Multi-dimensional Index Structures

Krzysztof Dembczyński

Intelligent Decision Support Systems Laboratory (IDSS)
Poznań University of Technology, Poland

Software Development Technologies
Master studies, second semester
Academic year 2017/18 (winter course)

Review of the previous lectures

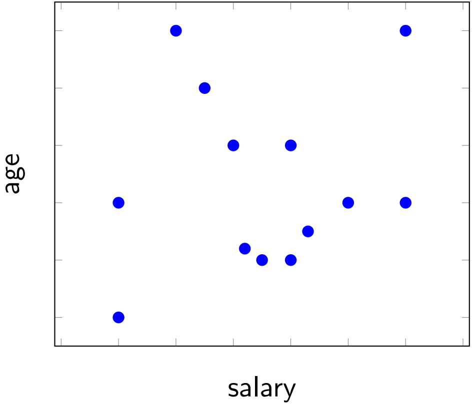
- Mining of massive datasets
- Classification and regression
- Evolution of database systems
- MapReduce

Outline

Outline

Multi-dimensional structures

- Conventional index structures are one dimensional and are not suitable for multi-dimensional search queries.



Multi-dimensional structures

- Typical applications:

Multi-dimensional structures

- Typical applications:
  - Geographic Information Systems (GIS): where-am-I queries.

Multi-dimensional structures

- Typical applications:
  - Geographic Information Systems (GIS): where-am-I queries.
  - Computer vision: find the most similar picture.

Multi-dimensional structures

- Typical applications:
  - Geographic Information Systems (GIS): where-am-I queries.
  - Computer vision: find the most similar picture.
  - Learning: decision trees, rules, nearest neighbors.

Multi-dimensional structures

- Typical applications:
  - Geographic Information Systems (GIS): where-am-I queries.
  - Computer vision: find the most similar picture.
  - Learning: decision trees, rules, nearest neighbors.
  - Recommender systems: find the most similar users/items.

Multi-dimensional structures

- Typical applications:
  - Geographic Information Systems (GIS): where-am-I queries.
  - Computer vision: find the most similar picture.
  - Learning: decision trees, rules, nearest neighbors.
  - Recommender systems: find the most similar users/items.
  - Similarity of documents: plagiarism, mirror pages, articles from the same source.

Multi-dimensional structures

- Typical applications:
  - Geographic Information Systems (GIS): where-am-I queries.
  - Computer vision: find the most similar picture.
  - Learning: decision trees, rules, nearest neighbors.
  - Recommender systems: find the most similar users/items.
  - Similarity of documents: plagiarism, mirror pages, articles from the same source.
  - ...

Multi-dimensional structures

- Typical types of multi-dimensional queries:

Multi-dimensional structures

- Typical types of multi-dimensional queries:
  - Partial match queries: for specified values for one or more dimensions find all points matching those values in those dimensions:

    `where salary = 5000 and age = 30`

Multi-dimensional structures

- Typical types of multi-dimensional queries:
  - Partial match queries: for specified values for one or more dimensions find all points matching those values in those dimensions:

```
where salary = 5000 and age = 30
```

  - Range queries: for specified ranges for one or more dimensions find all the points within those ranges:

```
where salary between 3500 and 5000
    and age between 25 and 35
```

Multi-dimensional structures

- Typical types of multi-dimensional queries:
  - Partial match queries: for specified values for one or more dimensions find all points matching those values in those dimensions:

    ```
    where salary = 5000 and age = 30
    ```

  - Range queries: for specified ranges for one or more dimensions find all the points within those ranges:

    ```
    where salary between 3500 and 5000
        and age between 25 and 35
    ```

  - Nearest-neighbor queries: find the closest one or more points to a given point.

Multi-dimensional structures

- Typical types of multi-dimensional queries:
  - Partial match queries: for specified values for one or more dimensions find all points matching those values in those dimensions:

    ```
    where salary = 5000 and age = 30
    ```

  - Range queries: for specified ranges for one or more dimensions find all the points within those ranges:

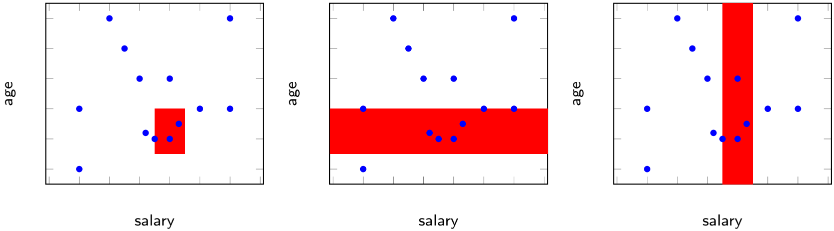
    ```
    where salary between 3500 and 5000
        and age between 25 and 35
    ```

  - Nearest-neighbor queries: find the closest one or more points to a given point.
  - Where-am-I queries: for a given point, where this point is located (in which shape).

Multi-dimensional queries with conventional indexes

- Consider a range query:

```
where salary between 3500 and 5000
and age between 25 and 35
```



- To answer the query:
  - Scan along either index at once,
  - Intersect the elements returned by indexes
- This approach produces many false hits on each index!

Nearest neighbor queries

- Brute force search:

Nearest neighbor queries

- Brute force search:
  - Given a query point q scan through each of n data points in database

Nearest neighbor queries

- Brute force search:
  - Given a query point q scan through each of n data points in database
  - Computational complexity for 1-NN query:

Nearest neighbor queries

- Brute force search:
  - Given a query point q scan through each of n data points in database
  - Computational complexity for 1-NN query: $\mathcal{O}(n)$.

Nearest neighbor queries

- Brute force search:
  - Given a query point q scan through each of n data points in database
  - Computational complexity for 1-NN query: $\mathcal{O}(n)$.
  - Computational complexity for k-NN query:

Nearest neighbor queries

- Brute force search:
  - Given a query point q scan through each of n data points in database
  - Computational complexity for 1-NN query: $\mathcal{O}(n)$.
  - Computational complexity for k-NN query: $\mathcal{O}(n \log k)$!

Nearest neighbor queries

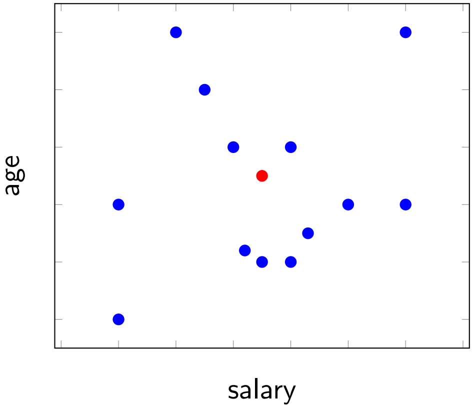
- Brute force search:
  - Given a query point q scan through each of n data points in database
  - Computational complexity for 1-NN query: $\mathcal{O}(n)$.
  - Computational complexity for k-NN query: $\mathcal{O}(n \log k)$!
- With large databases linear complexity can be too costly.

