

Query and Update Efficient B⁺-Tree Based Indexing of Moving Objects

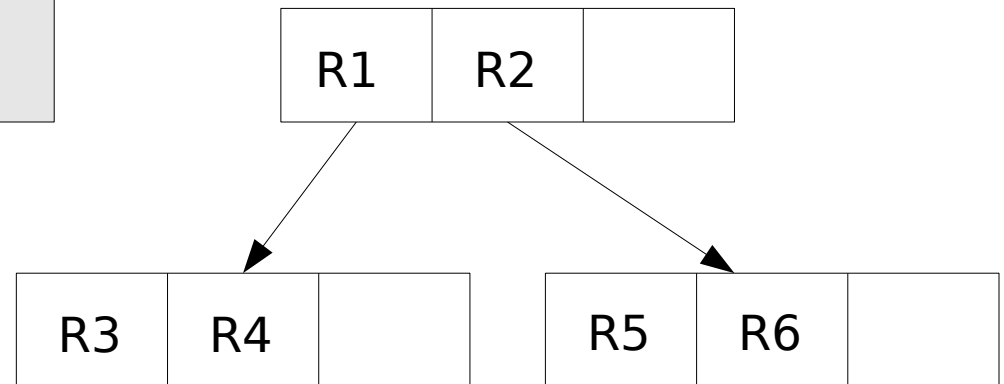
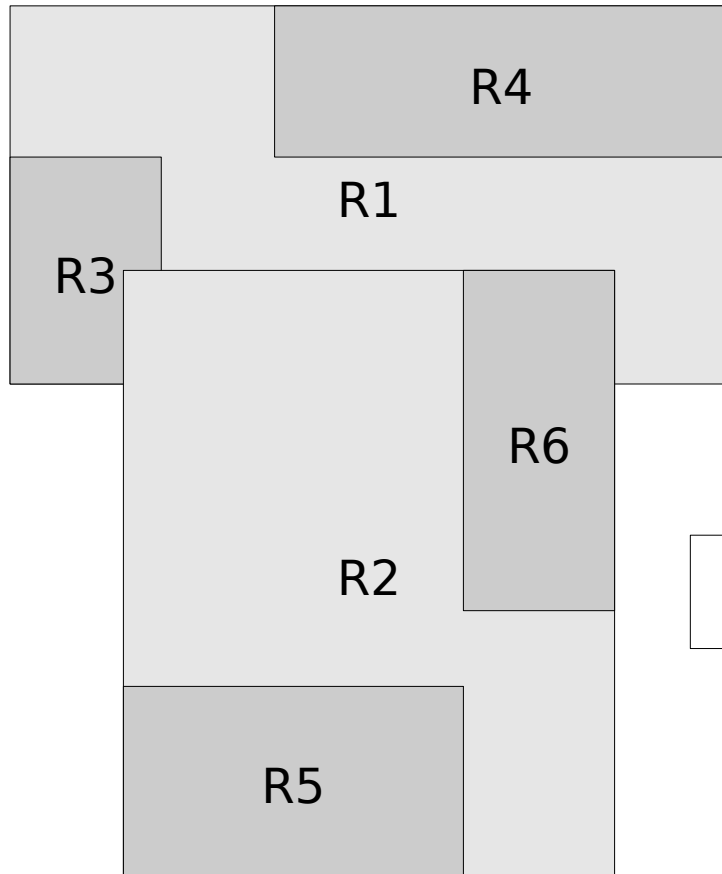
Overview

- Introduction
- Related Work
- B^x-tree
 - Structure
 - Queries
 - Updates
- Performance
- Conclusion
- Questions

Introduction

- Moving objects (users of wireless devices)
 - Frequent updates, change of dataset
- B⁺-tree
 - Efficient for insertion and retrieval
 - High fanout, records in leaf nodes, pointers for fast in-order traversal
- Linearize data
 - Space-filling curves
- B^x-tree (x denotes the space dimension)
 - Profits from advantages of B⁺-tree
 - Index for multidimensional data

Related Work - Time-parametrized R-tree



TPR-tree

Based on R-tree, extends R*-tree

Structure – B^x-tree – Object representation

- Objects are linear functions: $O = ((x, y), (vx, vy), t)$
 - x/y: coordinates, vx/vy: velocities, t: time of last update
- Time information
 - Time is partitioned into phases
 - Object is part of a partition according to last update time
 - Partition has label time
 - Binary representation of number of partition
- Location information
 - Space-filling curve is applied to every partition
 - Binary representation of value on space-filling curve
- Index key: time information + location information

