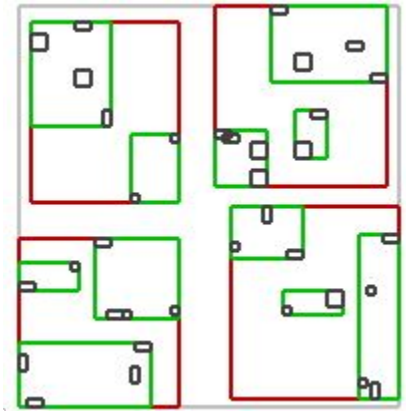
An efficient external R-tree for very large datasets

Bachelor’s Thesis

Noah Nock
31.03.2025

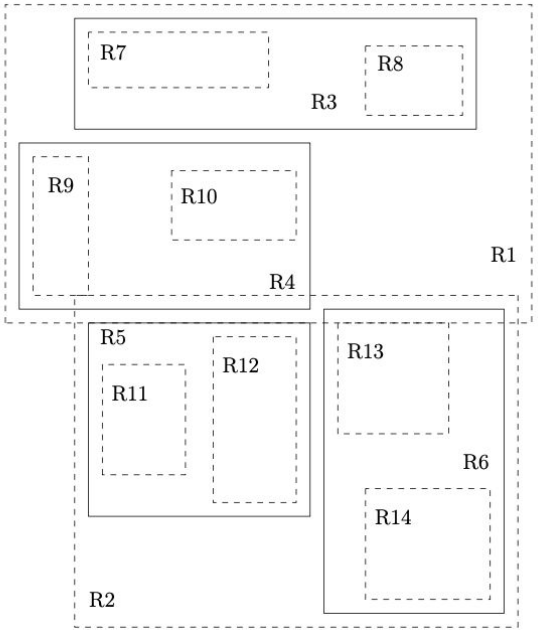
Spatial indexing and querying

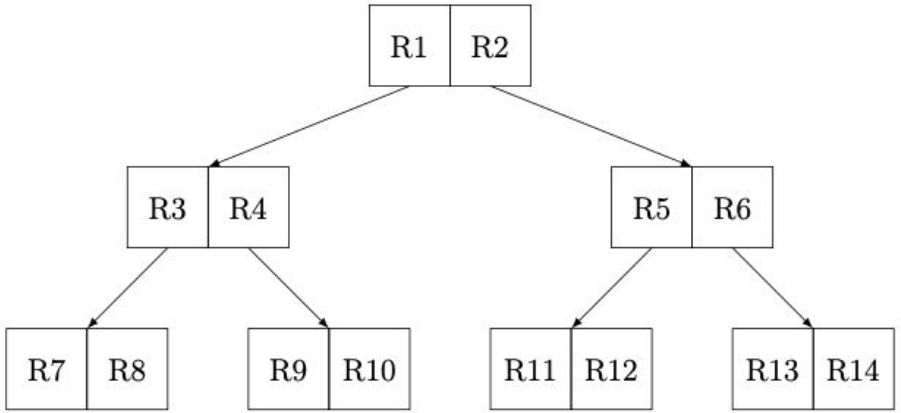
- Idea to include spatial indexing into SPARQL engine Qlever



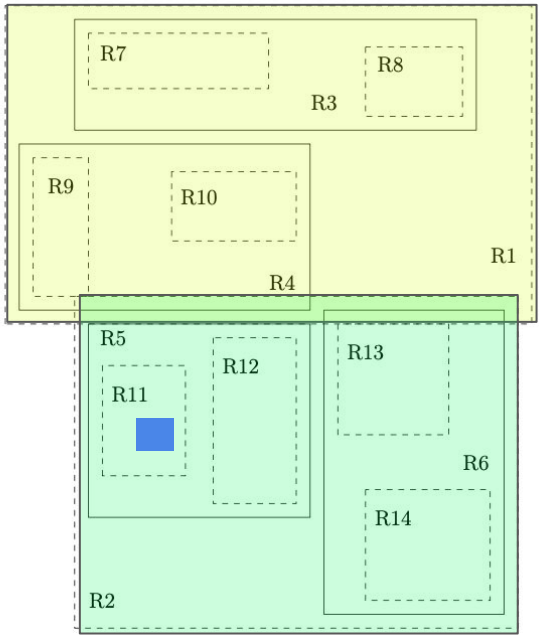
R-tree
(https://www.boost.org/doc/libs/1_84_0/libs/geometry/doc/html/img/index/rtree/rstar.png)

