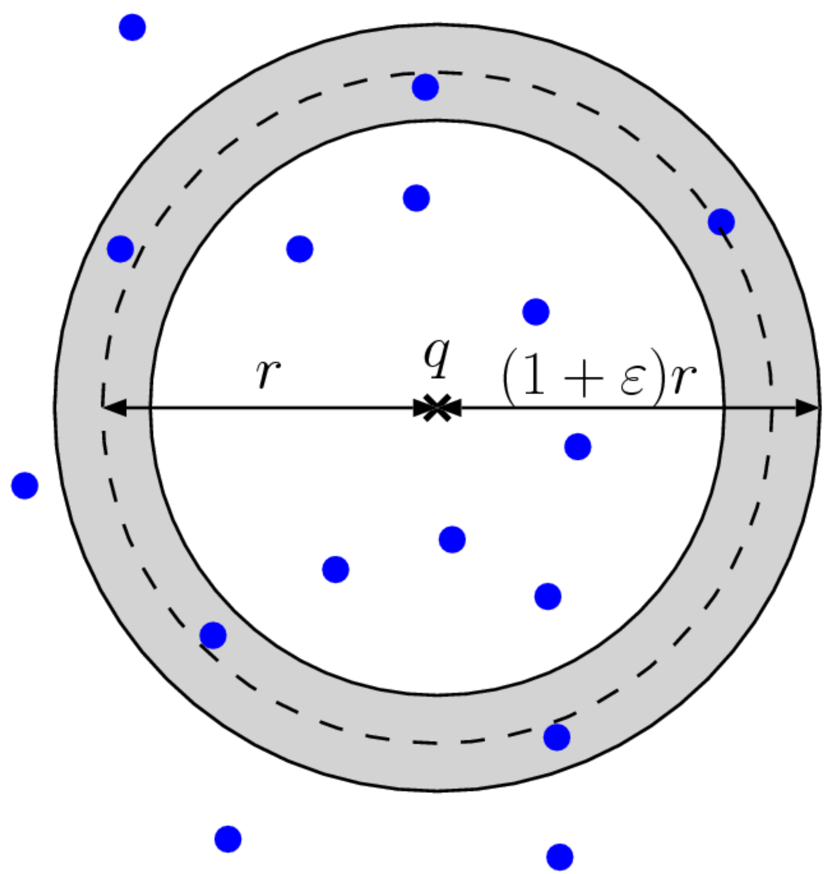
A Unified Approach to Approximate Proximity Searching



- Sunil Arya

  Hong Kong University of Science and Technology

- Guilherme D. da Fonseca

  UNIRIO

- David M. Mount

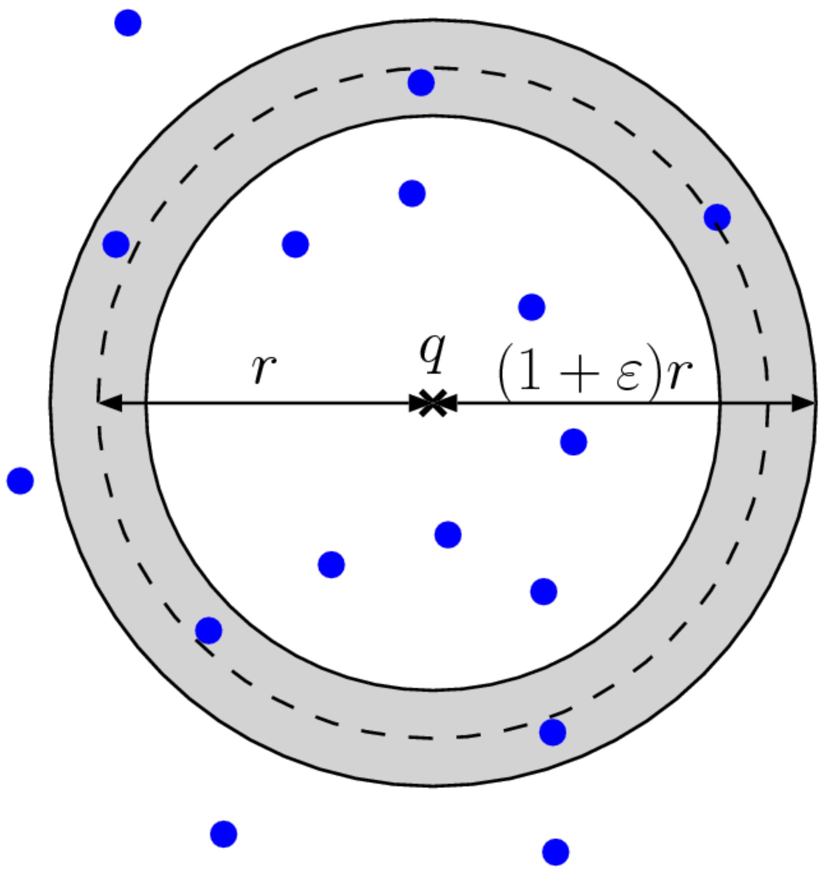
  University of Maryland

ESA, Liverpool, UK

09/2010

(Approximate) Proximity Problems

- Preprocess a weighted set of *n* points such that given a query point *q* and often a radius *r* we can determine:



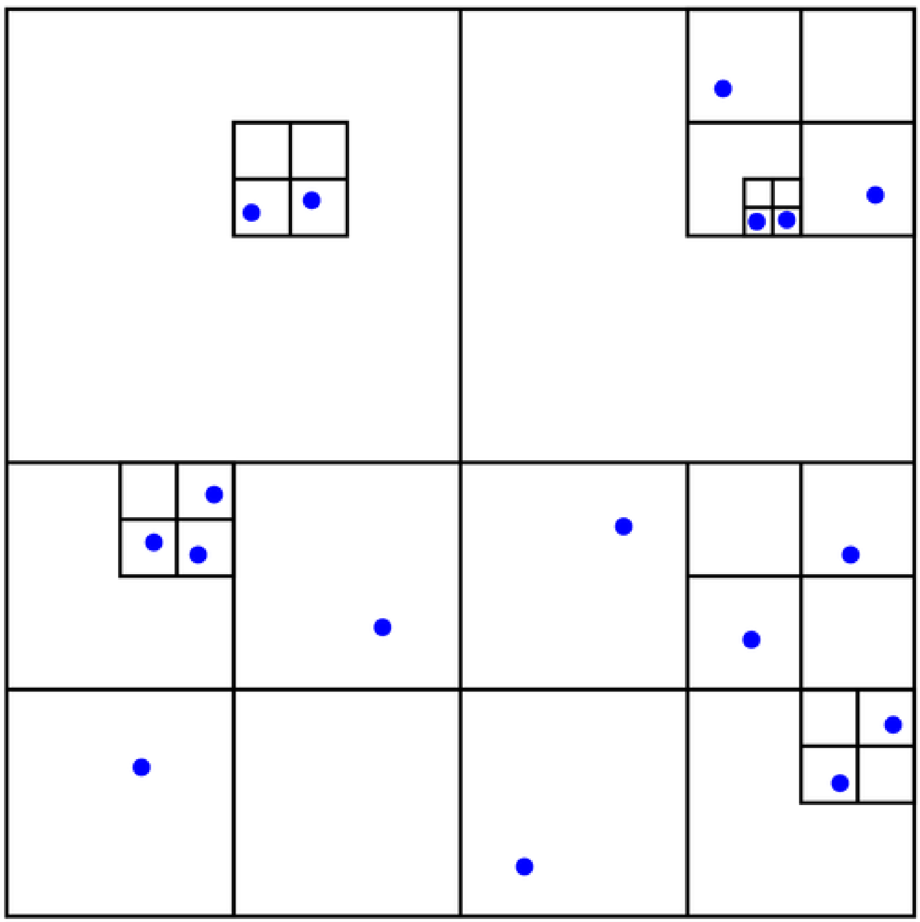
- *Spherical range queries*: the sum of the weights of the points within distance *r* from *q*.
    - Special case: the sum is *idempotent* ($x + x = x$).
- *Spherical emptiness queries*: if there is a point within radius *r* from *q*.
- *Nearest neighbor queries*: the closest point to *q*.

Motivation

- Numerous applications.
- Exact solutions are inefficient for dimension $d > 2$.
- The most efficient previous solutions are rather complicated.
- Solution to different problems used different tools.
- It is hard to see how the properties of each problem are exploited.

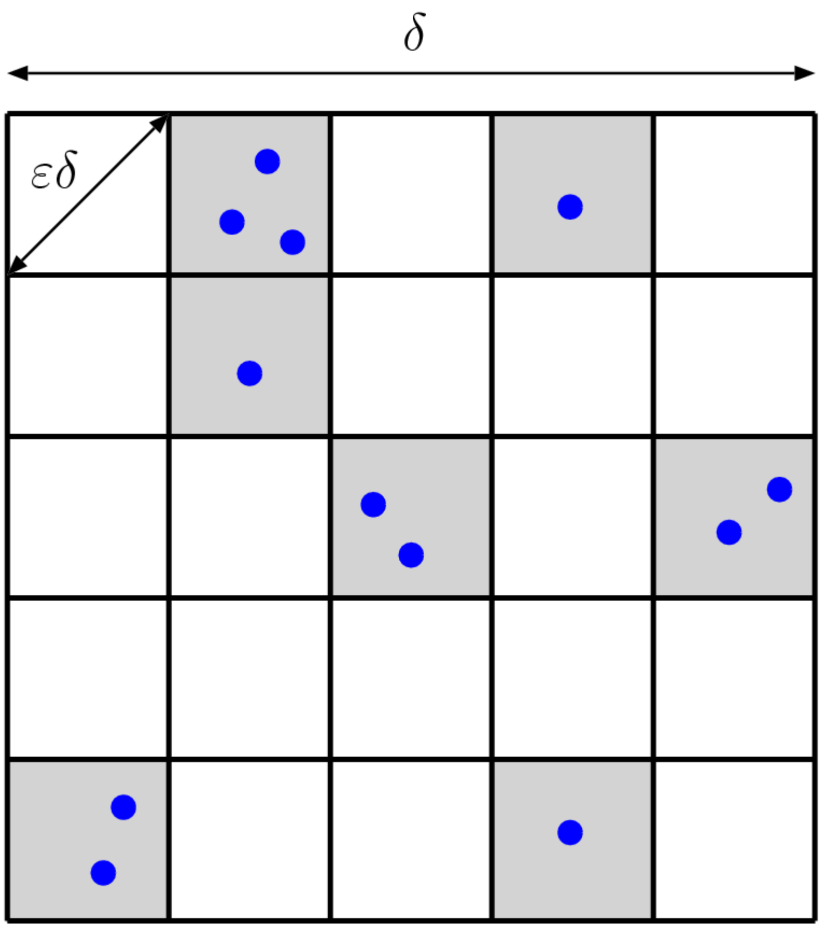
- We present a simple unifying approach that yields efficient solutions to all aforementioned proximity problems, making it clear how each property is exploited.

Quadtrees



- A quadtree is a recursive subdivision of the bounding box into 2^d equal boxes.
- Subdivisions are called quadtree boxes.
- Compression reduces storage to $O(n)$.
- Pointers can be added to allow searching the quadtree in $O(\log n)$ time.
- Preprocessing takes $O(n \log n)$ time.