Structure - B^x-tree

Index key: **time information** + location information

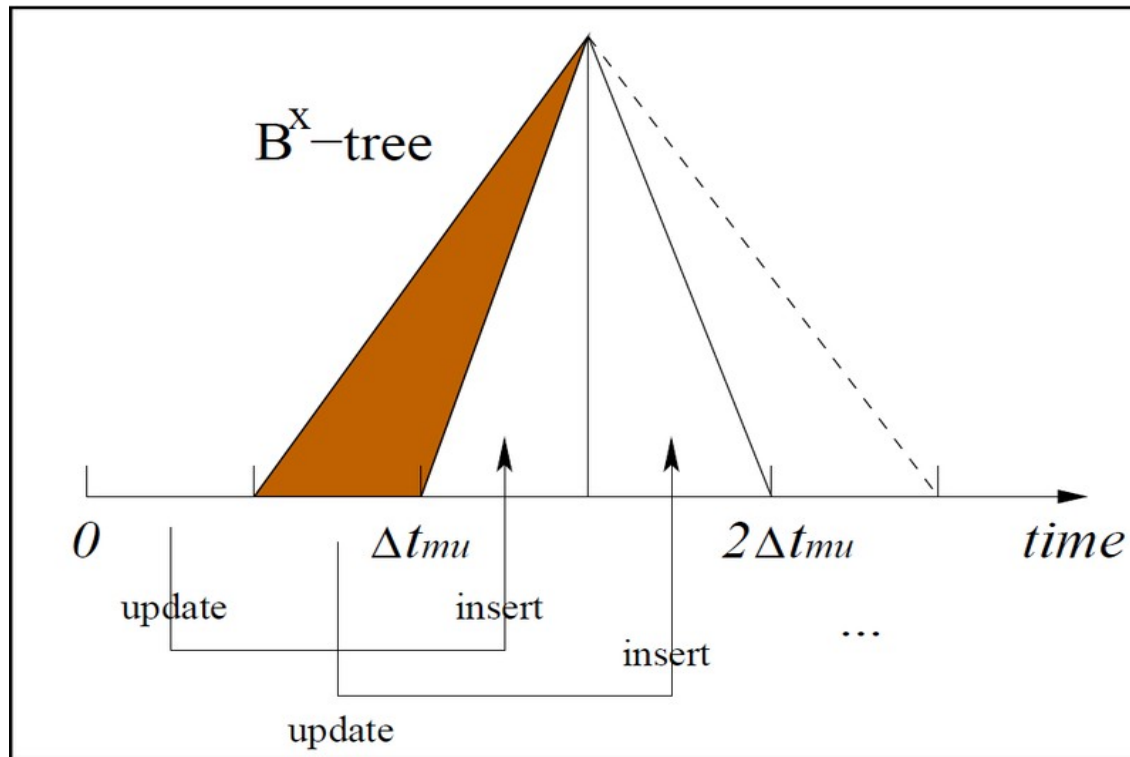
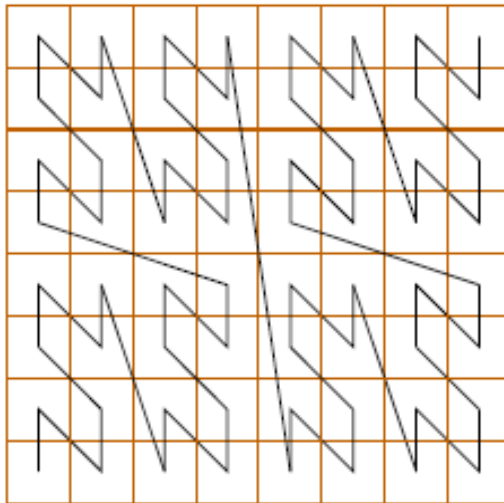