Nearest neighbor queries

- Brute force search:
  - Given a query point q scan through each of n data points in database
  - Computational complexity for 1-NN query: $\mathcal{O}(n)$.
  - Computational complexity for k-NN query: $\mathcal{O}(n \log k)$!
- With large databases linear complexity can be too costly.
- Can we do better?

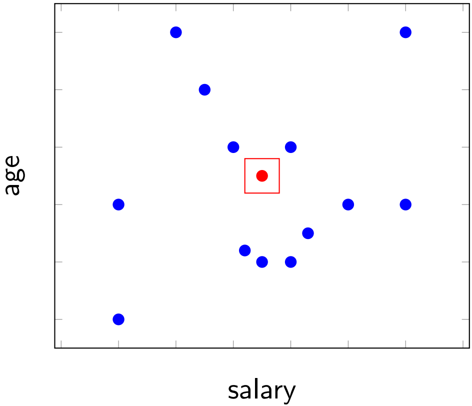
Nearest neighbor queries

- To solve the nearest neighbor search one can ask the range query and select the point closest to the target within that range.



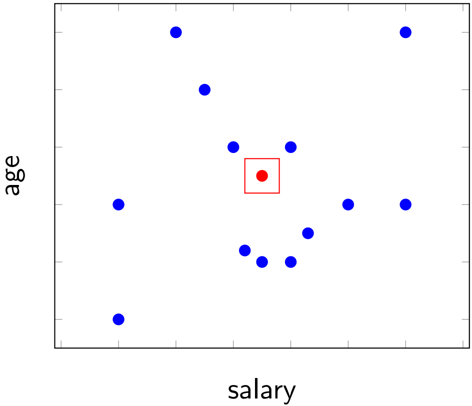
Nearest neighbor queries

- To solve the nearest neighbor search one can ask the range query and select the point closest to the target within that range.



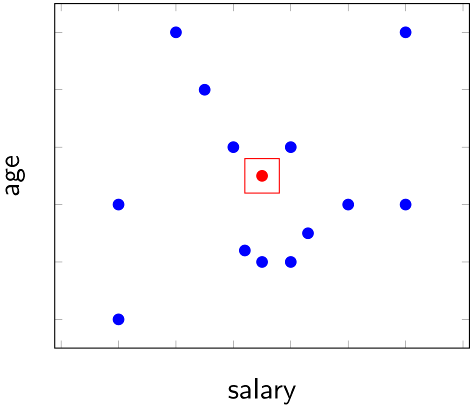
Nearest neighbor queries

- To solve the nearest neighbor search one can ask the range query and select the point closest to the target within that range.
- There are two situations we need to take into account:



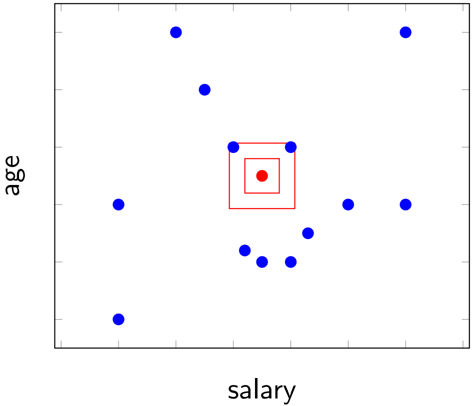
Nearest neighbor queries

- To solve the nearest neighbor search one can ask the range query and select the point closest to the target within that range.
- There are two situations we need to take into account:
  - There is no point within the selected range.



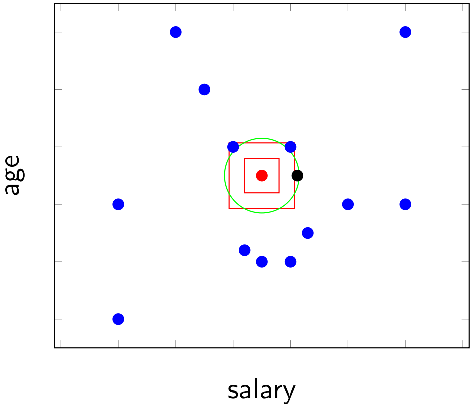
Nearest neighbor queries

- To solve the nearest neighbor search one can ask the range query and select the point closest to the target within that range.
- There are two situations we need to take into account:
  - There is no point within the selected range.



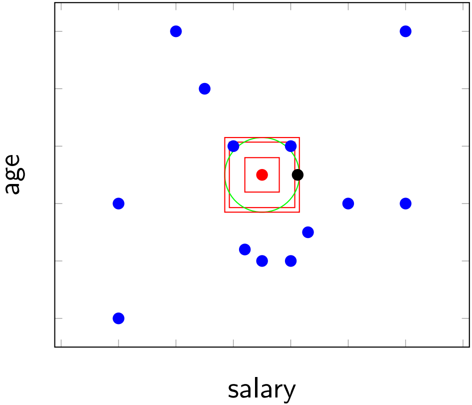
Nearest neighbor queries

- To solve the nearest neighbor search one can ask the range query and select the point closest to the target within that range.
- There are two situations we need to take into account:
    - There is no point within the selected range.
    - The closest point within the range might not be the closest point overall.



Nearest neighbor queries

- To solve the nearest neighbor search one can ask the range query and select the point closest to the target within that range.
- There are two situations we need to take into account:
    - There is no point within the selected range.
    - The closest point within the range might not be the closest point overall.



Nearest neighbor queries

- A general technique for finding the nearest neighbor:

Nearest neighbor queries

- A general technique for finding the nearest neighbor:
  - Estimate the range in which the nearest point is likely to be found.

Nearest neighbor queries

- A general technique for finding the nearest neighbor:
  - Estimate the range in which the nearest point is likely to be found.
  - Execute the corresponding range query.

Nearest neighbor queries

- A general technique for finding the nearest neighbor:
  - Estimate the range in which the nearest point is likely to be found.
  - Execute the corresponding range query.
  - If no points are found within that range, repeat with a larger range, until at least one point will be found.

Nearest neighbor queries

- A general technique for finding the nearest neighbor:
  - Estimate the range in which the nearest point is likely to be found.
  - Execute the corresponding range query.
  - If no points are found within that range, repeat with a larger range, until at least one point will be found.
  - Consider, whether there is the possibility that a closer point exists outside the range used. If so, increase appropriately the range once more and retrieve all points in the larger range to check.

