Question 1: Denormalization and rewriting (1 P.)

a) For this question, we will take another look at the schema from sheet 2:

Store:	[SID, City, EID]
Employee:	[EID, Name, Salary, SID]
Article:	[AID, Name, Producer, Price, ProductionDate]
Inventory:	[AID, SID, Count]
Invoice:	[IID, SID, Customer]
Item:	[IID, SID, AID, Count]

How can you denormalize this schema, to answer the following queries efficiently? Which downsides will this have?
Describe the normal forms of the original and changed schema in your answer. Also note which additional steps have to be taken to keep the data consistent.

Required submission: Denormalized table schema; Explanation and downsides of denormalization; Query after denormalization; Explanation on how to keep data consistent; Normal form of original table schema; Normal form of new table schema.

- Q7: All employees per city
- Q8: The price of all items in given invoice IID = 5.

Solution

To answer these queries efficiently, we can store all required data in one table to avoid joins. The drawback is, that we store redundant information and special care has to be provided to keep it consistent.

Q7: The original query could look like this

```
SELECT s.City city, e.* FROM Store s, Employee e
  WHERE s.SID = e.SID
```

To avoid this join, we could store the city in the employee table. This would require more storage, but would not incur additional overhead–if cities are only added but never changed. As the city is only transitively functional dependent of the `EID`, the adapted relation would be in 2NF.

Q8: The original query could look like this

```
SELECT i.*, (i.Count*a.Price) FROM Item i, Article a
WHERE i.AID = a.AID AND i.IID = 5
```

To avoid this join, the price of a single article could be stored with the item. But the price could be changed regularly, thereby two tables would have to be modified to ensure consistency. If we also want to prevent the multiplication, we could store the total price of this item, instead of a single price. But then we'd also have to modify the total price if the count changes, which is improbable, as we are working with an invoice, which should not be changed after creating it. As above, the new relation would be in 1NF.

b) Rewrite the following queries to more efficient queries, returning the same result set. Consider the conditions noted for each of the queries.

Required submission: Simplified query; Explanation why changes where made.

i) Database as specified above

```
SELECT DISTINCT *
FROM Employee
```

Solution

As all tuples–especially with primary key–are requested, the `DISTINCT` does not serve any purpose and can be discarded.

ii) There is no index.

```
SELECT MAX(E.Salary)
FROM Employee E
GROUP BY E.EID
HAVING E.EID=2002
```

Solution

As we are grouping by the primary key, only one tuple will be in each group. We can thereby discard the `GROUP BY`, `HAVING` and `MAX()` statements and move the condition into the `WHERE` clause.

iii) Attribute `SID` in `Employee` is a foreign key, referencing `SID` in `Store`.

```
SELECT S.SID
FROM Store S, Employee E
WHERE  S.SID = E.SID
```

Solution

The join is unnecessary, as we could directly select `E.SID`. There can be no `NULL` values, as this attribute is a foreign key.

Question 2: Min-Hashing (1 P.)

Implement Min-Hashing in a language of your choice. Your program has to take two text files as input and parse them into sets of words. The program has to calculate and display:

- Jaccard coefficient between the two texts.
- The similarity estimated by Min-Hashing with k different hash functions (for each $k \in [1, 6]$)
- The similarity estimated by Min-Hashing with k different min values of one hash functions (for each $k \in [1, 6]$)

You can use the template in OLAT, which already parses the files and provides 6 hash functions. Submit the code and the output of your program when executed with the two data files provided in OLAT. If you do not use the template, also submit instructions on how to compile and execute your program.

Required submission: Source Code in separate file; Output after executing code with provided data files; (Compile/Execute instructions if not using template).

Solution

When using the template, the output should be:

```
Min Hashing exercise:
==========================================
Reading file: file1.txt
The quick brown fox jumps over the lazy dog
Reading file: file2.txt
The small brown fox jumps over the fat dog
==========================================
Calculating Jaccard similarity: 0.6363636363636364
==========================================
Calculating similarity for k = 1 hash functions: 1.0
Calculating similarity for k = 2 hash functions: 1.0
Calculating similarity for k = 3 hash functions: 1.0
Calculating similarity for k = 4 hash functions: 0.75
Calculating similarity for k = 5 hash functions: 0.8
Calculating similarity for k = 6 hash functions: 0.8333333333333334
==========================================
Calculating similarity for k = 1 hash values: 1.0
Calculating similarity for k = 2 hash values: 0.5
Calculating similarity for k = 3 hash values: 0.6666666666666666
Calculating similarity for k = 4 hash values: 0.75
Calculating similarity for k = 5 hash values: 0.8
Calculating similarity for k = 6 hash values: 0.8333333333333334
==========================================
```

Question 3: Quadtrees (1 P.)

a) Give 8 points, with coordinates in the range $[0, 100]$, and sort them into two insertion orders. The first should have less empty leaf nodes if inserted into a quadtree than if they were inserted into a PR quadtree. The second one the other way around.

Submit the points, the two insertion orders and draw the (PR) quadtrees. You can either draw them as tree or as the grid visualization.