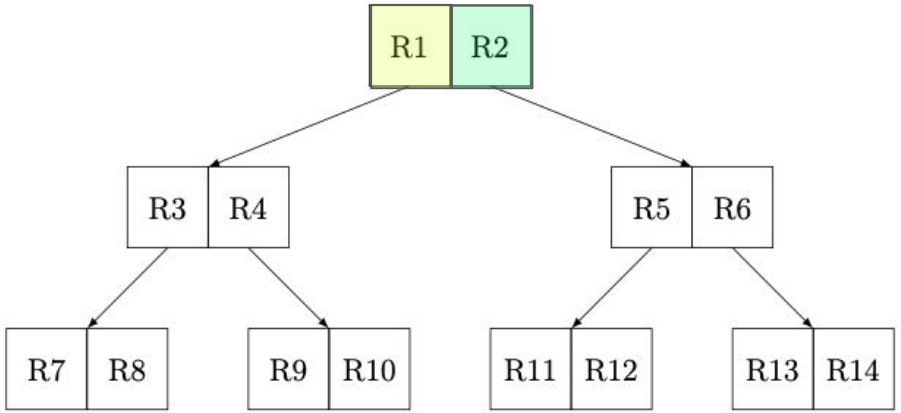
The R-tree data structure



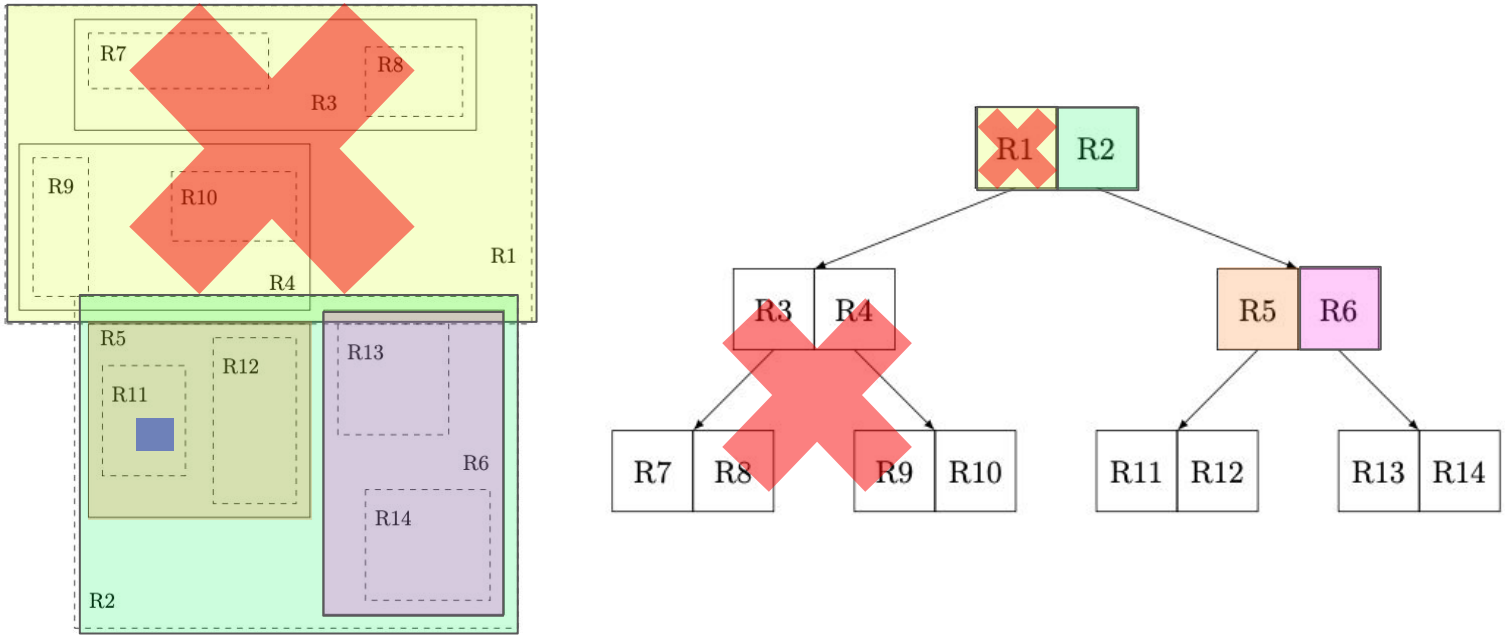


The R-tree data structure

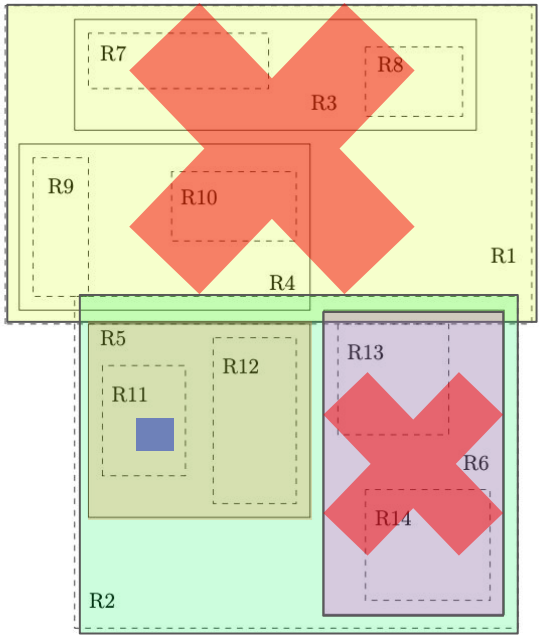


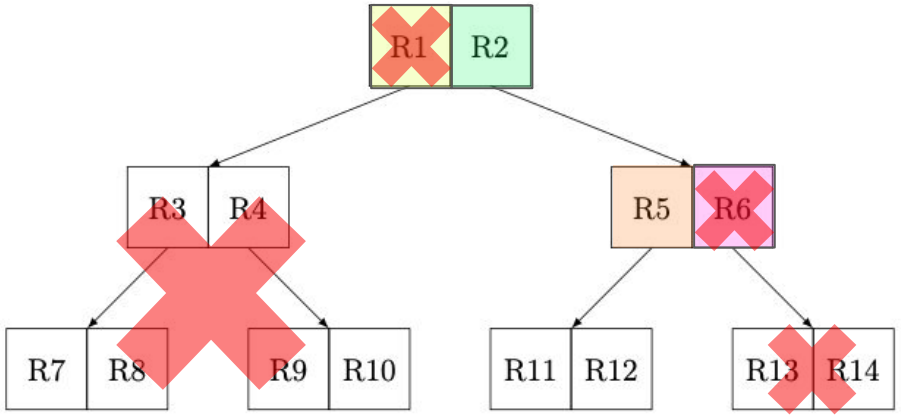


The R-tree data structure

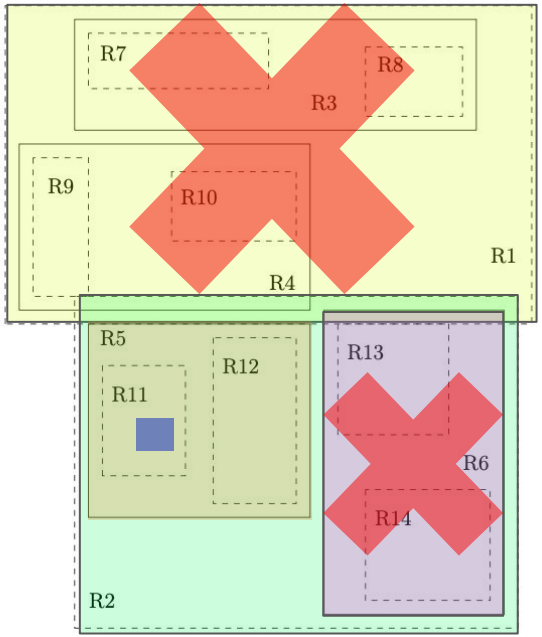


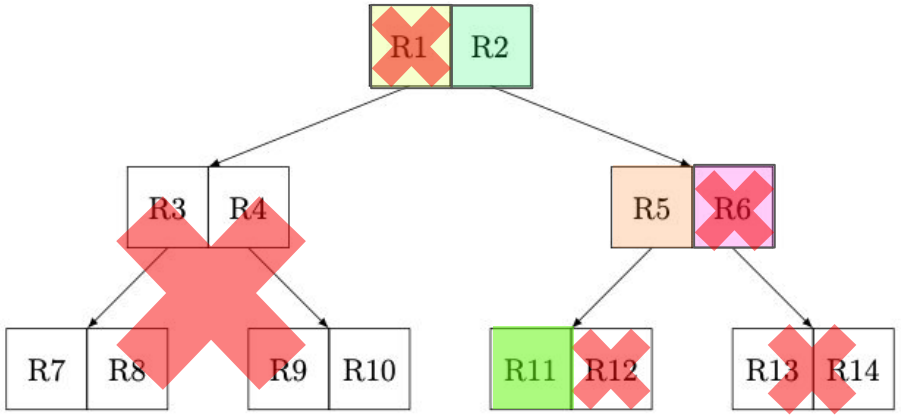
The R-tree data structure





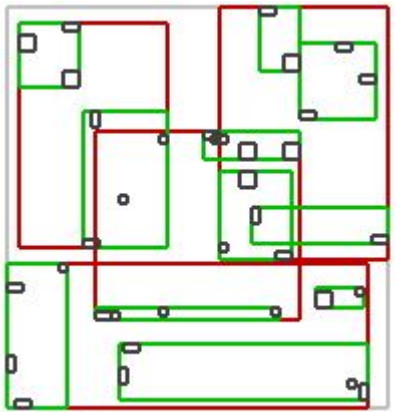
The R-tree data structure



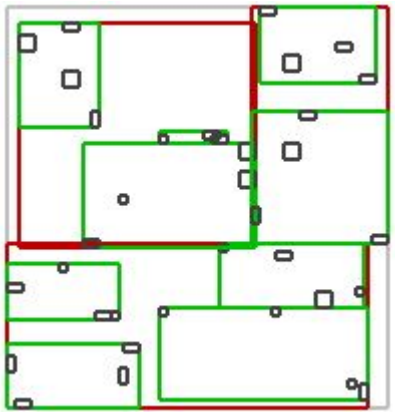


R-tree building

- Can be built one entry at the time -> possibly bad structure
- Know all entries beforehand -> Optimal solution can be found (Bulk loading)



Linear Algorithm
(https://www.boost.org/doc/libs/1_84_0/libs/geometry/doc/html/img/index/rtree/linear.png)