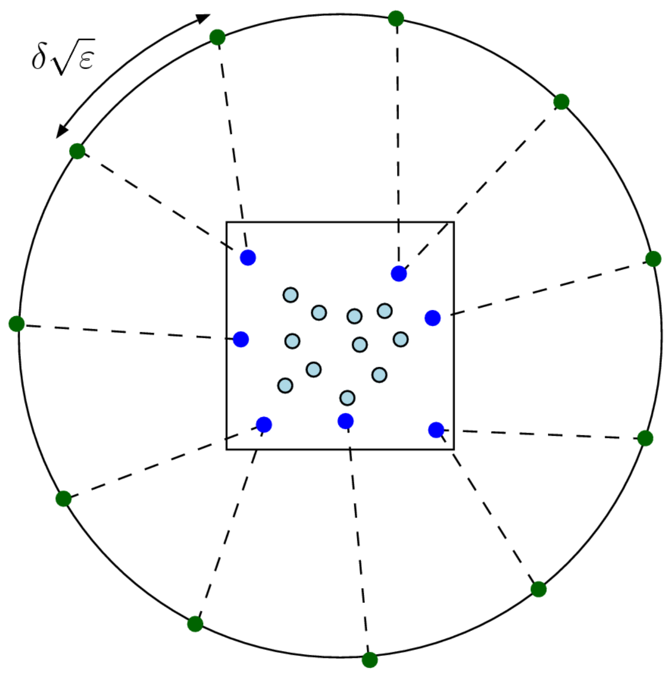
Key Lemma



- Consider a grid subdividing a quadtree box *v* of size δ into cells of size εδ.
- Let *c*(*v*) denote the number of non-empty grid cells.
- *c*(*v*) can be as high as min(*n*, $1/\varepsilon^d$).
- Fortunately, summing for all O(*n*) nodes we get the following lemma:

$$\sum_v c(v) = O(n \log(1/\varepsilon)).$$

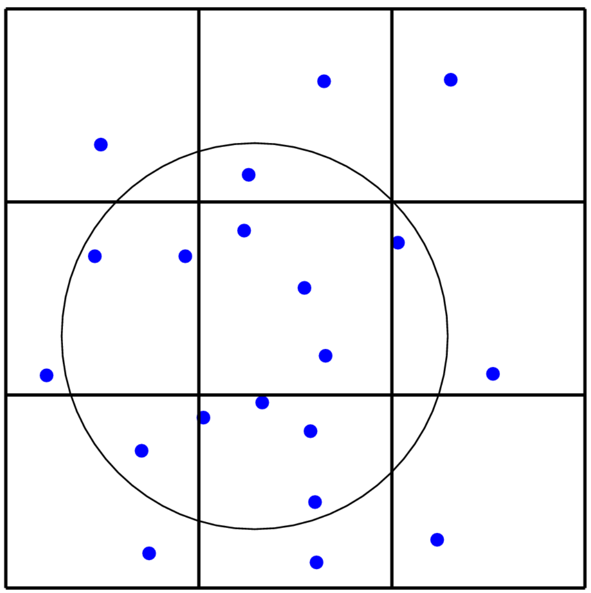
Simple Data Structure for Emptiness



- For each quadtree box v, we have two cases:
  - (i) If $c(v) \le 1/\varepsilon^{(d-1)/2}$, then we store the list of points that define $c(v)$.
  - (ii) Otherwise, we store a coreset with $O(1/\varepsilon^{(d-1)/2})$ points (Figure).

- The storage for (i) is upper bounded by $\Sigma c(v) = O(n \log(1/\varepsilon))$.
- The storage for (ii) is upper bounded by $O(n \log(1/\varepsilon))$ since only $O(n \log(1/\varepsilon) / \alpha)$ nodes can have $c(v) > \alpha$.