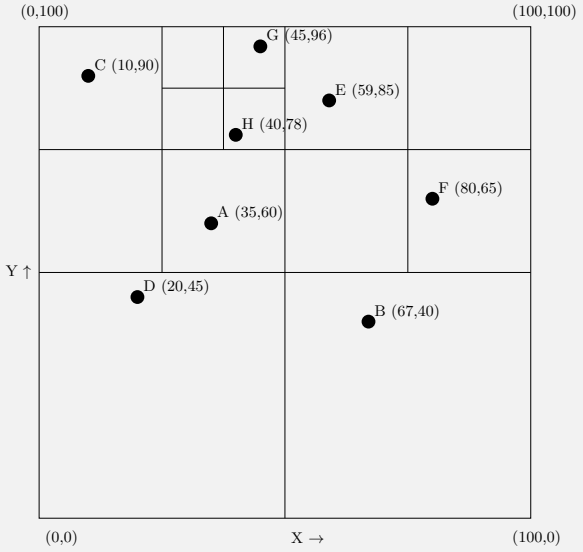
Required submission: 8 named (x,y) coordinates; Two insertion orders; Drawn PR quadtree and quadtree per insertion order.

Solution

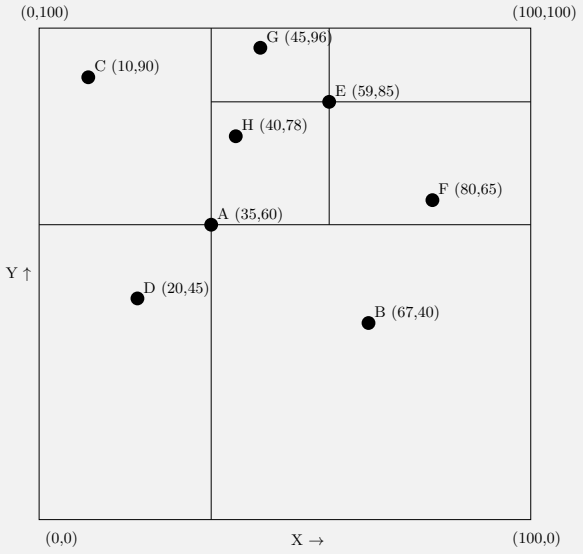
We chose the points

$$A(35, 60), B(67, 40), C(10, 90), D(20, 45)$$
$$E(59, 85), F(80, 65), G(45, 96), H(40, 78)$$

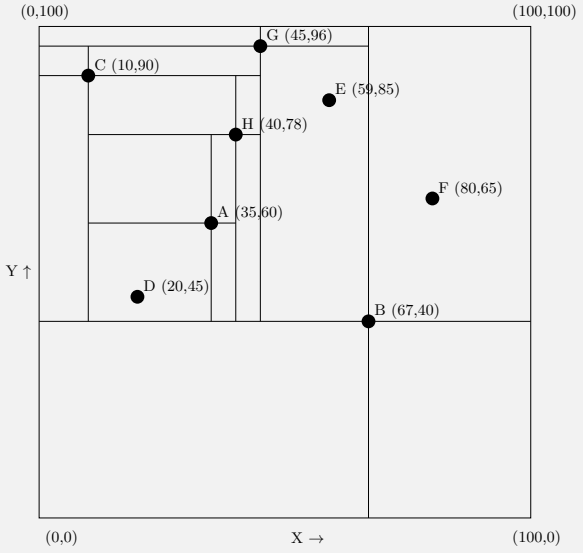
For the PR quadtree, the insertion order does not matter, it will always result in the same structure. As we can see, there are 5 empty leaves.



If we now choose the insertion order A, B, C, D, E, F, G, H we get the following quadtree with one empty leaf.



But if we use the insertion order B, G, C, H, A, D, E, F we get the following quadtree with 13 empty leaves.



b) Assuming we have the whole dataset present upon insertion, provide a pseudo code that optimizes the creation of the quadtree, such that the height of the resulting quadtree is minimized. Note that before the bulk-loading preprocessing of the data is allowed.

Required submission: Pseudo code; Pseudo code explanation.

Solution

The algorithm presorts the data primarily by the x coordinate and secondarily by the y coordinate. In each iterations it takes the median element of the ordered list as the point according to which a split is conducted. This version is optimized, since no sub-tree can possibly contain more than half the total number of nodes. More information is provided in "Quad Trees A Data Structure for Retrieval on Composite Keys".

```
presort data by x primarily and y secondarily

createQuadTree(data)
  if data.size == 0
    return
  if data.size == 1
    return create_leaf(data)
  split_point = select_median_point(data)
  node = create_node(split_point)
  regions = divide_data(data, split_point);
  for i in regions:
    node.addChild(i,createQuadTree(regions[i]))
  return node
```

c) Assuming we are given uniformly distributed point data, what is the probability that a node at depth k contains a particular point in a PR Quad tree? Additionally, for a collection of v points, calculate the probability that none of the points lies in a given cell at depth k?