Packing Algorithm
(https://www.boost.org/doc/libs/1_84_0/libs/geometry/doc/html/img/index/rtree/bulk.png)

Problem with common R-tree implementations

- Bulk loading requires all entries to be known -> all entries get loaded in RAM
  e.g. in the C++ Boost library
- Lack of serialization
- Bad memory usage in building and searching

-> Most implementations do not scale well with very large data sets

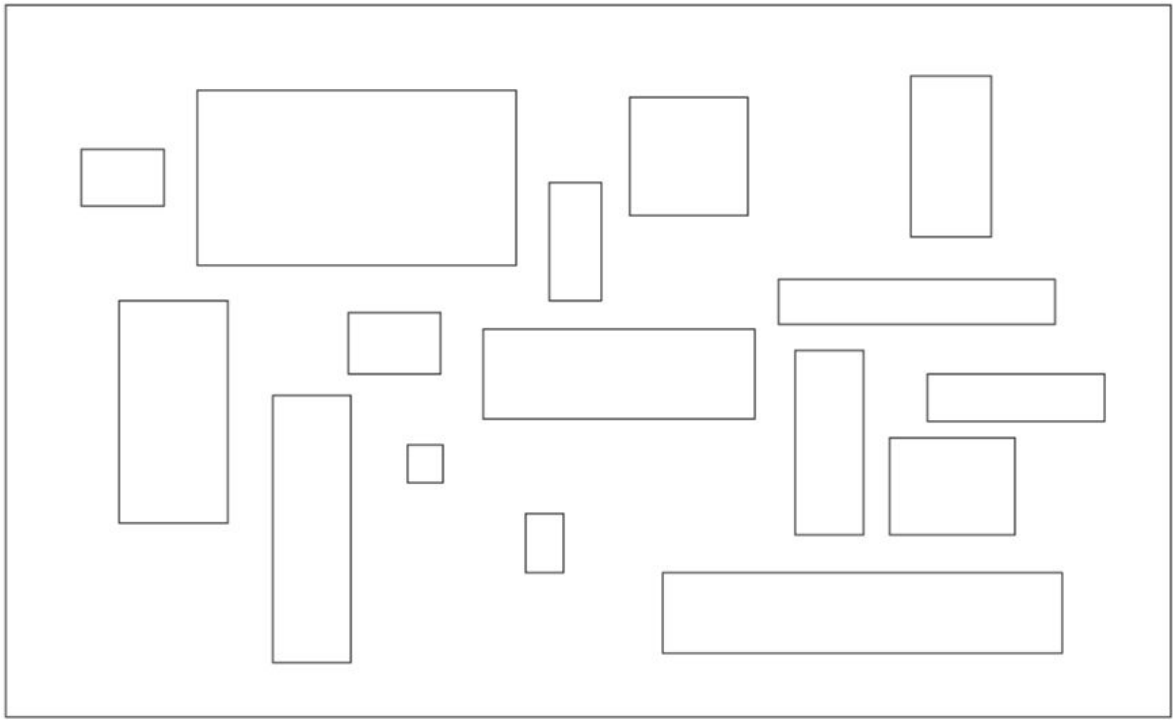
My goals

- Developing an efficient R-tree building algorithm and its implementation
  - building a R-tree in respect to a given memory limit
- Instant querying without loading in a whole R-tree
- Evaluate its performance