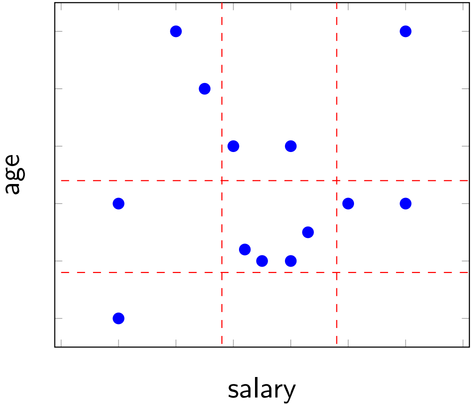
Multidimensional index structures

- Hash-table-like approaches
- Tree-like approaches

Outline

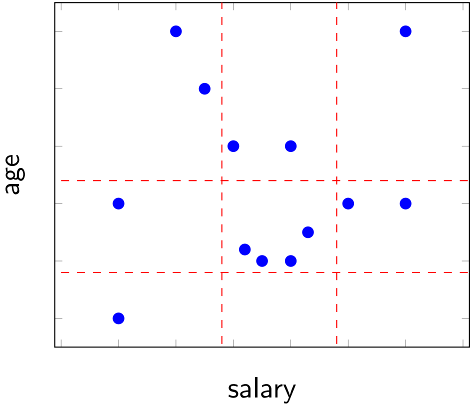
Grid files

- The space of points partitioned in a grid.



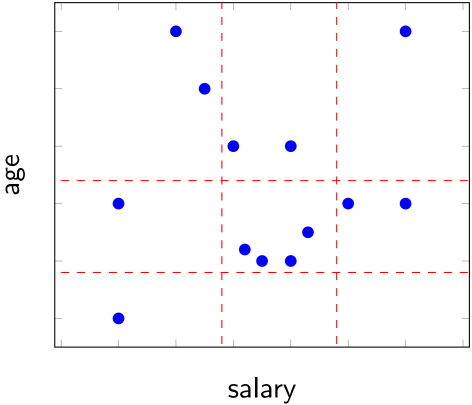
Grid files

- The space of points partitioned in a grid.
- In each dimension, grid lines partition the space into stripes.



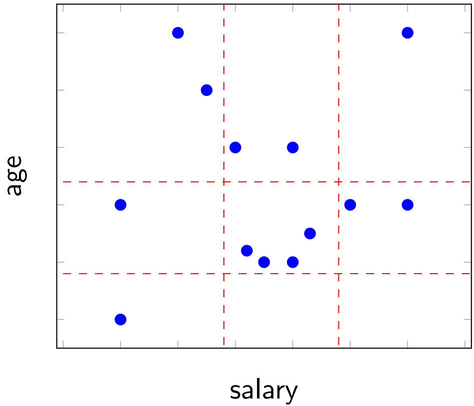
Grid files

- The space of points partitioned in a grid.
- In each dimension, grid lines partition the space into stripes.
- The number of grid lines in different dimensions may vary.



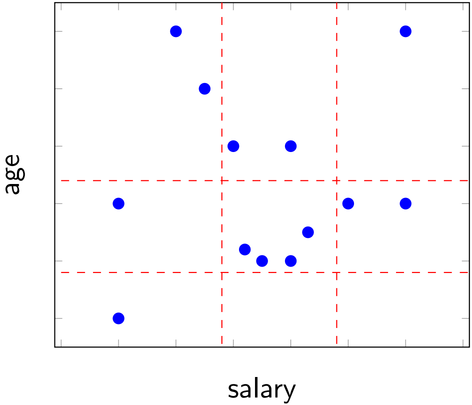
Grid files

- The space of points partitioned in a grid.
- In each dimension, grid lines partition the space into stripes.
- The number of grid lines in different dimensions may vary.
- Spacings between adjacent grid lines may also vary.



Grid files

- The space of points partitioned in a grid.
- In each dimension, grid lines partition the space into stripes.
- The number of grid lines in different dimensions may vary.
- Spacings between adjacent grid lines may also vary.
- Each region corresponds to a bucket.



Grid files

- Lookup in Grid Files:

Grid files

- Lookup in Grid Files:
  - Look at each component of a point and determine the position of the point in the grid for that dimension.

Grid files

- Lookup in Grid Files:
  - Look at each component of a point and determine the position of the point in the grid for that dimension.
  - The positions of the point in each of the dimensions together determine the bucket.

Grid files

- Lookup in Grid Files:
  - Look at each component of a point and determine the position of the point in the grid for that dimension.
  - The positions of the point in each of the dimensions together determine the bucket.
- Insertion into Grid Files:

Grid files

- Lookup in Grid Files:
  - Look at each component of a point and determine the position of the point in the grid for that dimension.
  - The positions of the point in each of the dimensions together determine the bucket.
- Insertion into Grid Files:
  - Follow the procedure for lookup of the record and place the new record to that bucket

Grid files

- Lookup in Grid Files:
  - Look at each component of a point and determine the position of the point in the grid for that dimension.
  - The positions of the point in each of the dimensions together determine the bucket.
- Insertion into Grid Files:
  - Follow the procedure for lookup of the record and place the new record to that bucket
  - If there is no room in the bucket:

Grid files

- Lookup in Grid Files:
  - Look at each component of a point and determine the position of the point in the grid for that dimension.
  - The positions of the point in each of the dimensions together determine the bucket.
- Insertion into Grid Files:
  - Follow the procedure for lookup of the record and place the new record to that bucket
  - If there is no room in the bucket:
    - Add overflow blocks to the buckets, as needed, or

Grid files

- Lookup in Grid Files:
  - Look at each component of a point and determine the position of the point in the grid for that dimension.
  - The positions of the point in each of the dimensions together determine the bucket.
- Insertion into Grid Files:
  - Follow the procedure for lookup of the record and place the new record to that bucket
  - If there is no room in the bucket:
    - Add overflow blocks to the buckets, as needed, or
    - Reorganize the structure by adding or moving the grid lines.

Accessing buckets of a grid file

- For each dimension with large number of stripes create an index over the partition values.

Accessing buckets of a grid file

- For each dimension with large number of stripes create an index over the partition values.
- Given a value v in some coordinate, search for the corresponding partition values (the lower end) and get one component of the address of the corresponding bucket.

Accessing buckets of a grid file

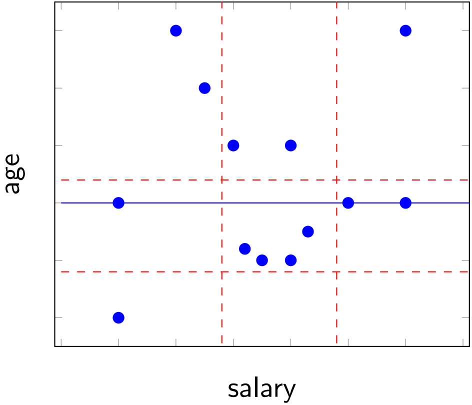
- For each dimension with large number of stripes create an index over the partition values.
- Given a value v in some coordinate, search for the corresponding partition values (the lower end) and get one component of the address of the corresponding bucket.
- Given all components of the address from each dimension, find where in the matrix (grid file) the pointer to the bucket falls.

