

            if (!traverseL && !traverseR)
                node = *--stackPtr; // pop
            else {
                node = (traverseL) ? childL : childR;
                if (traverseL && traverseR)
                    *stackPtr++ = childR; // push
            }
        }
        while (node != NULL);
    }
```

Traversal – Iterative

- From 3.9ms to 0.91 ms

- All threads are executing the same loop over and over
- Threads are in sync even though they are traversing completely different parts of the tree

- Still data divergence

- Solution:
  - Exploit BVH to process groups of nearby objects

Traversal – BVH-List

```
__global__ void findPotentialCollisions( CollisionList list,
                                          BVH            bvh)
{
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < bvh.getNumLeaves())
    {
        NodePtr leaf = bvh.getLeaf(idx);
        traverseIterative(list, bvh,
                          bvh.getAABB(leaf),
                          bvh.getObjectIdx(leaf));
    }
}
```

- now 0.43 ms

Traversal – BVH-List

- Report each overlap only once → 0.25ms

```
__device__ void traverseIterative ( CollisionList& list, const BVH&bvh,
  const AABB& queryAABB, int queryObjectIdx, NodePtr node) {
    ...

    // Ignore overlap if the subtree is fully on the
    // left-hand side of the query.

    if (bvh.getRightmostLeafInLeftSubtree(node) <= queryLeaf)
        overlapL = false;

    if (bvh.getRightmostLeafInRightSubtree(node) <= queryLeaf)
        overlapR = false;

    ...
}
```

Dual Tree Traversal

- Query for inner nodes instead of leaves
  - should save quite some work
- Need to keep GPU busy from the beginning
- Idea:
  - Don‘t start with root but a few layers down,
    e.g. 256x256 potential tests
  - then recurse / iterate

- Problem: Divergence!!!
  - Execution and data divergence
  - Drastically different execution times
    one thread stops as soon as there is no overlap, the other continues
  - Clever book keeping and load balancing have been tried

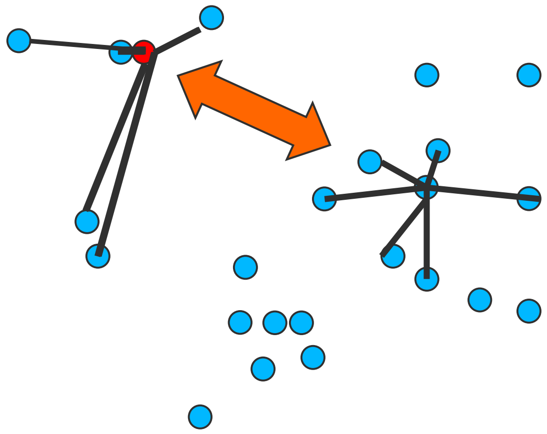
BVH-List vs. Dual Tree Traversal

- You get close to the BVH-list performance but not better

- BVH-list does more (unnecessary work)
- But, it
  - is simpler to code
  - shows less divergence
  - is more flexible to optimize

N-Body Problems

- Ubiquitous N-Body Problems
  - Astrophysics
  - Molecular Dynamics
  - Particle discretizations for PDEs
  - Data Mining
  - Irregular Sampling in Graphics
  - etc.
- Simple Observations
  - Points have no intrinsic topology
  - Metric/Kernel relations matter K(x,y)
  - N^2 interactions for all to all
- Typical Problems
  - Nearest neighbors
  - Weighted interpolation
  - Partition of unity
  - Kernel density, Multipole



Today

- N-Body Problem Introduction
- Nearest Neighbor Search in High Dimensions
- Excerpts from a Tutorial
  - Fast N-body Algorithms for Massive Datasets
  - by Alexander Gray
  - presented at the 2008 SIAM Conference on Data Mining