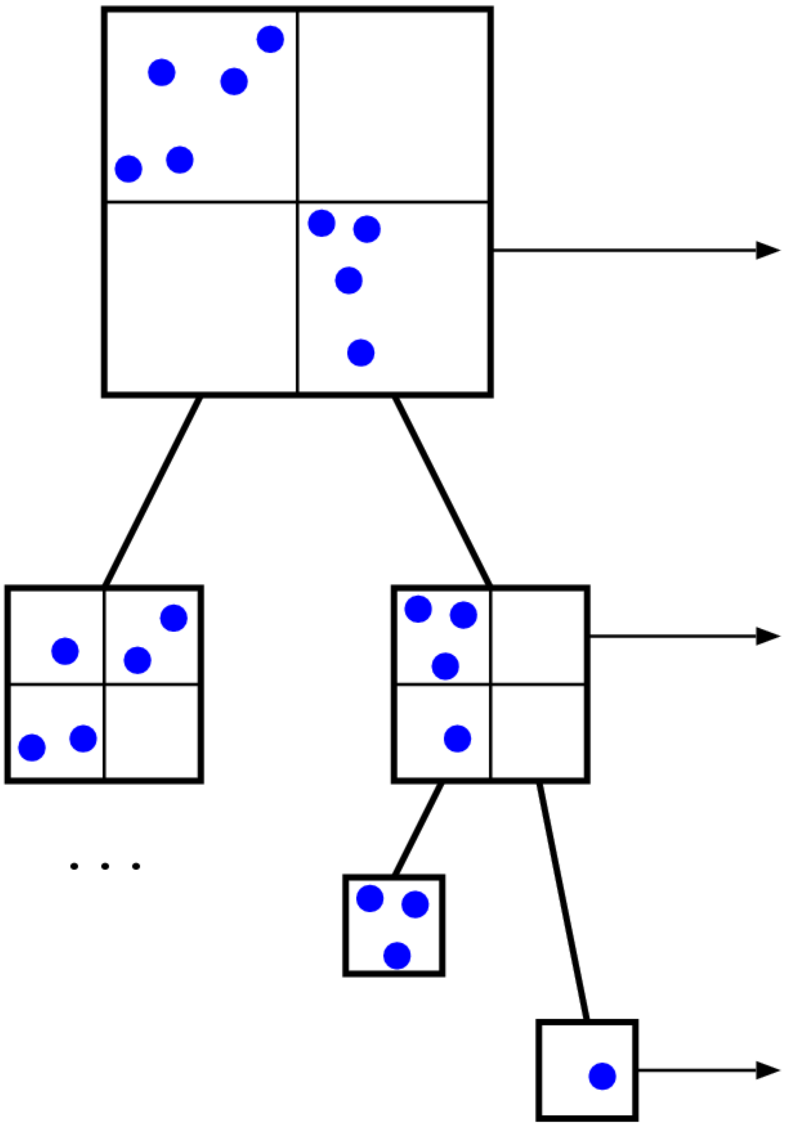
Answering Queries



- In O(log n) time, we find a set of O(1) cells of size at most $r/2$ that cover the query ball of radius r.
- For each point in each cell, we check whether it is contained in the query ball, and answer the query accordingly.

- Query takes $O(1/\varepsilon^{(d-1)/2})$ time per cell, and total time $O(\log n + 1/\varepsilon^{(d-1)/2})$.
- Correctness follows from the fact that a box of size δ only handles balls of radius at least 2δ.
- *Module*: data structure for spherical queries of size at least 2δ inside a box of size δ.

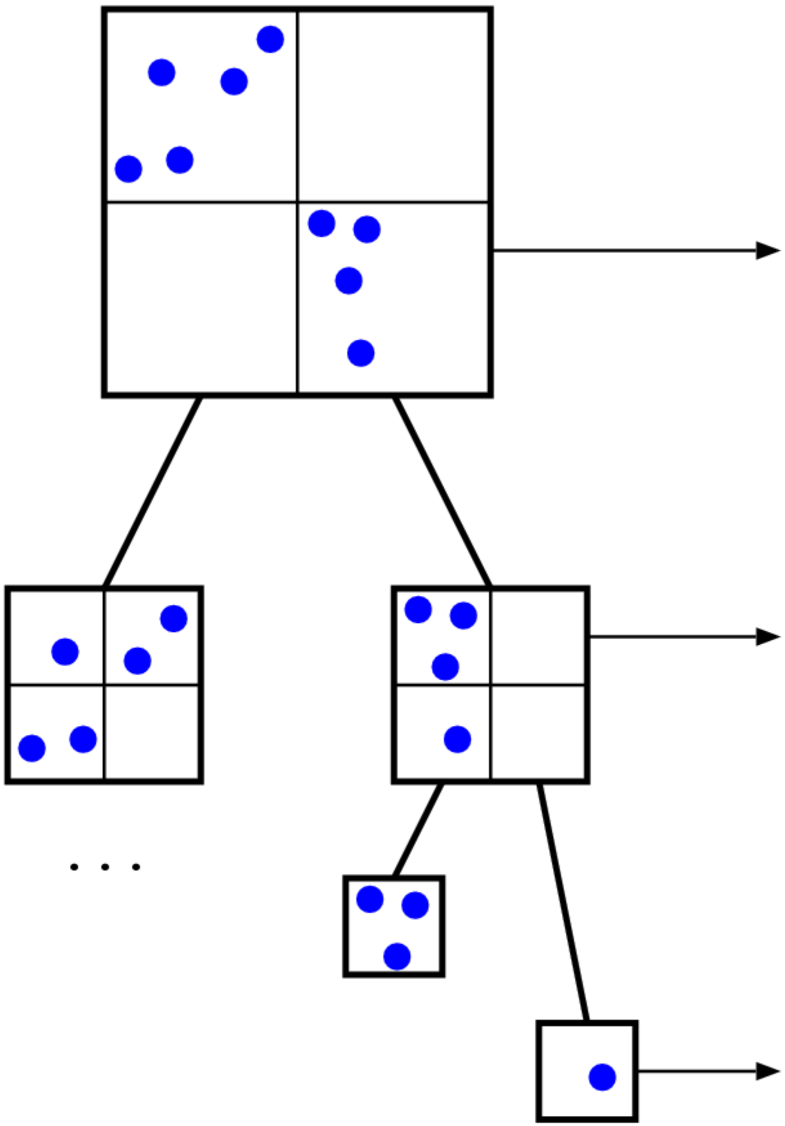
Framework



For a given parameter α:

- Nodes with $c(v) > \alpha$ store an *insensitive module*: data structure whose storage S does *not* depend on $c(v)$.
- Nodes with $c(v) \leq \alpha$ store an adaptive module: data structure whose size $s(c(v))$ goes down as $c(v)$ goes down.
- Leaf nodes just store the single point contained in them.

Storage



Total storage for each type of node:

- $c(v) > \alpha$: $O(n\ S \log(1/\varepsilon) / \alpha)$.

