Philipps University of Marburg
Department of Mathematics and Computer Science
Database Research Group

Winter semester 2024/2025
16. 12. 2024

Prof. Dr. Bernhard Seeger
Amir El-Shaikh, M.Sc.
Marius Kuhrt, M.Sc.

Exercises for the lecture
Implementation of Database Systems

Submission: 17. 01. 2025,
no later than 8:00 p.m.
via the ILIAS platform

Exercise 5

Task 5.1: Persisted Bloom Filter (7+4+4) **(15 Points)**

A Bloom Filter is a space-efficient probabilistic data structure used to test whether an element is a member of a set. It supports quick look-ups and is particularly useful for applications where space is limited, and some false positives are acceptable, but false negatives are not. Internally, it uses a bit array and multiple hash functions to map elements. When querying, the filter may indicate that an element is *possibly present* (with a chance of false positives) or *definitely not present*. Due to their design, Bloom Filters are highly efficient for scenarios like caching, database indexing, and network routing. For more information, visit the Wikipedia page on Bloom Filters.

a) Implement the following class constructors and `close` method of `BloomFilter<E>`:

1. `BloomFilter(numBytes)` initializes the instance and allocates $8 \times$`numBytes` bytes for the bit vector. This means that each array position represents only a single bit of each byte.
2. `BloomFilter(DataInput)` reconstructs the `BloomFilter` from the given `DataInput`. The first 4 bytes are the serialized `numBytes` parameter, followed by the serialized bits. Note that each 8 bits from the bit vector are serialized into a single byte.
3. Implement the class method void `close(DataOutPut)` accordingly.

b) Complete the following class methods in `BloomFilter<E>`:

- `boolean containsMaybe(T)` checks wether an element was possibly inserted before.
- `void add(T)` adds an element to the bloom filter.
- `void reset()` resets the state of the bloom filter, i.e. as if newly created.

c) Create the class `BloomList<E>`, which extends a `LinkedList<E>` and uses a `BloomFilter<E>`. To support efficient operations, make sure the following methods are overridden/implemented:

- 1. `public boolean add(E)`, `public void add(int, E)` and `public boolean set(int, E);`
  2. `public boolean addAll(int, Collection<? extends E>);`
  3. `public boolean contains(Object);`
  4. `public void clear()` and `public void resetBloomFilter().`
- Test your implementation by inserting 10_000 random integers into a `BloomList` and a `LinkedList` and measuring the look-up time in both lists for random integers (which are probably not contained). How does the size of the bloom filter impact look-up times?

Task 5.2: External Linear Hashing (2+4+4+5) (15 Points)

Linear Hashing is a hashing strategy that de-amortizes table expansions by splitting at most one hash bucket at a time. We will implement a linear hash map that is backed by disk. It consists of two files, a primary file that holds the main hash table in consecutive blocks from the beginning, and a secondary file of overflow blocks and some metadata.

The `ExternalLinearHashmap` class contains the scaffolding to implement the hash map for arbitrary key-value pairs, provided they can be serialized and de-serialized. The nested `HashBucket` class represents a hash bucket, a `HashBlock` represents a block in the primary or secondary file. Each block consists of of a number of key-value pairs and (possibly) an ID of a block in the secondary file for the next block in the overflow list, where an ID of 0 indicates the end of the list. Be aware that changes to a `HashBlock` are only persisted on disk after calling `update` on the respective container.

Please refer to Chapter 8.5.2 of the lecture notes for the details of Linear Hashing.

a) First, implement `int realHashIndex(K key)` which calculates the actual bucket index for the given key, taking into account the current expansion pointer and level.

b) Next, implement `ProbeResult HashBucket.probe(K key)`. This method iterates over the primary and all overflow blocks to find a key-value pair with the given key. It must construct and return the `ProbeResult` as according to the documentation so that the caller can perform the necessary operation.

c) Implement `V HashBucket.insert(K key, V value)`. It must probe the bucket and, if the key is already contained, perform an update and return the previous value. Otherwise the pair is added to the bucket, which might include appending an overflow block.

d) Finally, implement `void performExpansion()` to perform an expansion step. This operation splits the bucket currently pointed at by the expansion pointer into two by rehashing all contained elements with the hash function of the next level. After a full expansion, the expansion pointer is reset to 0 and the level is incremented.

Hint: Examine `HashBucket.setElements` and `HashBucket.iterator` for information on how to deal with the blocks in a bucket.

Note: The following tasks are optional but *highly* encouraged.

Task 5.3: Space-Filling Curves **(0 Points)**

a) Draw the following space-filling curves in the provided grids

i lexicographic ordering

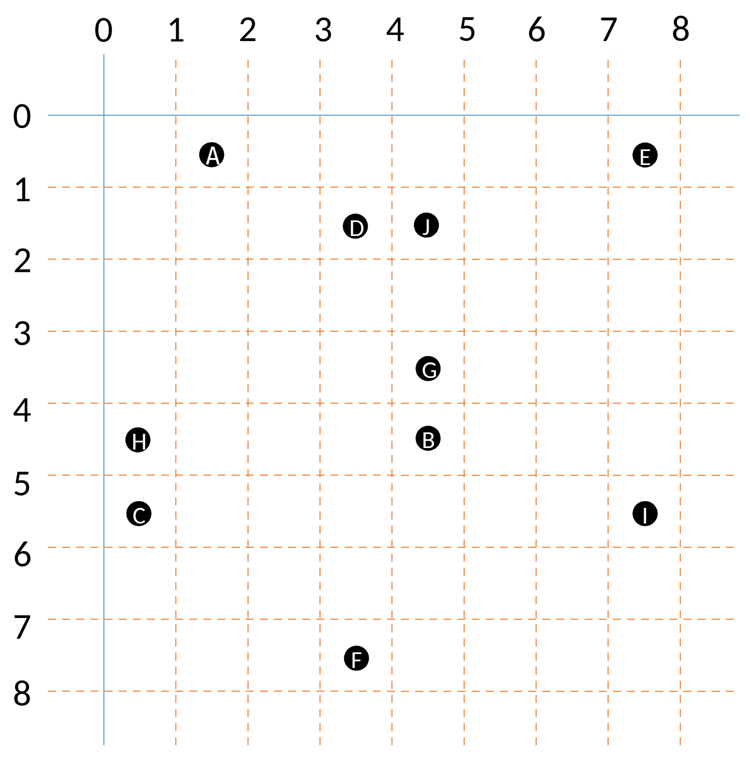
ii Z-order curve (starting at $(0, 0)$ and continuing to the right)