Accessing buckets of a grid file

- For each dimension with large number of stripes create an index over the partition values.
- Given a value v in some coordinate, search for the corresponding partition values (the lower end) and get one component of the address of the corresponding bucket.
- Given all components of the address from each dimension, find where in the matrix (grid file) the pointer to the bucket falls.
- If the matrix is sparse treat it as a relation whose attributes are corners of the nonempty buckets and a final attribute representing the pointer to the bucket.

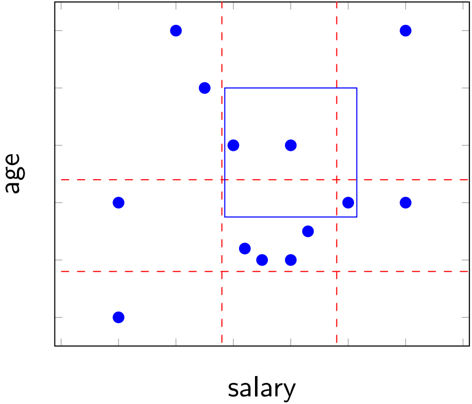
Grid files

- Partial-match queries: We need to look at all the buckets in dimension not specified in the query



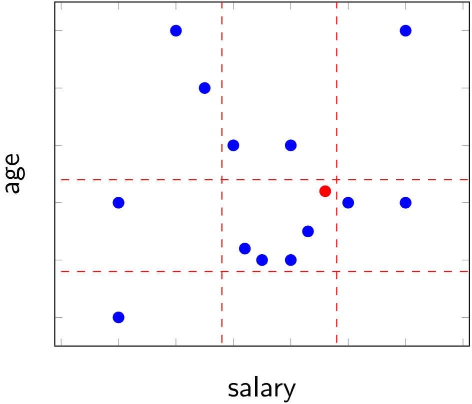
Grid files

- Range queries: We need to look at all the buckets that cover the rectangular region defined by the query



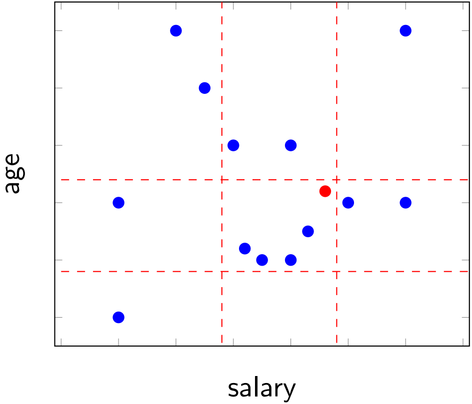
Grid files

- Nearest-neighbor queries:



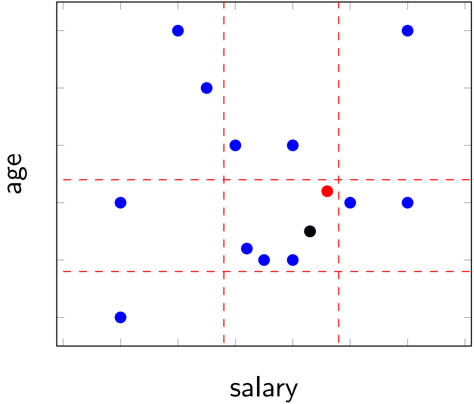
Grid files

- Nearest-neighbor queries:
  - Start with the bucket in which the point belongs.



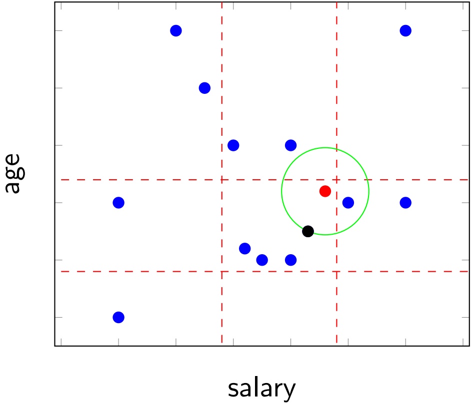
Grid files

- Nearest-neighbor queries:
  - Start with the bucket in which the point belongs.
  - If there is no point, check the adjacent buckets, for example, by spiral search; otherwise, find the nearest point to be a candidate.



Grid files

- Nearest-neighbor queries:
  - Start with the bucket in which the point belongs.
  - If there is no point, check the adjacent buckets, for example, by spiral search; otherwise, find the nearest point to be a candidate.
  - Check points in the adjacent buckets if the distance between the query point and the border of its bucket is less than the distance from the candidate.



Partitioned hash functions

- Hash functions can take a list of values as arguments, although typically there is only one argument.

Partitioned hash functions

- Hash functions can take a list of values as arguments, although typically there is only one argument.
- For example, one can compute

$$h(a, b)\,,$$

where a is an integer value and b is a character-string value, by adding the value of a to the value of the ASCII code for each character of b, dividing by the number of buckets, and taking the remainder.

Partitioned hash functions

- Hash functions can take a list of values as arguments, although typically there is only one argument.
- For example, one can compute

$$h(a, b)\,,$$

  where a is an integer value and b is a character-string value, by adding the value of a to the value of the ASCII code for each character of b, dividing by the number of buckets, and taking the remainder.
- This is, however, useful only in the queries that specify values for both a and b.

Partitioned hash functions

- Partitioned hash function h is a list of hash functions

$$(h_1, h_2, \ldots, h_n)\,,$$

such that h_i applies to a value for the i-th attribute and produces a sequence of k_i bits.

Partitioned hash functions

- Partitioned hash function h is a list of hash functions

$$(h_1, h_2, \ldots, h_n)\,,$$

  such that h_i applies to a value for the i-th attribute and produces a sequence of k_i bits.
- The bucket in which to place a point with values $(v_1, v_2, \ldots, v_n)$ for the n attributes is computed by concatenating the bit sequences:

$$h_1(v_1)h_2(v_2)\cdots h_n(v_n)$$

Partitioned hash functions

- Partitioned hash function h is a list of hash functions

$$(h_1, h_2, \ldots, h_n)\,,$$

  such that h_i applies to a value for the i-th attribute and produces a sequence of k_i bits.
- The bucket in which to place a point with values $(v_1, v_2, \ldots, v_n)$ for the n attributes is computed by concatenating the bit sequences:

$$h_1(v_1)h_2(v_2)\cdots h_n(v_n)$$

- The length of the hash is

$$\sum_{i=1}^{n} k_i = k$$

Partitioned hash functions

- **Example**: A hash table with 10-bit bucket number (1024 buckets)

Partitioned hash functions

- **Example**: A hash table with 10-bit bucket number (1024 buckets)
  - First 4-bits devoted to attribute a.

Partitioned hash functions

- **Example**: A hash table with 10-bit bucket number (1024 buckets)
  - First 4-bits devoted to attribute a.
  - Remaining 6-bits devoted to attribute b.