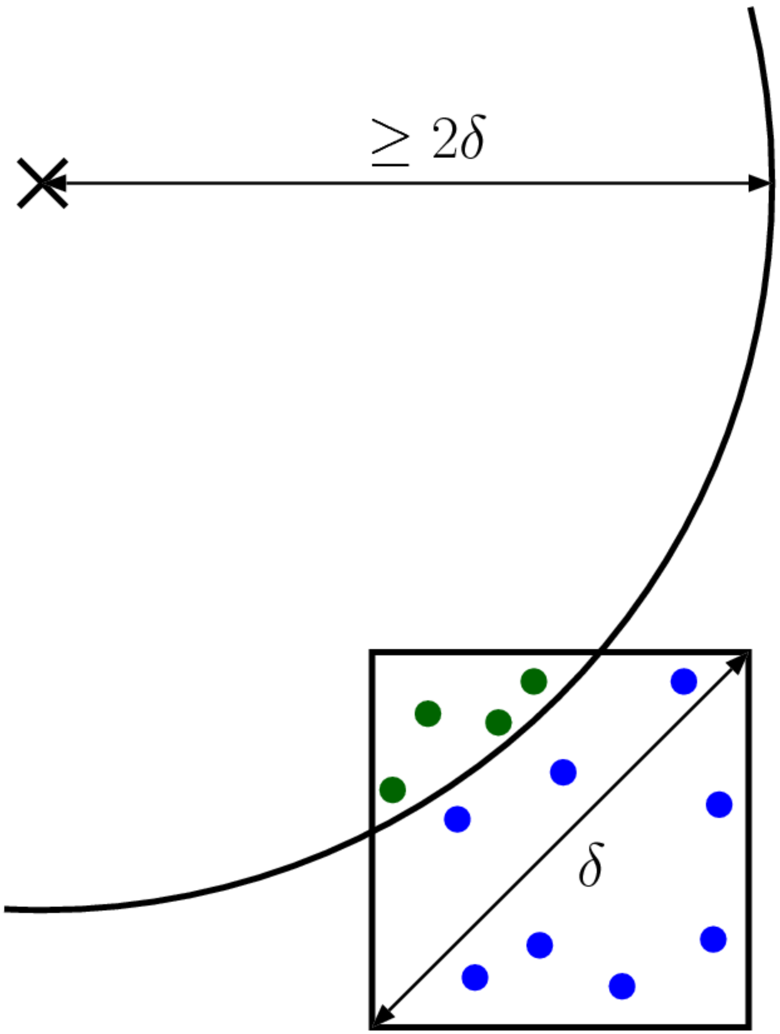
  At most $O(n \log(1/\varepsilon) / \alpha)$ nodes, each with $O(S)$ storage.

- $c(v) \leq \alpha$: $O(n\ s(\alpha) \log(1/\varepsilon) / \alpha)$.

  Each node with $O(s(\alpha))$ storage, where $s(.)$ is at least linear.

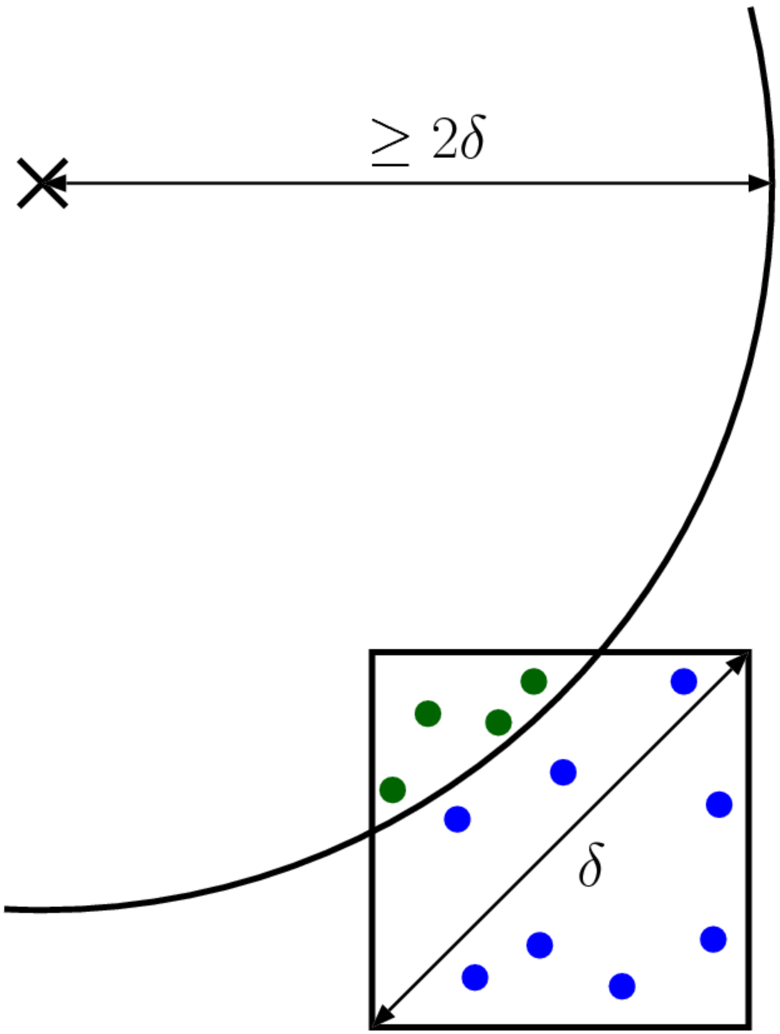
- Leaf nodes: $O(n)$.

Modules



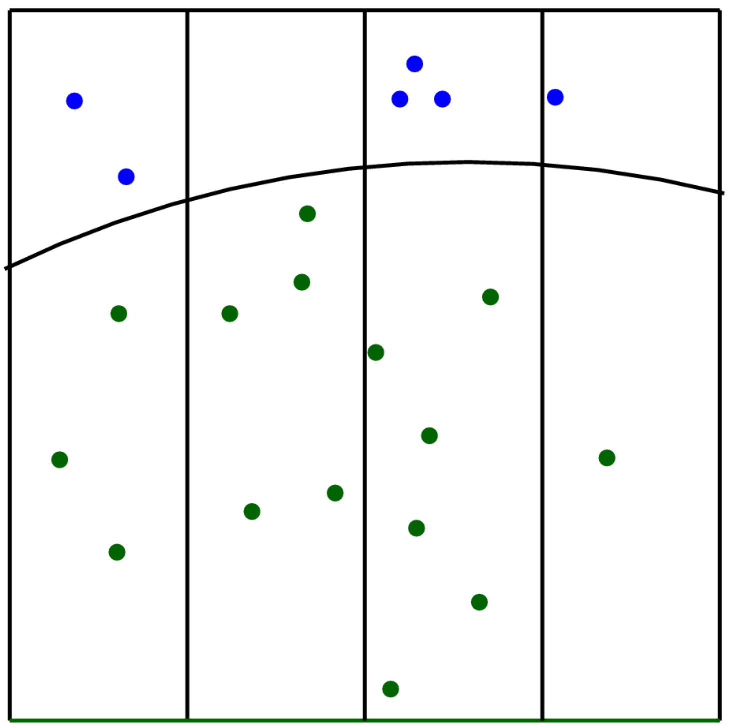
- A module is a data structure for spherical queries where all data points are inside a box of diameter δ and the query ball radius is at least 2δ.
- Points within distance εδ of the boundary may be misclassified.
- Queries are answered by locating and using a constant number of modules that cover the query ball.