TGS Algorithm

- Top-down greedy splitting algorithm (1997)[1]
- Used to bulk load R-trees
- Splits the search space into two subspaces at the time

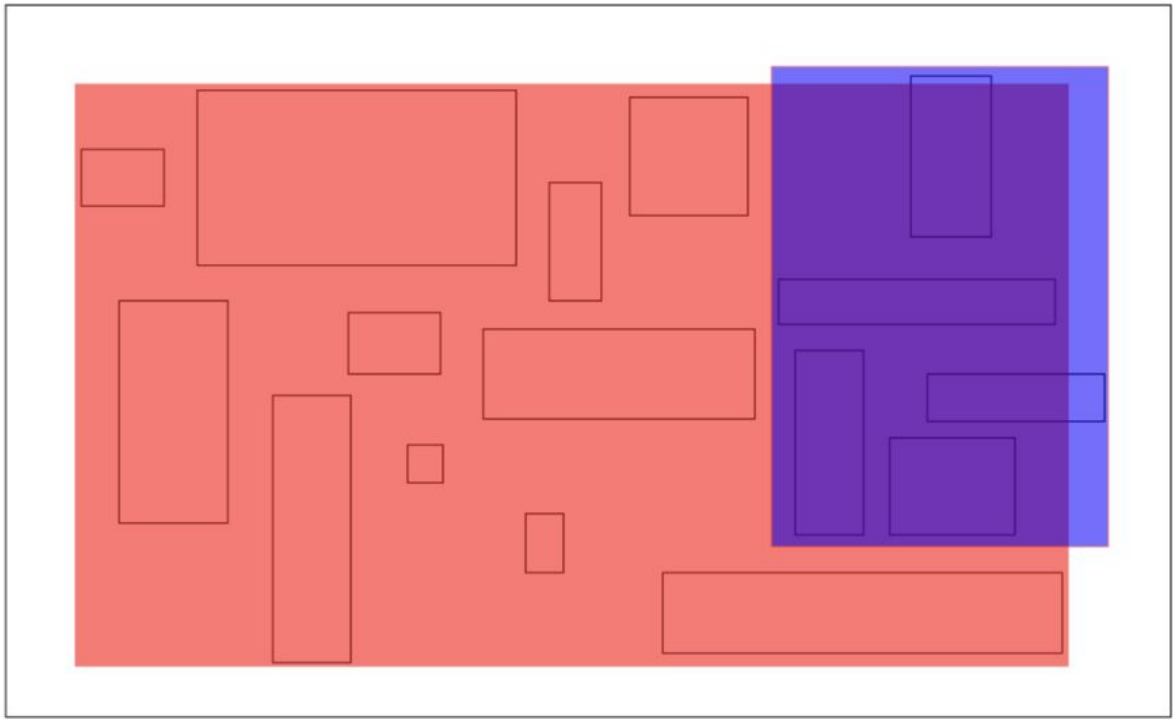
Finding the best split



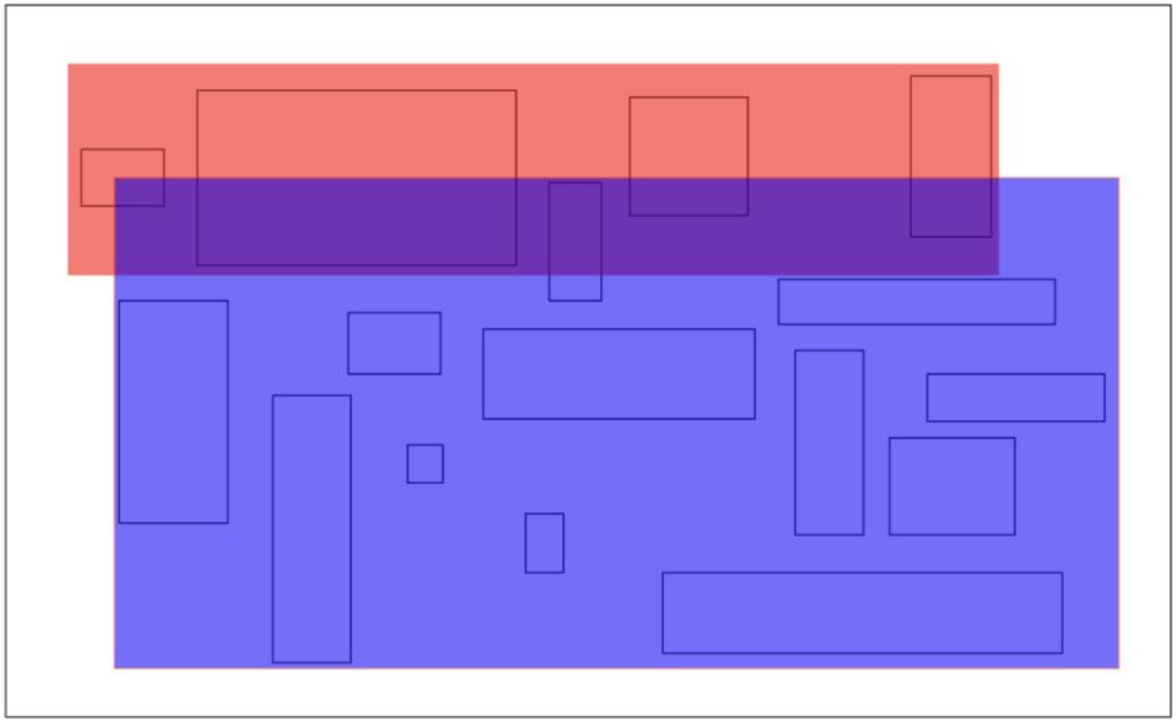
Finding the best split

Finding the best split

Finding the best split

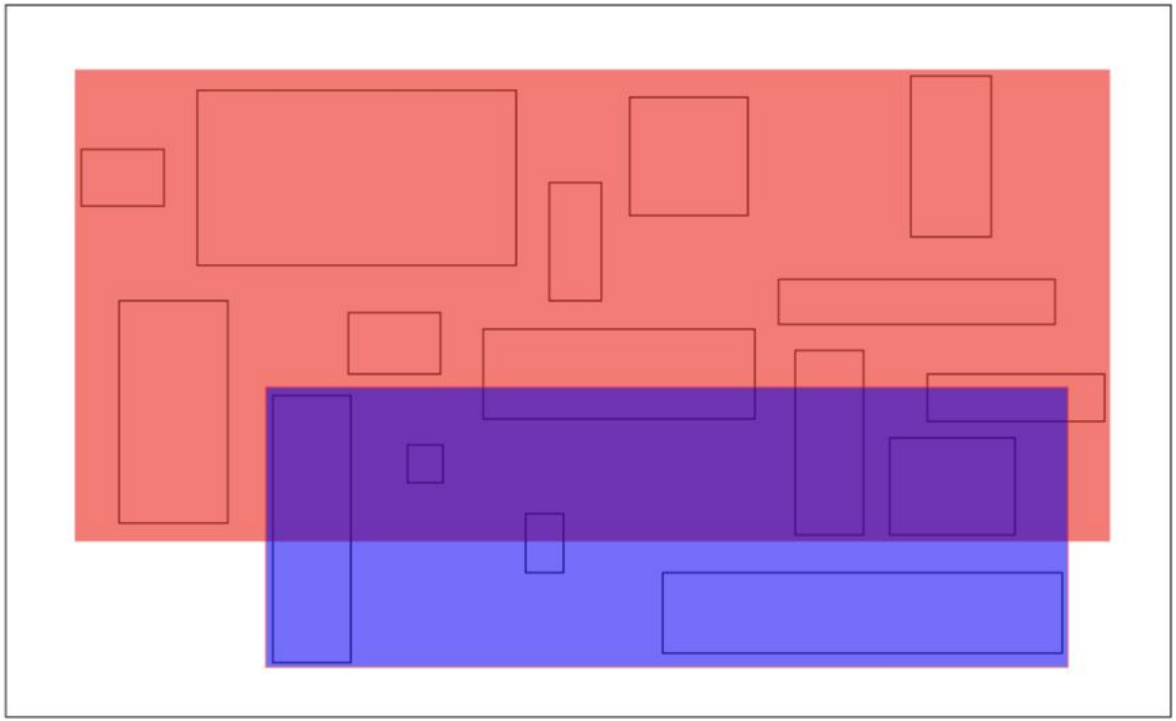


Finding the best split

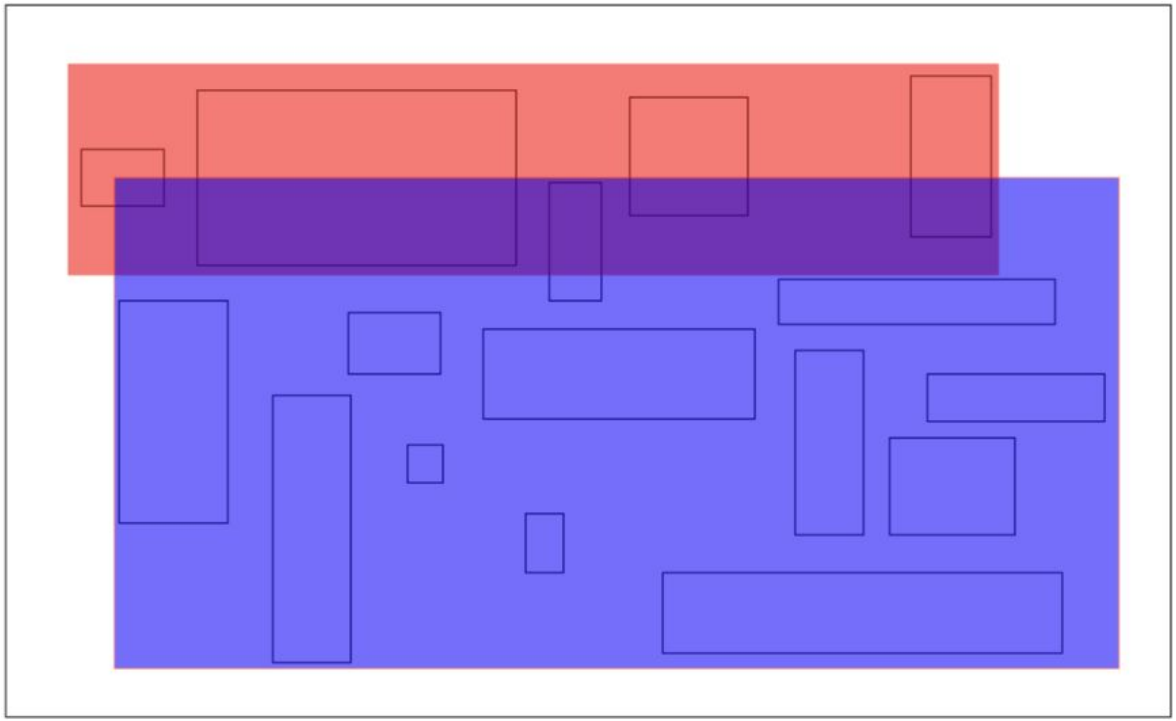


Finding the best split

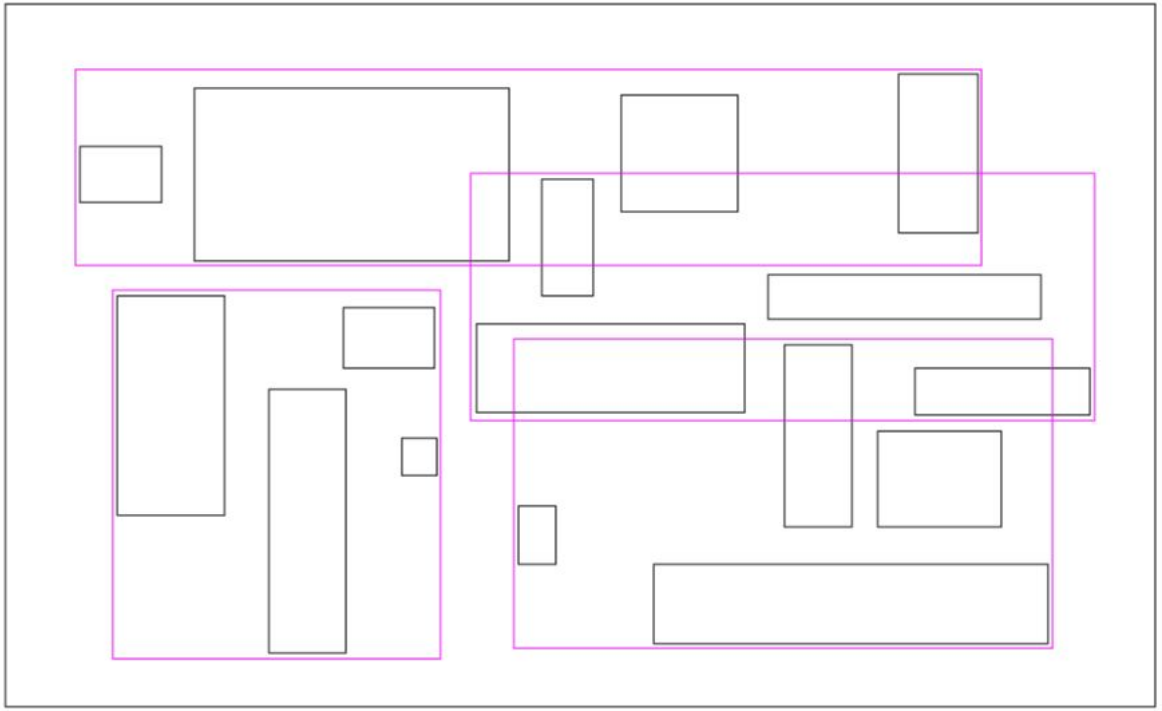
Finding the best split



Finding the best split



Finding the best split



Challenges and improvements of TGS

- Elements have to be sorted by both dimensions for every split
  - Fix the sorting after each split in linear time and O(1) memory consumption
- Not every element is needed for the split checks
  - Only elements at a multiple of *#elements_per_node* and the boundingboxes of the splits
- Building the tree in a depth first approach

Sorting

- How to keep both splits of a list sorted after splitting at a specific element?