Partitioned hash functions

- **Example**: A hash table with 10-bit bucket number (1024 buckets)
  - First 4-bits devoted to attribute a.
  - Remaining 6-bits devoted to attribute b.
  - For a tuple with a-value A and b-value B and other attributes not involved in the hash, we could obtain, for example:

    $$h_1(A) = 0101 \quad h_2(B) = 111000$$

    This tuple hashes to bucket 0101111000.

Partitioned hash functions

- **Example**: A hash table with 10-bit bucket number (1024 buckets)
  - First 4-bits devoted to attribute a.
  - Remaining 6-bits devoted to attribute b.
  - For a tuple with a-value A and b-value B and other attributes not involved in the hash, we could obtain, for example:

    $$h_1(A) = 0101 \quad h_2(B) = 111000$$

    This tuple hashes to bucket 0101111000.
  - Moreover, we get some advantage from knowing values for any one or more of the attributes that contribute to the hash function

Partitioned hash functions

- **Example**: A hash table with 10-bit bucket number (1024 buckets)
  - First 4-bits devoted to attribute a.
  - Remaining 6-bits devoted to attribute b.
  - For a tuple with a-value A and b-value B and other attributes not involved in the hash, we could obtain, for example:

    $$h_1(A) = 0101 \quad h_2(B) = 111000$$

    This tuple hashes to bucket 0101111000.
  - Moreover, we get some advantage from knowing values for any one or more of the attributes that contribute to the hash function
    - For instance, for a value A of attribute a with $h_1 = 0101$, we know that the tuples with a-value A are in the 64 buckets whose numbers are of the form $0101 \cdots\cdots$.

Grid files vs. Partitioned hashing

- Partitioned hash tables are useless for nearest-neighbor or range queries

Grid files vs. Partitioned hashing

- Partitioned hash tables are useless for nearest-neighbor or range queries
  - The physical distance between points is not reflected by the closeness of bucket numbers.

Grid files vs. Partitioned hashing

- Partitioned hash tables are useless for nearest-neighbor or range queries
  - The physical distance between points is not reflected by the closeness of bucket numbers.
  - By imposing that kind of correspondence between physical distance and hash values we reinvent the grid file.

Grid files vs. Partitioned hashing

- Partitioned hash tables are useless for nearest-neighbor or range queries
  - The physical distance between points is not reflected by the closeness of bucket numbers.
  - By imposing that kind of correspondence between physical distance and hash values we reinvent the grid file.
- Grid files will tend to leave many buckets empty if we deal with high dimensional and/or correlated data.

Grid files vs. Partitioned hashing

- Partitioned hash tables are useless for nearest-neighbor or range queries
  - The physical distance between points is not reflected by the closeness of bucket numbers.
  - By imposing that kind of correspondence between physical distance and hash values we reinvent the grid file.
- Grid files will tend to leave many buckets empty if we deal with high dimensional and/or correlated data.
  - Hash tables are more efficient in this regard.

Outline

Multiple-key indexes

- Multiple-key index can be seen as a kind of an index of indexes, or a tree in which the nodes at each level are indexes for one attribute.

Multiple-key indexes

- Multiple-key index can be seen as a kind of an index of indexes, or a tree in which the nodes at each level are indexes for one attribute.
- The indexes on each level can be of any type of conventional indexes.

Multiple-key indexes

- Multiple-key index can be seen as a kind of an index of indexes, or a tree in which the nodes at each level are indexes for one attribute.
- The indexes on each level can be of any type of conventional indexes.
- Coverage vs. size trade-off

Multiple-key indexes

- Multiple-key index can be seen as a kind of an index of indexes, or a tree in which the nodes at each level are indexes for one attribute.
- The indexes on each level can be of any type of conventional indexes.
- Coverage vs. size trade-off
  - More attributes in search key $\rightarrow$ index covers more queries, but takes up more disk space.

Multiple-key indexes

- Multiple-key index can be seen as a kind of an index of indexes, or a tree in which the nodes at each level are indexes for one attribute.
- The indexes on each level can be of any type of conventional indexes.
- Coverage vs. size trade-off
  - More attributes in search key $\rightarrow$ index covers more queries, but takes up more disk space.
- **Example**: An index on attributes (a, b)

Multiple-key indexes

- Multiple-key index can be seen as a kind of an index of indexes, or a tree in which the nodes at each level are indexes for one attribute.
- The indexes on each level can be of any type of conventional indexes.
- Coverage vs. size trade-off
  - More attributes in search key $\rightarrow$ index covers more queries, but takes up more disk space.
- **Example**: An index on attributes (a, b)
  - Search key is (a, b) combination.

Multiple-key indexes

- Multiple-key index can be seen as a kind of an index of indexes, or a tree in which the nodes at each level are indexes for one attribute.
- The indexes on each level can be of any type of conventional indexes.
- Coverage vs. size trade-off
  - More attributes in search key $\rightarrow$ index covers more queries, but takes up more disk space.
- **Example**: An index on attributes (a, b)
  - Search key is (a, b) combination.
  - Index entries sorted by a value.

Multiple-key indexes

- Multiple-key index can be seen as a kind of an index of indexes, or a tree in which the nodes at each level are indexes for one attribute.
- The indexes on each level can be of any type of conventional indexes.
- Coverage vs. size trade-off
  - More attributes in search key $\rightarrow$ index covers more queries, but takes up more disk space.
- **Example**: An index on attributes (a, b)
  - Search key is (a, b) combination.
  - Index entries sorted by a value.
  - Entries with same a value are sorted by b value, the so-called lexicographic sort.

Multiple-key indexes

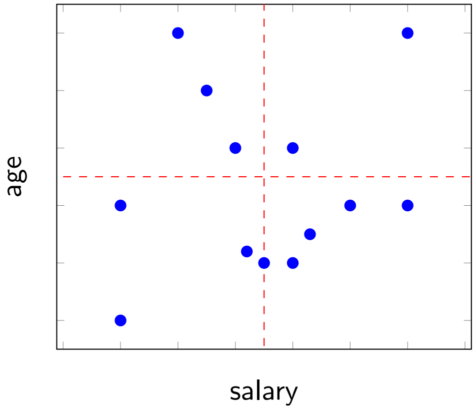
- Multiple-key index can be seen as a kind of an index of indexes, or a tree in which the nodes at each level are indexes for one attribute.
- The indexes on each level can be of any type of conventional indexes.
- Coverage vs. size trade-off
  - More attributes in search key $\rightarrow$ index covers more queries, but takes up more disk space.
- **Example**: An index on attributes (a, b)
  - Search key is (a, b) combination.
  - Index entries sorted by a value.
  - Entries with same a value are sorted by b value, the so-called lexicographic sort.
  - A query `SELECT SUM(B) FROM R WHERE A=5` is covered by the index.