Adaptive Modules



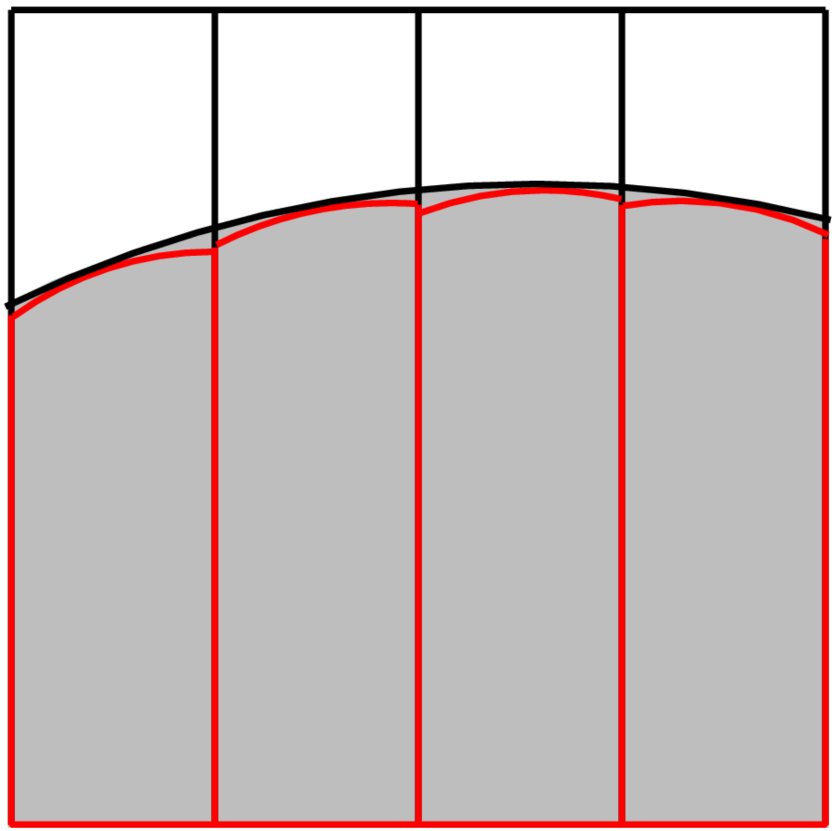
- The storage of an adaptive module depends on the number *n* of points stored in the module.
- A simple module with storage and query time O(*n*) consists of the list of points.
- A more sophisticated module which offers small improvements is a data structure for exact spherical range searching.

Insensitive Modules



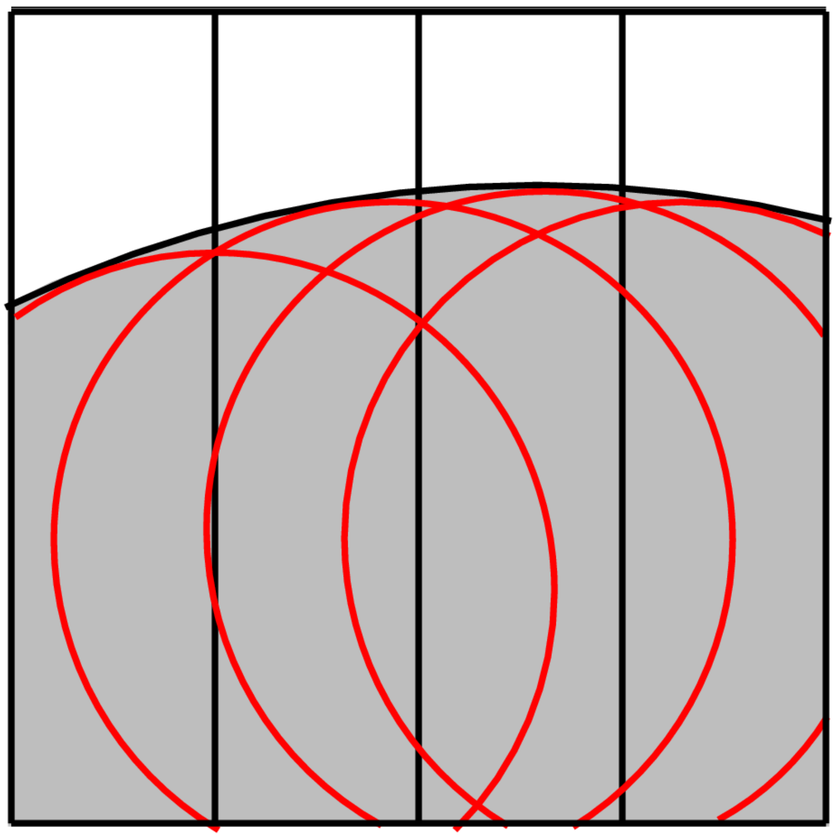
- The storage of an insensitive module does not depend on the number of points stored.
- The data structure can be build independently for each of a set of at most $1/\varepsilon$ different query radii.
- Let $\gamma \in [1,1/\varepsilon]$ be a tradeoff parameter.
- We divide the box into $1/(\varepsilon\gamma)^{d-1}$ columns where the query is answered in constant time.
- Query time = number of columns.