Figure: B^x-tree. Wikipedia

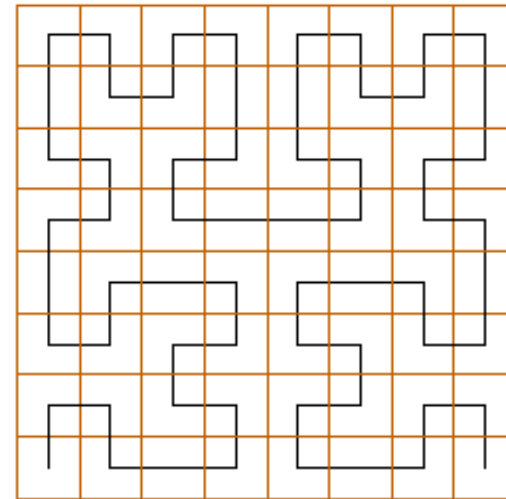
Structure - Space-filling curves

Index key: time information + **location information**

Preserve proximity: points close in space are close on curve



Peano curve (Z-curve)



Hilbert curve (H-curve)

Figure: „Query and Update Efficient B⁺ tree based Indexing of Moving Objects“
by Christian S. Jensen, Dan Lin, and Beng Chin Ooi. 2004

Structure - B^x-tree - Example (1)

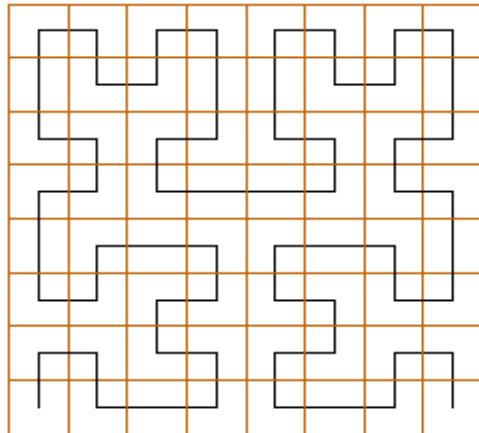
- $\Delta t_{mu} = 60$
- $n = 2$
- $O_1 = ((5,6),(0, -0.1), 0)$
- $O_2 = ((3,2),(0.1, -0.025), 20)$

- $t_{lab} = (t_u + \Delta t_{mu} / n)$ rounded to next future label timestamp
- $index_partition = (t_{lab} / (\Delta t_{mu} / n) - 1) \bmod (n + 1)$

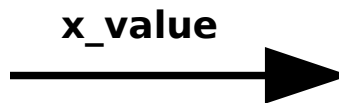
- $x' = x + vx * (t_{lab} - t_u)$
- $y' = y + vy * (t_{lab} - t_u)$

Structure - B^x-tree - Example (2)

- $\Delta t_{\text{nu}} = 60$
- $n = 2$
- $O_1 = ((5,6),(0, -0.1), 0)$
- $O_2 = ((3,2),(0.1, -0.025), 20)$



Hilbert curve (H-curve)



21	22	25	26	37	38	41	42
20	23	24	27	36	39	40	43
19	18	29	28	35	34	45	44
16	17	30	31	32	33	46	47
15	12	11	10	53	52	51	48
14	13	8	9	54	55	50	49
1	2	7	6	57	56	61	62
0	3	4	5	58	59	60	63

(0,0)

Structure - B^x-tree - Example (3)

- $\Delta t_{\text{nu}} = 60$
- $n = 2$
- $O_1 = ((5,6),(0, -0.1), 0)$
- $O_2 = ((3,2),(0.1, -0.025), 20)$
- $\text{index_key} = (\text{index_partition})_2 + (\text{x_value})_2$
+: concatenation

Updates - Insertion, Deletion, Updates

- Insertion (like in the B⁺-tree)
 - Compute index key and then insert
- Deletion (like in the B⁺-tree)
 - Index keys for latest insertion of every object stored in additional data structure
 - Search for key
- Update
 - When parameters of moving function change
 - Calculate new index key, insert into new partition

Queries – Range Query

- Return all objects in a rectangular range at a specific time (current or in the future)
- Query-window enlargement
 - Query time can be earlier or later than timestamp of phase
 - Enlarged query-window encloses all objects that can possibly be in query-window at query time
 - Naive: take maximum speed of all objects to calculate enlargement
- Traverse partitions to find all objects in enlarged query-window
- Check result set for correctness

Queries - Range Query - Example

- Rectangle: query region
- Shaded: enlarged query-window
- Objects: o1, o2, o3, o4
- Filled circles: objects at reference time of phase
- Empty circles: objects at query time
- Arrows: velocity vectors