Multiple-key indexes

- Multiple-key index can be seen as a kind of an index of indexes, or a tree in which the nodes at each level are indexes for one attribute.
- The indexes on each level can be of any type of conventional indexes.
- Coverage vs. size trade-off
  - More attributes in search key $\rightarrow$ index covers more queries, but takes up more disk space.
- **Example**: An index on attributes (a, b)
  - Search key is (a, b) combination.
  - Index entries sorted by a value.
  - Entries with same a value are sorted by b value, the so-called lexicographic sort.
  - A query `SELECT SUM(B) FROM R WHERE A=5` is covered by the index.
  - But for a query `SELECT SUM(A) FROM R WHERE B=5` records with $B = 5$ are scattered throughout index.

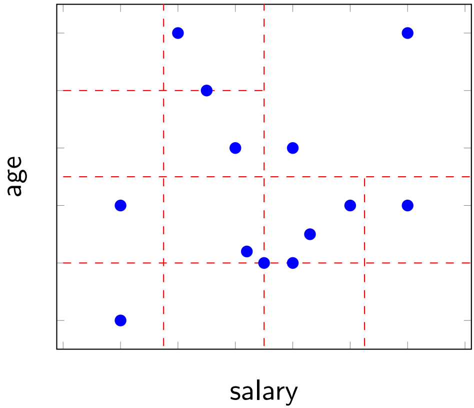
Quad trees

- Quad tree splits the space into 2^d equal sub-squares (cubes), where d is number of attributes.



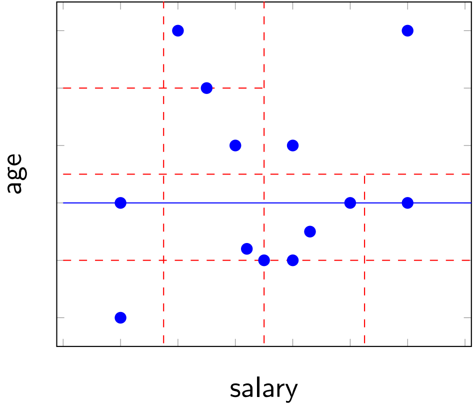
Quad trees

- Quad tree splits the space into 2^d equal sub-squares (cubes), where d is number of attributes.
- Repeat the partition until: only one pixel left; only one point left; only a few points left.



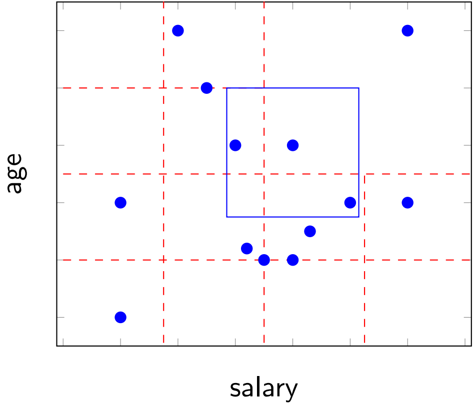
Quad trees

- Partial-match queries: We need to look at all cubes that intersect the condition of queries.



Quad trees

- Range queries: We need to look at all cubes that cover the region defined by the query



Quad trees

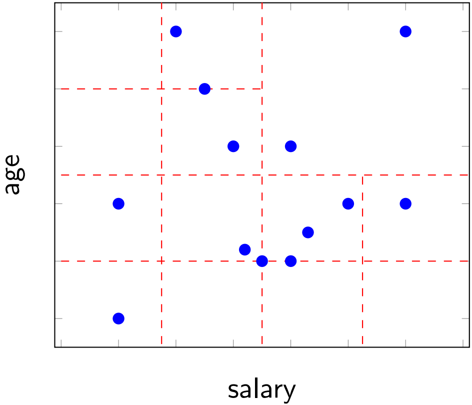
- Nearest neighbor search for point q:

```
Put  the root on the priority queue with the min distance = 0
Repeat {
    Pop the next node T from the priority queue
        if (min distance > r ) {
            the candidate is the nearest neighbor;
            break;
        }
        if (T is leaf) {
            examine point(s) in T and find the candidate;
            update r to be distance between q and the candidate;
        }
        else {
            for each child C of T {
                if( C intersects with the ball of radius r around q) {
                    compute the min distance from q to any point in C;
                    add C to the priority queue with the min distance;
                }
            }
    }
}
```

- Start search with $r = \infty$.
- Whenever a candidate point is found, update r.
- Only investigate nodes with respect to current r.

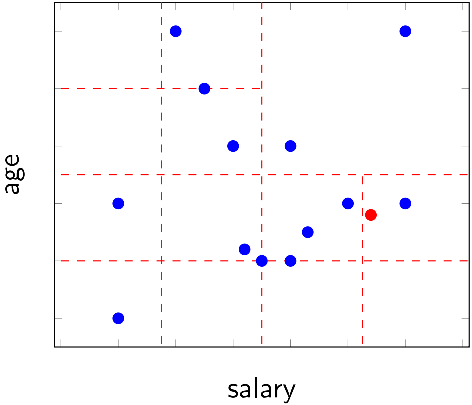
Quad trees

- Nearest neighbor search for point q:



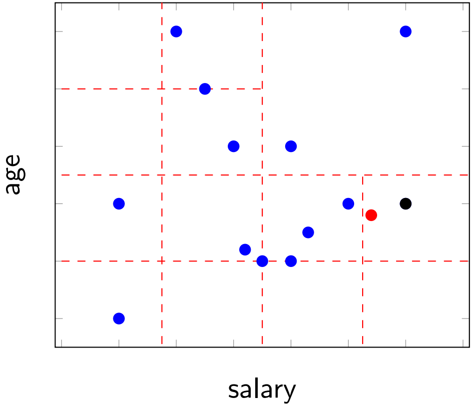
Quad trees

- Nearest neighbor search for point q:



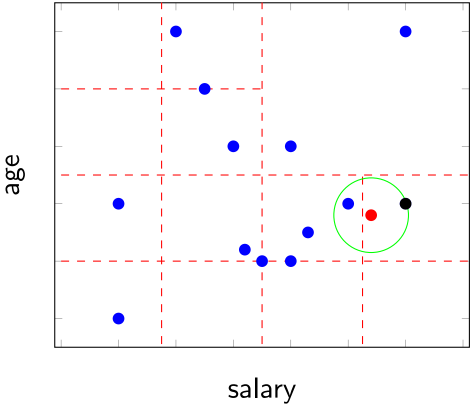
Quad trees

- Nearest neighbor search for point q:



Quad trees

- Nearest neighbor search for point q:



kd-trees

- kd-trees use only one-dimensional splits: widest or alternate dimensions in round-robin fashion.

kd-trees

- kd-trees use only one-dimensional splits: widest or alternate dimensions in round-robin fashion.
- Splits the dimension at median of the chosen region (can use the center of the region, too).

kd-trees

- kd-trees use only one-dimensional splits: widest or alternate dimensions in round-robin fashion.
- Splits the dimension at median of the chosen region (can use the center of the region, too).
- Stop criterion similar to quad trees.