Generators



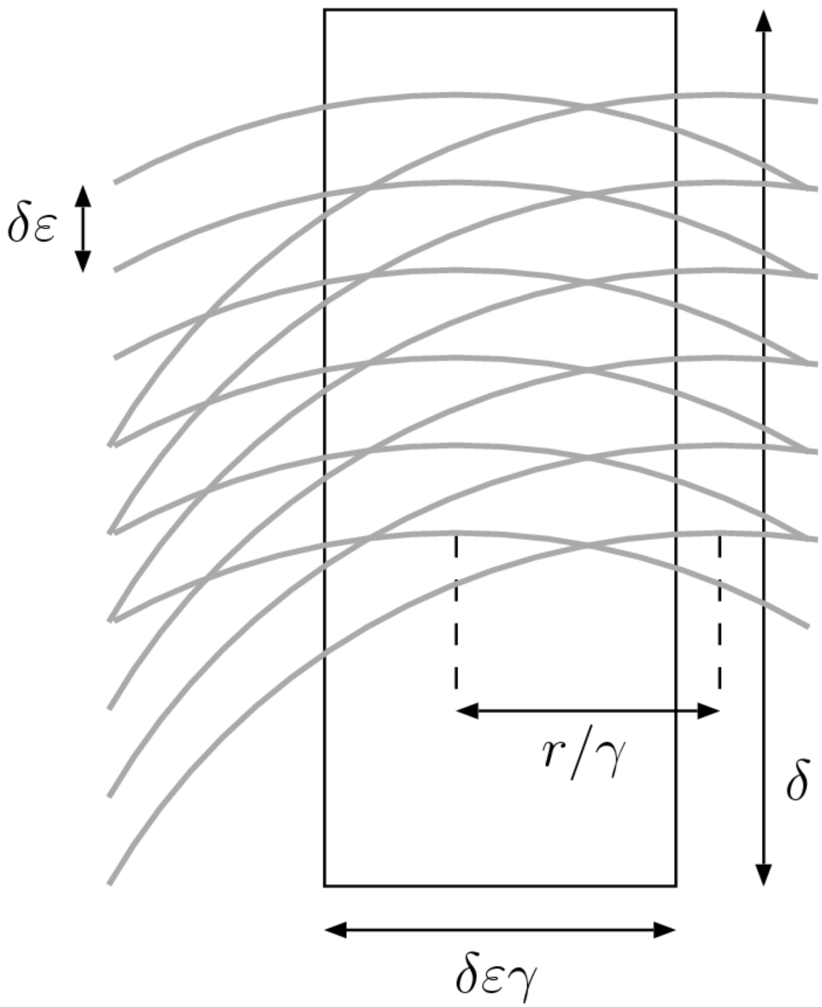
- We precompute the sum for a set of generators.
- Each generator is a (cropped) ball of radius *r*, approximately equal to the query radius.
- When answering a query in the general version, we need disjoint generators.
- When we have idempotence, generators can overlap.

Generators



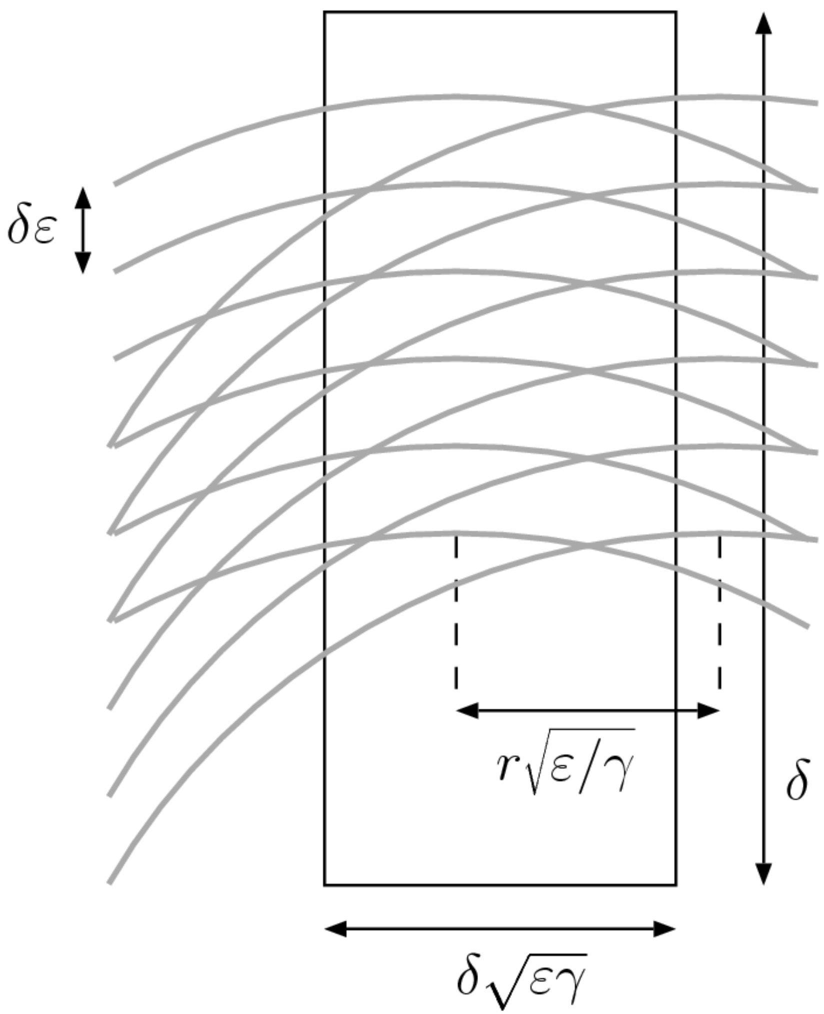
- We precompute the sum for a set of generators.
- Each generator is a (cropped) ball of radius *r*, approximately equal to the query radius.
- When answering a query in the general version, we need disjoint generators.
- When we have idempotence, generators can overlap.