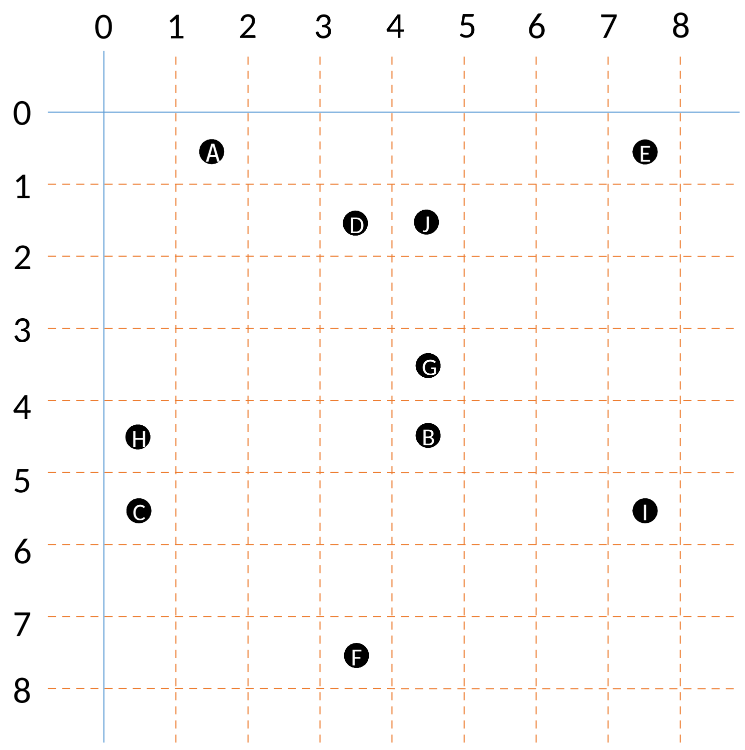
iii Hilbert curve (starting at $(0, 0)$ and continuing downwards)

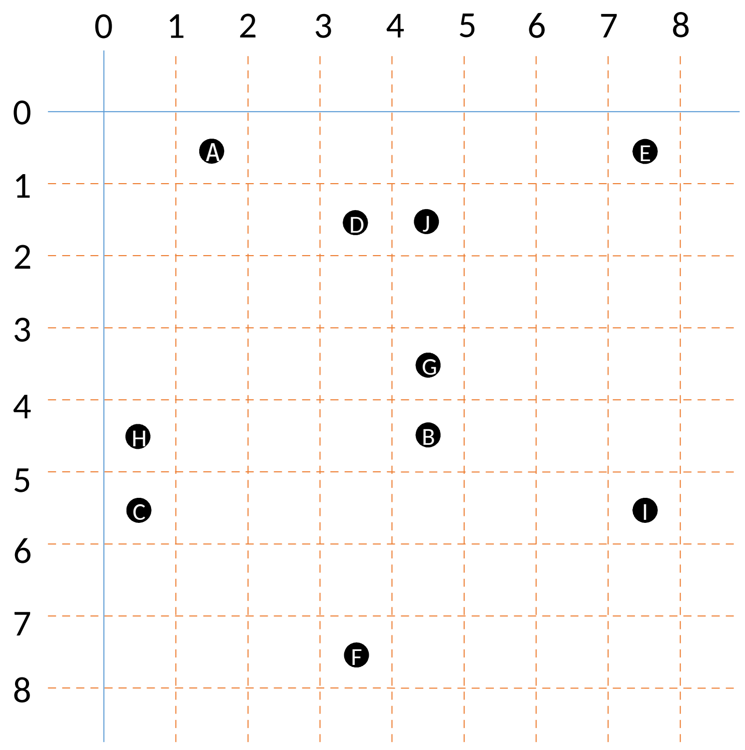
b) For each point, provide the nearest point according to the Manhattan metric. For each of these pairs of points, calculate the distance according to the three given orders.

c) Assume that the data is stored according to the ordering from a) in an array. Now give the distance in array positions for the point pairs from b).

d) What do you notice about task b) and c)?




0
1
2
3
4
5
6
7
8
A
E
D
J
G
H
B
C
I
F


0
1
2
3
4
5
6
7
8
A
E
D
J
G
H
B
C
I
F

Task 5.4: Construction of Spatial Index Structures (0 Points)

Given the two-dimensional coordinates from task 5.1. Create and draw the following indexes after each step. Insert the points alphabetically according to their label.

a) ZB$^+$ tree with $\langle 3, 2\rangle$ using the Z curve.

b) HB$^+$ tree with $\langle 3, 2\rangle$ using the Hilbert curve.

c) R* tree with $\langle 2, 4\rangle$. Store the points as the rectangle of the cell that contains them.

d) Run the following range query on the indexes: $[(3, 3), (4, 4)]$. What is the result and which nodes are visited in each case?