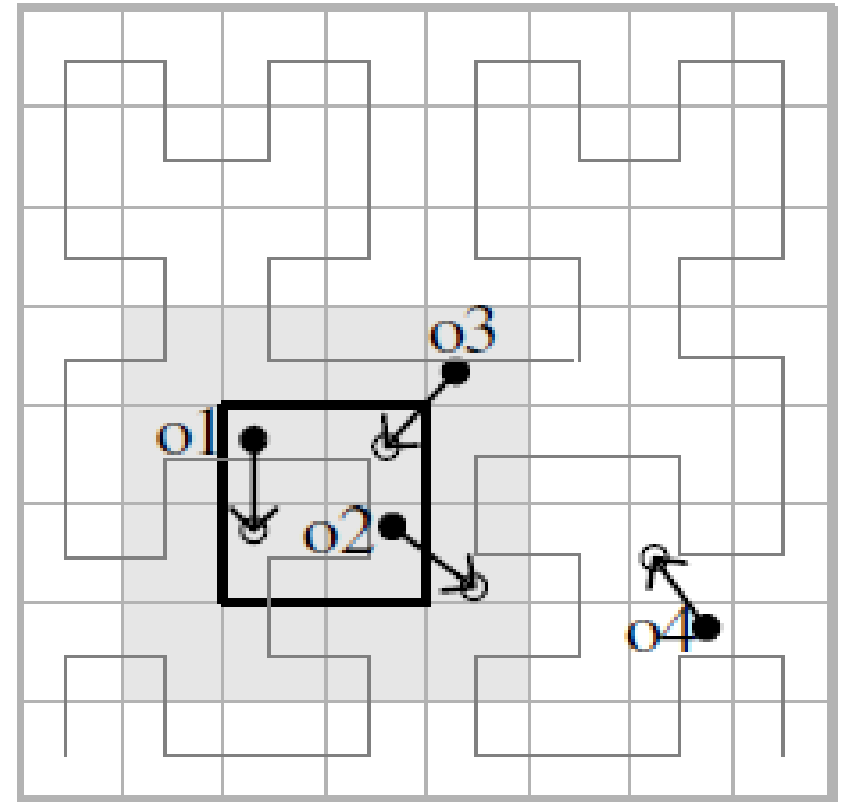


Figure: „Robust B⁺-Tree-Based Indexing of Moving Objects, in Proceedings of Seventh International Conference on Mobile Data Management“ by Jensen, C.S., D. Tiesyte, N. Tradisauskas. 2006

Queries – k NN Query, Continuous Queries

- k NN Query (k nearest neighbors)
 - Set of range queries
 - Initial range is estimated taking k and the number of objects into account
 - Terminate when k objects in result set otherwise enlarge rectangle with initially calculated range
- Continuous Queries
 - Query time: *now + offset*
 - Recompute result set at intervals, maintain in between
 - Range Query: insert if in range, delete if in range
 - k NN Query: insert if closer compared to result set, if deleted object in result set recompute

Performance - Range Query

- Performance
 - Many separate sections on space-filling curve require many 'jumps' (go back one level in tree and follow right sibling pointer)
 - Possibly all nodes retrieved instead of only the ones in range
- Performance tests: B^x-tree compared to TPR-tree
 - Varying dataset sizes
 - MBRs have bad performance
 - Higher probability of overlapping for large data set
 - More nodes to traverse

Performance - Updates

- Update is deletion followed by insertion
 - Index key for deletion can be looked up in additional data structure
 - Calculate key for insertion before affecting the tree
 - Same performance for updates as insertion in B^+ -tree
- Performance tests: B^x -tree compared to TPR-tree
 - Dataset size: Same effect as for queries

Conclusion

- B^x-tree
 - Not a multidimensional data structure
 - Efficient queries and updates
 - Easy to integrate
- Performance
 - Good results for range and *k*NN queries
 - Outperforms TPR-tree, no expensive enlargement of MBR

Conclusion - Outlook

- Parameters based on estimations
 - Δt_{mu} , initial range for k NN query
- Problems
 - Query regions with dense or sparse data (grid has fixed sized cells)
 - Query-window larger than needed
- B_r^x -tree
 - Optimize size of query-window enlargement
 - Only add relevant cells to enlarged query-window
- ST²B-tree (Self-tunable spatio-temporal B⁺-tree)
 - Based on B⁺-tree
 - Good adaption to sparse/dense data for querying

Questions?