General Version



- Each generator is cropped inside a column.
- Storage per column per radius: $O(\gamma^{d-1}/\varepsilon)$.
- Number of columns (query time): $O(1/(\varepsilon\gamma)^{d-1})$.
- Number of radii: $O(1+\varepsilon\gamma^2) = O(1/\varepsilon)$.
- Total storage: $O((1+\varepsilon\gamma^2) / \varepsilon^d)$.

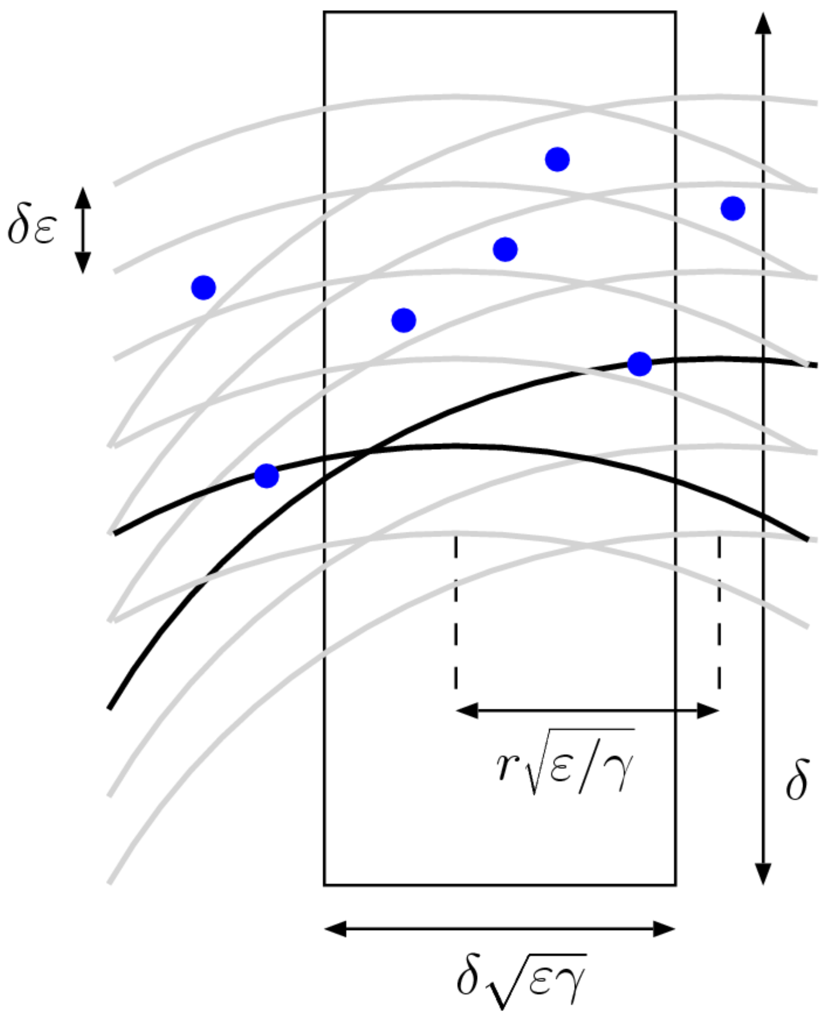
Idempotent Version



- If we do not crop the balls, then the generators are the same for every column.
- Therefore the total storage is the same as the storage per column in the general version.
- Make $\gamma \leftarrow \sqrt{\gamma/\varepsilon}$

  to get query time $O(1/(\varepsilon\gamma)^{(d-1)/2})$.
- Total storage: $O((\gamma / \varepsilon)^{(d+1)/2})$.

Emptiness Version



- In the emptiness version we can exploit monotonicity to compress the data structure.
- Only store the bottommost non-empty ball in each set.
- Reduces the storage by 1/ε.
- Query time $O(1/(\varepsilon\gamma)^{(d-1)/2})$.
- Storage: $O(\gamma^{(d+1)/2} / \varepsilon^{(d-1)/2})$.
- Approximate nearest neighbor queries reduce to O(log(1/ε)) spherical emptiness queries.

Complexities

- General spherical range query time: $\tilde{O}(1/(\varepsilon\gamma)^{d-1})$.
  Previous storage: $\tilde{O}(n\,\gamma^{d})$.
  New storage without Exact Range Searching :
  $\tilde{O}(n\,\gamma^{d-1}(1+\varepsilon\gamma^{2}))$.
- Idempotent spherical range query time: $\tilde{O}(1/(\varepsilon\gamma)^{(d-1)/2})$.
  Previous storage: $\tilde{O}(n\,\gamma^{d}/\varepsilon)$.
  New storage without ERS: $\tilde{O}(n\,\gamma^{d}/\varepsilon)$.
  New storage with ERS: $\tilde{O}(n\,\gamma^{d-1/2}/\sqrt{\varepsilon})$.
- Spherical emptiness query time: $\tilde{O}(1/(\varepsilon\gamma)^{(d-1)/2})$.
  Previous storage: $\tilde{O}(n\,\gamma^{d-1})$.
  New storage without ERS: $\tilde{O}(n\,\gamma^{d})$.
  Using ERS, query time: $\tilde{O}(1/(\varepsilon\gamma)^{(d-3)/2+1/d})$ and
  storage: $\tilde{O}(n\,\gamma^{d-2})$.

Thank you!