  - trivial for the ordering in the same dimension as the split
  - non trivial for the other ordering
- Solution: each element keeps track of its position in both orderings

Sorting

centerX	centerY	orderX	orderY
2.3	4.6		
2.7	1.4		
4.3	3.8		
5.6	5.5		
5.9	2.3		

Sorting

centerX	centerY	orderX	orderY
2.3	4.6	1	
2.7	1.4	2	
4.3	3.8	3	
5.6	5.5	4	
5.9	2.3	5	

Sorting

centerX	centerY	orderX	orderY
2.7	1.4	2	
5.9	2.3	5	
4.3	3.8	3	
2.3	4.6	1	
5.6	5.5	4	

Sorting

centerX	centerY	orderX	orderY
2.7	1.4	2	1
5.9	2.3	5	2
4.3	3.8	3	3
2.3	4.6	1	4
5.6	5.5	4	5

Sorting

centerX	centerY	orderX	orderY
2.3	4.6	1	4
2.7	1.4	2	1
4.3	3.8	3	3
5.6	5.5	4	5
5.9	2.3	5	2

centerX	centerY	orderX	orderY
2.7	1.4	2	1
5.9	2.3	5	2
4.3	3.8	3	3
2.3	4.6	1	4
5.6	5.5	4	5

Sorting and Splitting

centerX	centerY	orderX	orderY
2.3	4.6	1	4
2.7	1.4	2	1
4.3	3.8	3	3
5.6	5.5	4	5
5.9	2.3	5	2

centerX	centerY	orderX	orderY
2.7	1.4	2	1
5.9	2.3	5	2
4.3	3.8	3	3
2.3	4.6	1	4
5.6	5.5	4	5

Sorting and Splitting

centerX	centerY	orderX	orderY
2.3	4.6	1	4
2.7	1.4	2	1
4.3	3.8	3	3
5.6	5.5	4	5
5.9	2.3	5	2

centerX	centerY	orderX	orderY
2.7	1.4	2	1