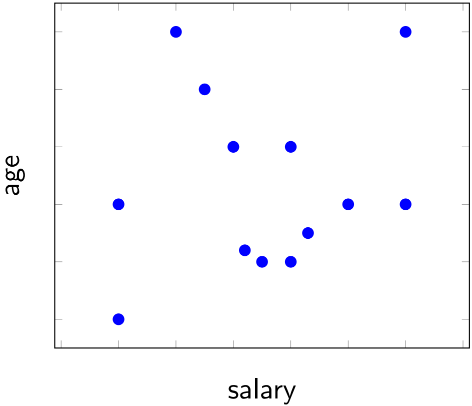
kd-trees

- kd-trees use only one-dimensional splits: widest or alternate dimensions in round-robin fashion.
- Splits the dimension at median of the chosen region (can use the center of the region, too).
- Stop criterion similar to quad trees.
- Similar operations as for quad trees.

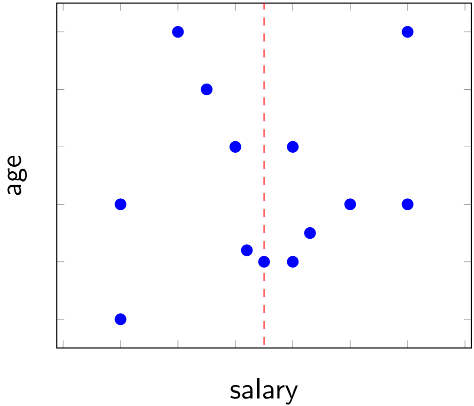
kd-trees

- kd-trees use only one-dimensional splits: widest or alternate dimensions in round-robin fashion.
- Splits the dimension at median of the chosen region (can use the center of the region, too).
- Stop criterion similar to quad trees.
- Similar operations as for quad trees.
- Advantages: no (or less) empty spaces, only linear space.

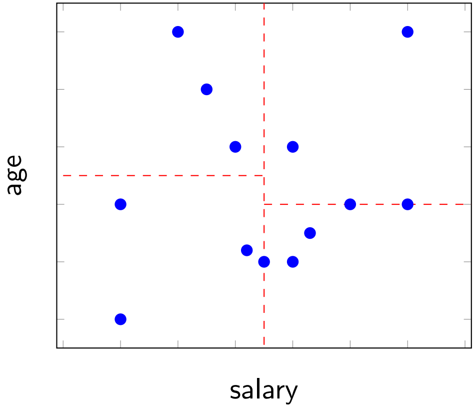
kd-trees



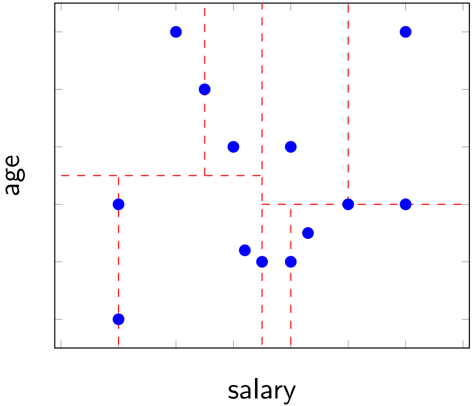
kd-trees



kd-trees



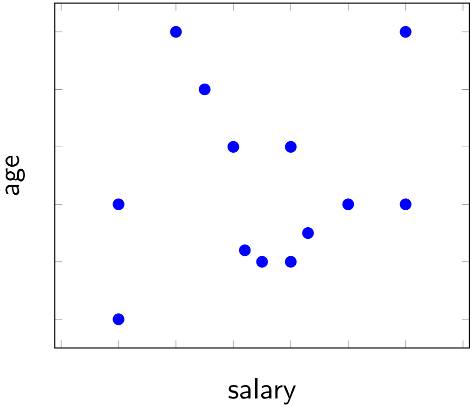
kd-trees



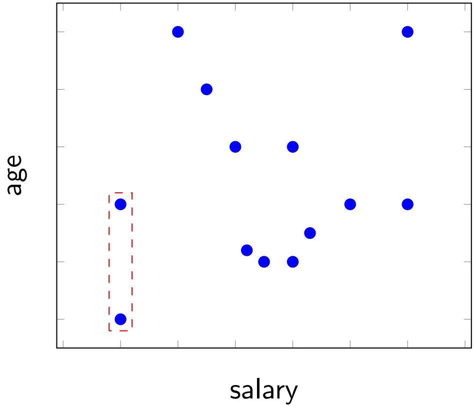
R-trees

- Similar in construction to B-trees.
- A kind of bottom-up approach (where kd-tree are top-down).
- Suitable for where-am-I queries, but also for the other types of queries (similar operations as before).
- Can deal with points and shapes.
- Avoid empty spaces.
- The regions may overlap.
- Work well in low dimensions, but may have problems with high. dimensions.