Required submission: Formula to calculate probability;

Solution

The probability that a node at depth k contains a particular point is $1/4^k$. The probability that none of the points lies in a given cell at depth k is $(1 - 1/4^k)^v$.

Question 4: kd tree (1 P.)

a) Insert the following values, in provided order, into an empty kd tree, with $k = 3$:

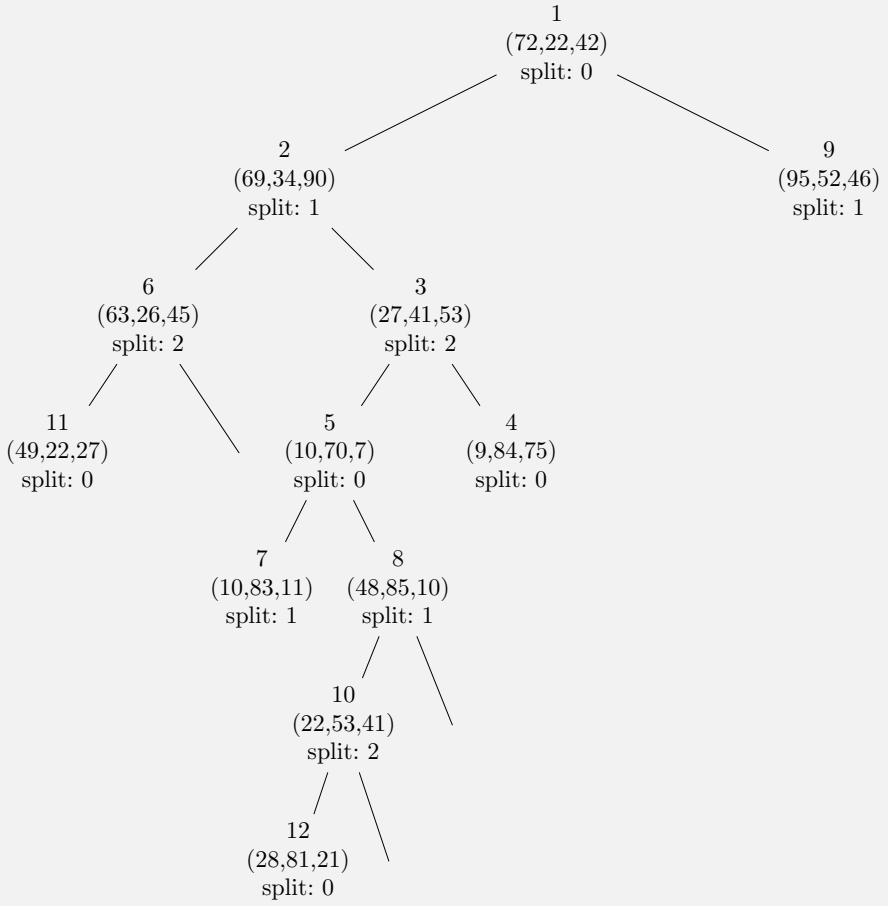
1. (72, 22, 42)
2. (69, 34, 90)
3. (27, 41, 53)
4. (9, 84, 75)
5. (10, 70, 7)
6. (63, 26, 45)
7. (10, 83, 11)
8. (48, 85, 10)
9. (95, 52, 46)
10. (22, 53, 41)
11. (49, 22, 27)
12. (28, 81, 21)

Draw the result as a tree, like shown in the lecture. Note which values you used for the split.

Required submission: Visualized kd tree; Split dimension per node.

Solution

The numbers of the nodes correspond to the tuple number above. "split: $i \in [0, k-1]$" signals, which coordinate is used to split. (e.g.: split 1 for tuple (A,B,C) means that tuples are split using coordinate B)



b) Adaptive kd tree

Provide an algorithm, in pseudo code, that takes a list of coordinates in k dimensions and builds a kd tree. This tree may have a maximum of ten data points in its leaves. The dimension used for the split should be the one with the largest variance. The value used for the split, should be the average of the split dimension of all relevant coordinates.

Required submission: Pseudo code; Pseudo code explanation.

Solution

The following algorithm takes a list of coordinates as parameter. It uses the helper function `split_dimension`, which returns the index of the dimension used for the split. The `avg` function, calculates the pivot value, by using the split dimension and the set of nodes.

```
function create_tree(nodes)
 if |nodes| ≤ 10
   return new Leaf(nodes)
 sd = split_dimension(nodes)
 pivot = avg(nodes, sd)
 smaller_part = nodes.select(λx : x[sd] < pivot)
 bigger_part = nodes.select(λx : x[sd] ≥ pivot)
 return new Inner(pivot, sd, create_tree(smaller_part),
                  create_tree(bigger_part))
```

```
function split_dimension(nodes)
 pivot_index = -1
 pivot_variance = -1
 for  i ∈ [0, k − 1]
   avg = AVG(nodes.map(λx : x[i]))
   variance = SUM(nodes.map(λx : (x[i] − avg)^2))/nodes.count()
   if  variance>pivot_variance
     pivot_variance = variance
     pivot_index = i
 return pivot
```