5.9	2.3	5	2
4.3	3.8	3	3
2.3	4.6	1	4
5.6	5.5	4	5

Sorting and Splitting

centerX	centerY	orderX	orderY
2.3	4.6	1	4
2.7	1.4	2	1
4.3	3.8	3	3
5.6	5.5	4	5
5.9	2.3	5	2

centerX	centerY	orderX	orderY
2.7	1.4	2	1
5.9	2.3	5	2
4.3	3.8	3	3
2.3	4.6	1	4
5.6	5.5	4	5

Sorting and Splitting

centerX	centerY	orderX	orderY
2.3	4.6	1	4
2.7	1.4	2	1
4.3	3.8	3	3
5.6	5.5	4	5
5.9	2.3	5	2

centerX	centerY	orderX	orderY
2.7	1.4	2	1
5.9	2.3	5	2
4.3	3.8	3	3
2.3	4.6	1	4
5.6	5.5	4	5

Sorting and Splitting

centerX	centerY	orderX	orderY
2.3	4.6	1	4
2.7	1.4	2	1
4.3	3.8	3	3
5.6	5.5	4	5
5.9	2.3	5	2

centerX	centerY	orderX	orderY
2.7	1.4	2	1
5.9	2.3	5	2
4.3	3.8	3	3
2.3	4.6	1	4
5.6	5.5	4	5

Sorting and Splitting

centerX	centerY	orderX	orderY
2.3	4.6	1	4
2.7	1.4	2	1
4.3	3.8	3	3
5.6	5.5	4	5
5.9	2.3	5	2

centerX	centerY	orderX	orderY
2.7	1.4	2	1
5.9	2.3	5	2
4.3	3.8	3	3
2.3	4.6	1	4
5.6	5.5	4	5

Sorting and Splitting

centerX	centerY	orderX	orderY
2.3	4.6	1	4
2.7	1.4	2	1
4.3	3.8	3	3
5.6	5.5	4	5
5.9	2.3	5	2

centerX	centerY	orderX	orderY
2.7	1.4	2	1
5.9	2.3	5	2
4.3	3.8	3	3
2.3	4.6	1	4
5.6	5.5	4	5