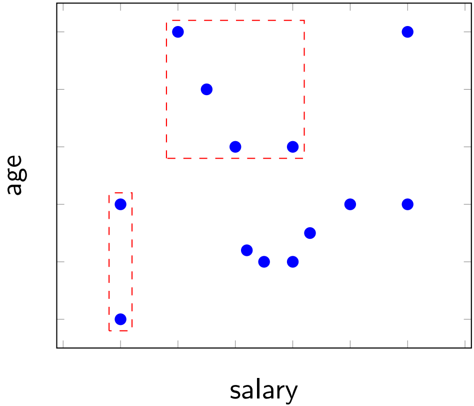
R-trees



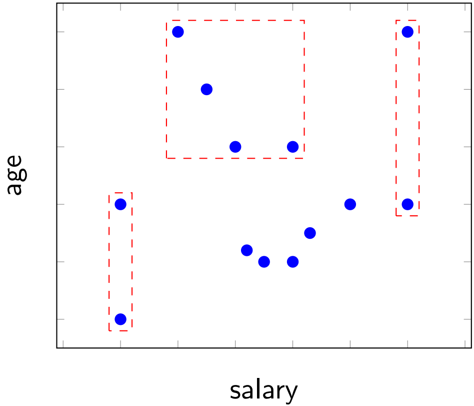
R-trees



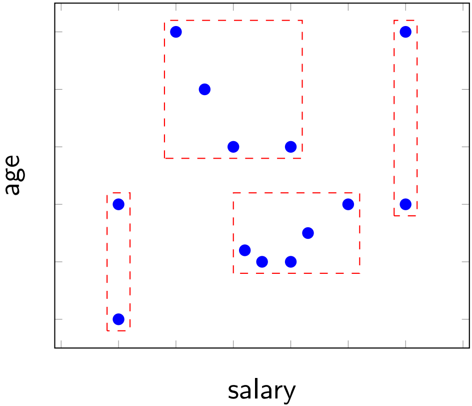
R-trees



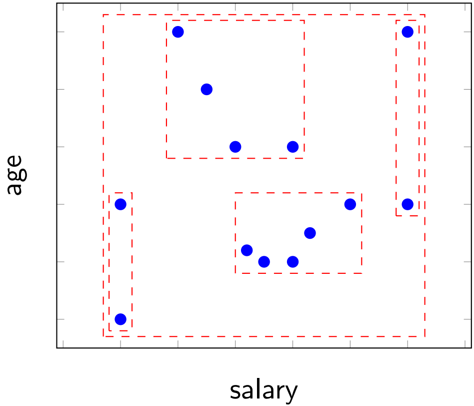
R-trees



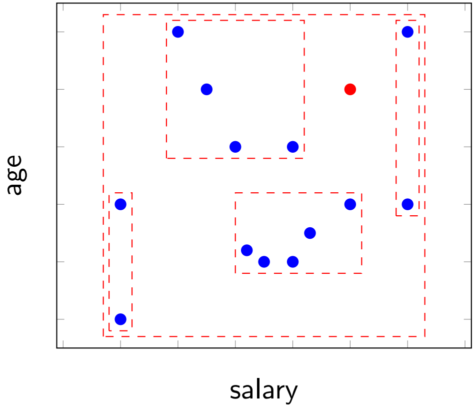
R-trees



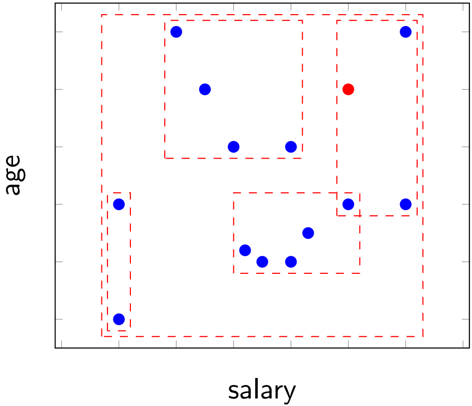
R-trees



R-trees



R-trees



Additional aspects of multidimensional indexes

- Adaptation to secondary storage.
- Balancing of the tree structures.
- Storing data only in leaves or in internal nodes and leaves.
- Many variations of the structures presented.

Problems with nearest neighbor search

- Exponential query time
  - The query time is from $\log n$ to $\mathcal{O}(n)$, but can be exponential in d.
  - Tree structures are good when $n \gg 2^d$.
  - The curse of dimensionality.
- Solution: Approximate nearest neighbor search.

The curse of dimensionality

- In high-dimensional spaces almost all pairs of points are equally far away from one another.

The curse of dimensionality

- In high-dimensional spaces almost all pairs of points are equally far away from one another.
- In other words, the neighborhood becomes very large

The curse of dimensionality

- In high-dimensional spaces almost all pairs of points are equally far away from one another.
- In other words, the neighborhood becomes very large
- **Example**:

The curse of dimensionality

- In high-dimensional spaces almost all pairs of points are equally far away from one another.
- In other words, the neighborhood becomes very large
- **Example**:
  - Task: Find the 5-nearest neighbor in the unit hypercube.

The curse of dimensionality

- In high-dimensional spaces almost all pairs of points are equally far away from one another.
- In other words, the neighborhood becomes very large
- **Example**:
  - Task: Find the 5-nearest neighbor in the unit hypercube.
  - There are 5000 points uniformly distributed.

The curse of dimensionality

- In high-dimensional spaces almost all pairs of points are equally far away from one another.
- In other words, the neighborhood becomes very large
- **Example**:
  - Task: Find the 5-nearest neighbor in the unit hypercube.
  - There are 5000 points uniformly distributed.
  - The query point: The origin of the space.

The curse of dimensionality

- In high-dimensional spaces almost all pairs of points are equally far away from one another.
- In other words, the neighborhood becomes very large
- **Example**:
  - Task: Find the 5-nearest neighbor in the unit hypercube.
  - There are 5000 points uniformly distributed.
  - The query point: The origin of the space.
  - For 1-dimensional hypercube (line), the average distance to capture all 5 nearest neighbors is $5/5000 = 0.001$.

The curse of dimensionality

- In high-dimensional spaces almost all pairs of points are equally far away from one another.
- In other words, the neighborhood becomes very large
- **Example**:
  - Task: Find the 5-nearest neighbor in the unit hypercube.
  - There are 5000 points uniformly distributed.
  - The query point: The origin of the space.
  - For 1-dimensional hypercube (line), the average distance to capture all 5 nearest neighbors is $5/5000 = 0.001$.
  - For 2 dimensional hypercube, we must go $\sqrt{0.001}$ in each direction to get a square that contains 0.001 of the volume.

The curse of dimensionality

- In high-dimensional spaces almost all pairs of points are equally far away from one another.
- In other words, the neighborhood becomes very large
- **Example**:
  - Task: Find the 5-nearest neighbor in the unit hypercube.
  - There are 5000 points uniformly distributed.
  - The query point: The origin of the space.
  - For 1-dimensional hypercube (line), the average distance to capture all 5 nearest neighbors is $5/5000 = 0.001$.
  - For 2 dimensional hypercube, we must go $\sqrt{0.001}$ in each direction to get a square that contains 0.001 of the volume.
  - In general, for d dimensions, we must go $(0.001)^{\frac{1}{d}}$.

The curse of dimensionality

- In high-dimensional spaces almost all pairs of points are equally far away from one another.
- In other words, the neighborhood becomes very large
- **Example**:
  - Task: Find the 5-nearest neighbor in the unit hypercube.
  - There are 5000 points uniformly distributed.
  - The query point: The origin of the space.
  - For 1-dimensional hypercube (line), the average distance to capture all 5 nearest neighbors is $5/5000 = 0.001$.
  - For 2 dimensional hypercube, we must go $\sqrt{0.001}$ in each direction to get a square that contains 0.001 of the volume.
  - In general, for d dimensions, we must go $(0.001)^{\frac{1}{d}}$.
  - For instance, for $d = 20$, it is 0.707, and for $d = 200$, it is 0.966.

Outline

Summary

- Multi-dimensional index structures:

Summary

- Multi-dimensional index structures:
  - Applications: partial match queries, range queries, where-am-I-queries, nearest-neighbor search.

Summary

- Multi-dimensional index structures:
  - Applications: partial match queries, range queries, where-am-I-queries, nearest-neighbor search.
  - Approaches: hash table-based, tree-like structures.

Summary

- Multi-dimensional index structures:
  - Applications: partial match queries, range queries, where-am-I-queries, nearest-neighbor search.
  - Approaches: hash table-based, tree-like structures.
  - Work good for low-dimensional problems – curse of dimensionality.

Bibliography

- H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book. Second Edition*.
  Pearson Prentice Hall, 2009
- Z. Królikowski. *Hurtownie danych: logiczne i fizyczne struktury danych*.
  Wydawnictwo Politechniki Poznańskiej, 2007
- P. Indyk. Algorithms for nearest neighbor search