Finding Similar Items III

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

Software Development Technologies
Master studies, second semester
Academic year 2018/19 (winter course)
Review of the previous lectures

• Mining of massive datasets.
• Classification and regression.
• Evolution of database systems.
• MapReduce
• MapReduce in Apache Spark
• Nearest neighbor search:
Review of the previous lectures

• Mining of massive datasets.
• Classification and regression.
• Evolution of database systems.
• MapReduce
• MapReduce in Apache Spark
• Nearest neighbor search:
  ▶ Minhash technique
Review of the previous lectures

• Mining of massive datasets.
• Classification and regression.
• Evolution of database systems.
• MapReduce
• MapReduce in Apache Spark
• Nearest neighbor search:
  ▶ Minhash technique
  ▶ Locality-sensitive hashing with minhash
Review of the previous lectures

• Mining of massive datasets.
• Classification and regression.
• Evolution of database systems.
• MapReduce
• MapReduce in Apache Spark
• Nearest neighbor search:
  ▶ Minhash technique
  ▶ Locality-sensitive hashing with minhash
  ▶ Distance measures
Review of the previous lectures

• Mining of massive datasets.
• Classification and regression.
• Evolution of database systems.
• MapReduce
• MapReduce in Apache Spark
• Nearest neighbor search:
  ▶ Minhash technique
  ▶ Locality-sensitive hashing with minhash
  ▶ Distance measures
  ▶ Theory of LSH
Review of the previous lectures

• Mining of massive datasets.
• Classification and regression.
• Evolution of database systems.
• MapReduce
• MapReduce in Apache Spark
• Nearest neighbor search:
  ▶ Minhash technique
  ▶ Locality-sensitive hashing with minhash
  ▶ Distance measures
  ▶ Theory of LSH
  ▶ LSH families for other distance measures
Outline

1. Motivation

2. Hash Structures for Multidimensional data

3. Tree Structures for Multidimensional Data

4. The curse of dimensionality

5. Summary
Outline

1 Motivation

2 Hash Structures for Multidimensional data

3 Tree Structures for Multidimensional Data

4 The curse of dimensionality

5 Summary
Multi-dimensional structures

- To speed up the **exact** search of **nearest neighbors** we need additional data structures.

![Diagram showing a scatter plot with age on the y-axis and salary on the x-axis.]
Multi-dimensional structures

• To speed up the **exact** search of **nearest neighbors** we need additional data structures.
• Conventional index structures are one dimensional and are not suitable for multi-dimensional search queries.

![Age vs Salary Scatter Plot]

*Note: The scatter plot illustrates the relationship between age and salary, with data points plotted on a two-dimensional graph.*
Multi-dimensional structures

- Besides nearest-neighbor queries we distinguish other 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
  - Where-am-I queries: for a given point, where this point is located (in which shape).
Multi-dimensional structures

• Besides **nearest-neighbor** queries we distinguish other 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

- Besides **nearest-neighbor** queries we distinguish other 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
• Besides **nearest-neighbor** queries we distinguish other 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
  ▶ **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

- 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.

![Diagram](image-url)
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:

```
<table>
<thead>
<tr>
<th>age</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
</tr>
</tbody>
</table>
```

![Diagram showing a scatter plot with points and a target point within a 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.

![Age vs Salary Plot](image.png)
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.
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:
  
  ▶ 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.
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.
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

1 Motivation

2 Hash Structures for Multidimensional data

3 Tree Structures for Multidimensional Data

4 The curse of dimensionality

5 Summary
Grid files

- The space of points partitioned in a grid.

![Plot of age vs salary with grid lines]
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.
• 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.
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.
• **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.
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
• 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.
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.
• 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.
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.
Outline

1. Motivation
2. Hash Structures for Multidimensional data
3. Tree Structures for Multidimensional Data
4. The curse of dimensionality
5. Summary
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

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.
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.
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 → index covers more queries, but takes up more disk space.
• Example: An index on attributes \((a, b)\)
• 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 → 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 → 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 → 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 \(\text{SELECT SUM}(B) \text{ FROM R WHERE } A=5\) 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 → 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 \(\text{SELECT SUM}(B) \text{ FROM } R \text{ WHERE } A=5\) is covered by the index.
  ▶ But for a query \(\text{SELECT SUM}(A) \text{ FROM } R \text{ WHERE } B=5\) 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 $q$: 

```
+---------+---------+---------+---------+
<table>
<thead>
<tr>
<th>salary</th>
<th>age</th>
<th>salary</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>24 / 33</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Quad trees

- Nearest neighbor search for point $q$: 

![Diagram of a quad tree with points plotted on a 2D plane with axes labeled 'age' and 'salary'.]
• Nearest neighbor search for point $q$: 

![Quad tree diagram](image)
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

![Graph showing age vs salary with some data points plotted.]
kd-trees

age

salary
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.
1 Motivation

2 Hash Structures for Multidimensional data

3 Tree Structures for Multidimensional Data

4 The curse of dimensionality

5 Summary
Problems with nearest neighbor search

- Exponential query time
  - The query time is from $\log n$ to $O(n)$, 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.
• 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

1 Motivation

2 Hash Structures for Multidimensional data

3 Tree Structures for Multidimensional Data

4 The curse of dimensionality

5 Summary
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


- P. Indyk. Algorithms for nearest neighbor search