Sorting and Splitting

centerX	centerY	orderX	orderY
2.3	4.6	1	4
2.7	1.4	2	1
4.3	3.8	3	3
5.6	5.5	4	5
5.9	2.3	5	2

centerX	centerY	orderX	orderY
2.7	1.4	2	1
5.9	2.3	5	2
4.3	3.8	3	3
2.3	4.6	1	4
5.6	5.5	4	5

Precomputing all possible split locations

- For branching factor m and element size n in the current node, there are S elements per child with $S = \left\lceil \frac{n}{m} \right\rceil$
- Each possible split location is at a multiple of S
- While iterating through the lists to perform the splits -> record the current boundingbox at each Sth position
- Results are m possible splits per dimension -> used for TGS algorithm
  - Results are two lists *possibleSplitsX* and *possibleSplitsY*

-> very little information needed in memory to decide the splits

R-tree building - Start to finish

1. Create the initial x- and y-sorting
   a. Loop through both sortings and create *possibleSplitsX* and *possibleSplitsY*
2. Look for the best split based on *possibleSplitsX* and *possibleSplitsY* with TGS
3. Perform the splits while maintaining the sorting
   a. Keep track of the new *possibleSplitsX* and *possibleSplitsY* lists
   b. Recursively split both splits until m splits of equal size are found -> all m child nodes
4. Choose one child node and repeat 2-4 until leaf nodes are reached
   a. Save each finished node on disk -> serialization of the R-tree
5. Repeat 4 until all nodes are finished

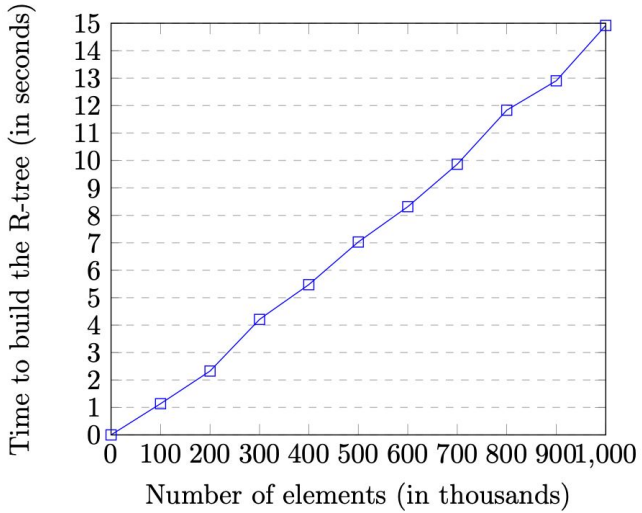
Internal and External data

- While splitting: algorithm processes one element at the time
  - does not matter if elements are stored on disk or loaded in RAM
- Internal data: elements are loaded in RAM -> faster access
- External data: elements are stored on disk and loaded one at the time -> suitable for low memory applications
- Dynamic combination of internal and external data
  - Split the elements externally until remaining elements fit in memory -> switch to internal

Runtime complexity

- Initial sorting runs in $O(n \cdot log(n)))$
- Deciding a split only relies on *possibleSplitsX* and *possibleSplitsY* -> $O(1)$
- Performing the split and preparing for the next splits at each tree level -> $O(n)$

-> $O(n \cdot log(n)))$



Space complexity

- Initial sorting: highly depends on sorting algorithm and implementation
  - can be done externally with a memory limit e.g. stxxl[2]
- Finding and performing a split -> $O(1)$
- Keeping track of the depth-first expansion of the R-tree -> $O(log(n))$

  -> $O(log(n))$

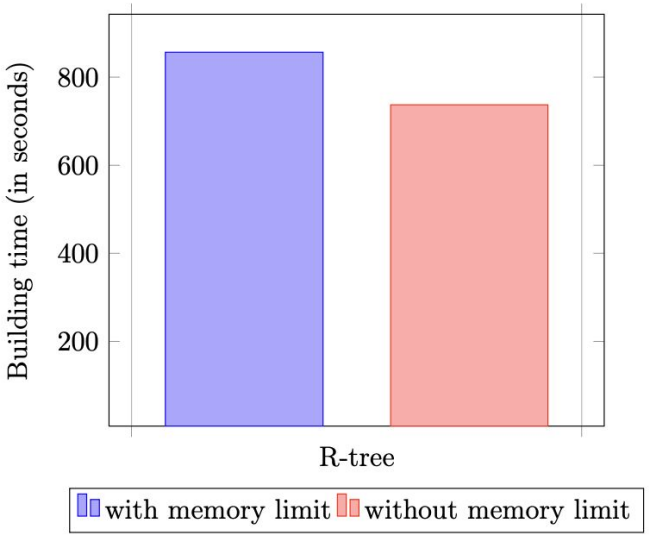
Runtime Analysis - Data samples

Use of three different datasets:

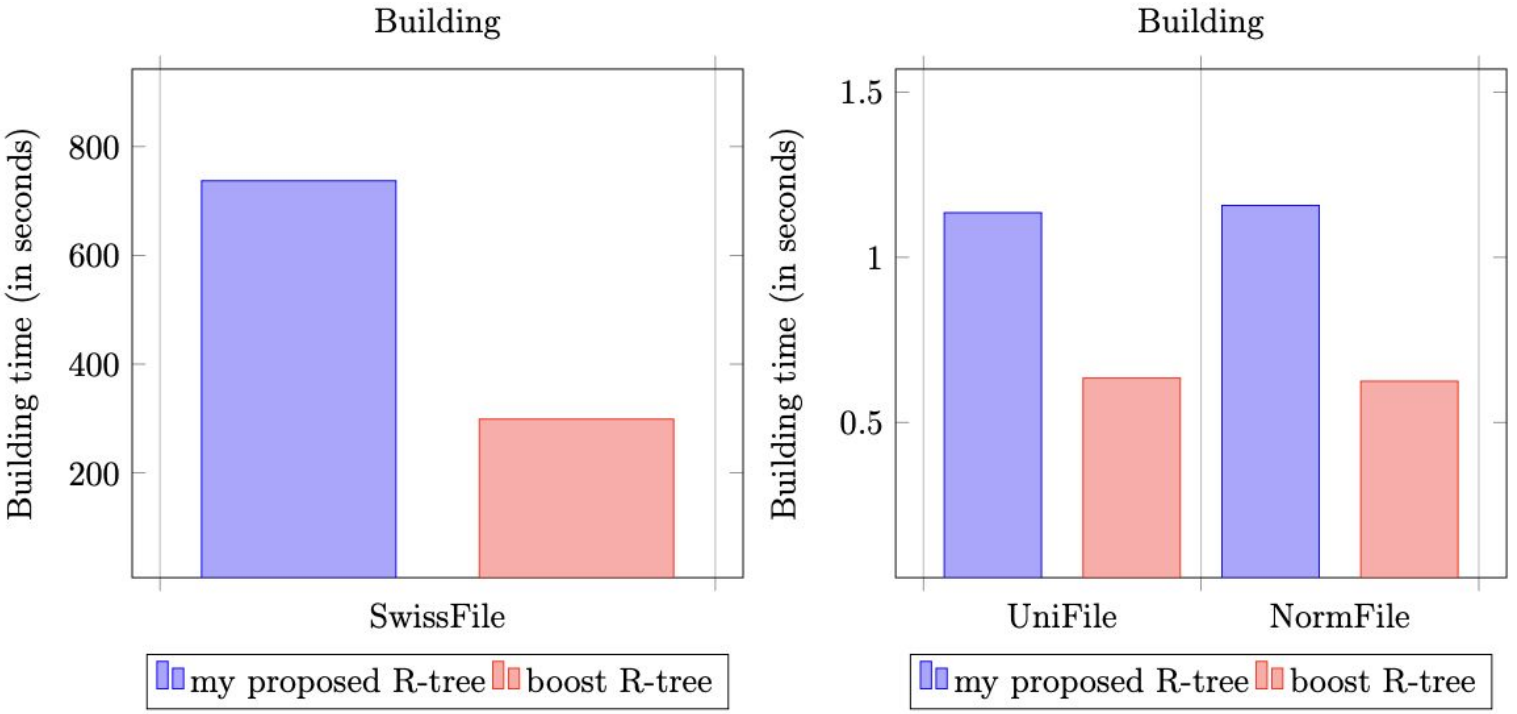
1. *SwissFile*: Real data of the boundingboxes of all objects in switzerland recorded by OpenStreetMap[3]. Consists of 33266131 entries.
2. *NormFile*: Synthetic data of 1,000,000 randomly generated boundingboxes over the area of switzerland. The generation is normally distributed.
3. *UniFile*: Synthetic data of 1,000,000 randomly generated boundingboxes over the area of switzerland. The generation is uniformly distributed.

Runtime Analysis - Internal vs. External Building

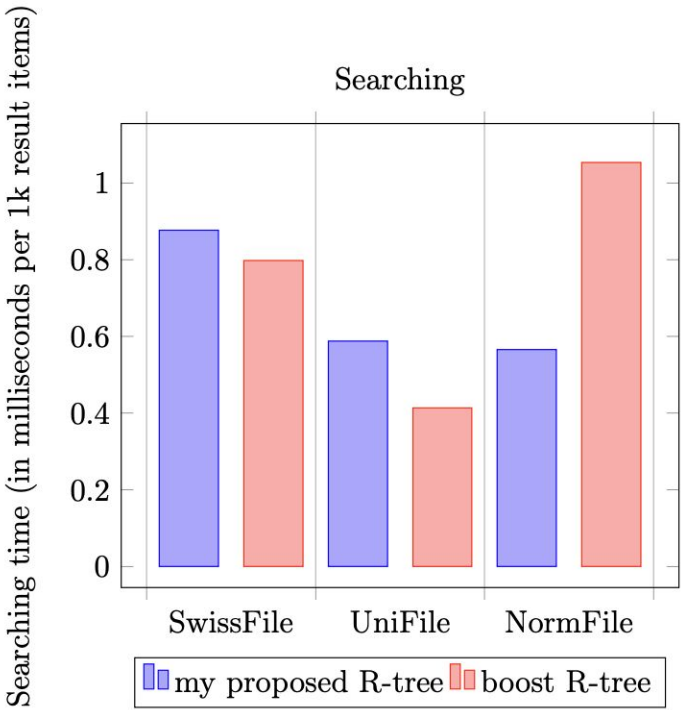
- Comparison between the building time of a R-tree based on *SwissFile* for:
  - No memory limit: The Algorithm used up to 5.32GB of RAM
  - Memory limit of 500MB



Runtime Analysis - Comparison to Boost



Runtime Analysis - Comparison to Boost



Conclusion

- Proposed an efficient and external R-tree building algorithm
- Able to respect given memory limit while building a R-tree
- Scales well with large data sets
- Increase in building time compared to Boost
- Maintaining reasonable querying time in comparison to the Boost R-tree

-> Usable for computing spatial indices on very large datasets

Sources

- Y. J. García R, M. A. López, and S. T. Leutenegger, “A greedy algorithm for bulk loading r-trees,” pp. 163–164, 1998.[1]
- B. Gehrels, B. Lalande, M. Loskot, A. Wulkiewicz, and L. Simonson, “Boost r-tree algorithm.” https://www.boost.org/doc/libs/1_87_0/libs/geometry/doc/html/geometry/spatial_indexes/introduction.html (Accessed: 25.03.2025).
- B. Gehrels, B. Lalande, M. Loskot, and A. Wulkiewicz, “Boost r-tree documentation.” https://beta.boost.org/doc/libs/1_82_0/libs/geometry/doc/html/geometry/reference/spatial_indexes/boost__geometry__index__rtree.html (Accessed: 25.03.2025).
- Stxxl documentation[2]. https://stxxl.org/tags/1.4.1/index.html (Accessed: 26.03.2025).
- OpenStreetMap[3]. https://www.openstreetmap.org/ (Accessed: 26.03.2025).