Handling Frequent Updates of Moving Objects

CIKM'05 paper by

Bin Lin and Jianwen Su
Dept. of Computer Science, University of California, Santa Barbara
Santa Barbara, CA, USA

Presented by
Laurynas Biveinis

Talk Outline

- Motivation
- Example: updates with TPR-tree
- Proposal: Lazy Group Update
- Updates with LGU
- Performance results
- Strong and weak points
- Relation to my project
- Conclusion

Motivation

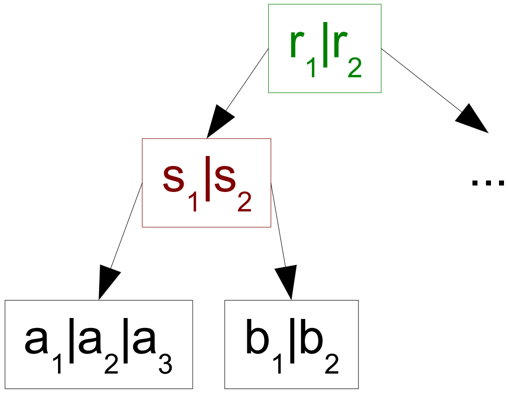
- A realistic scenario:
  - A city with 4 mln population
  - 1 mln cell phone users are using location-based services
  - Every LBS subscriber reports his/her position once per hour
  - Result: 280 updates per second!
- High update rate is a big challenge for any disk-based index structure

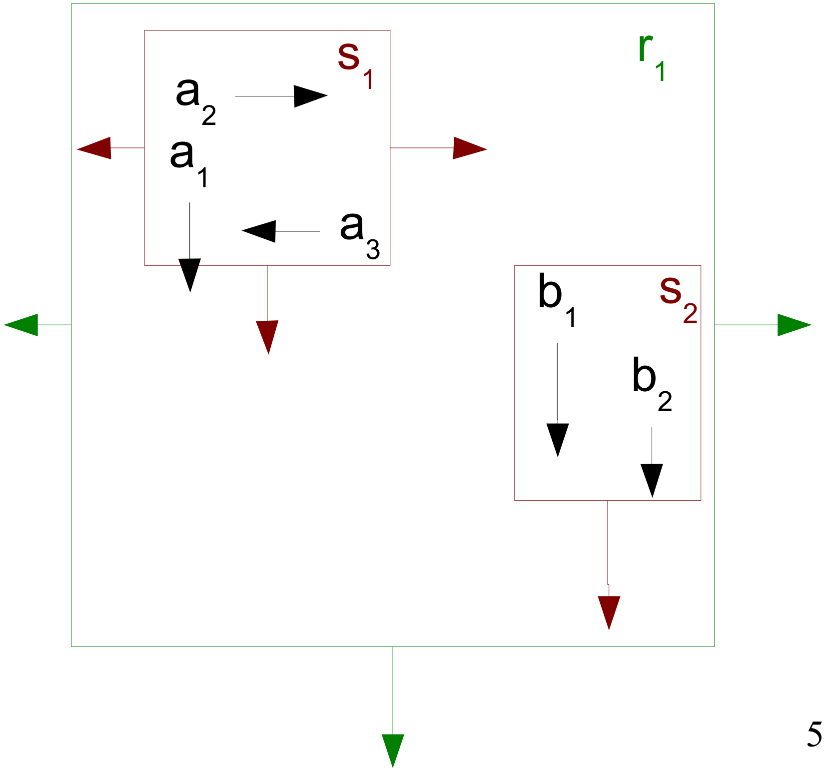
Talk Outline

- Motivation
- Example: updates with TPR-tree
- Proposal: Lazy Group Update
- Updates with LGU
- Performance results
- Strong and weak points
- Relation to my project
- Conclusion

Example: TPR-tree

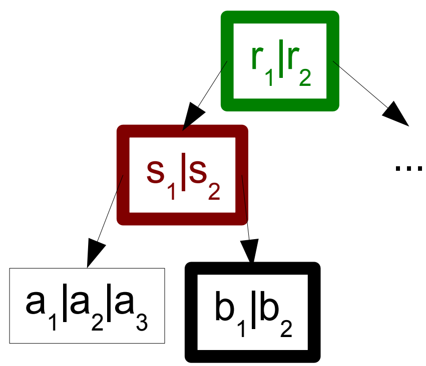
- A natural choice for indexing moving objects
  - Updates required only on velocity changes
  - But are they effective?

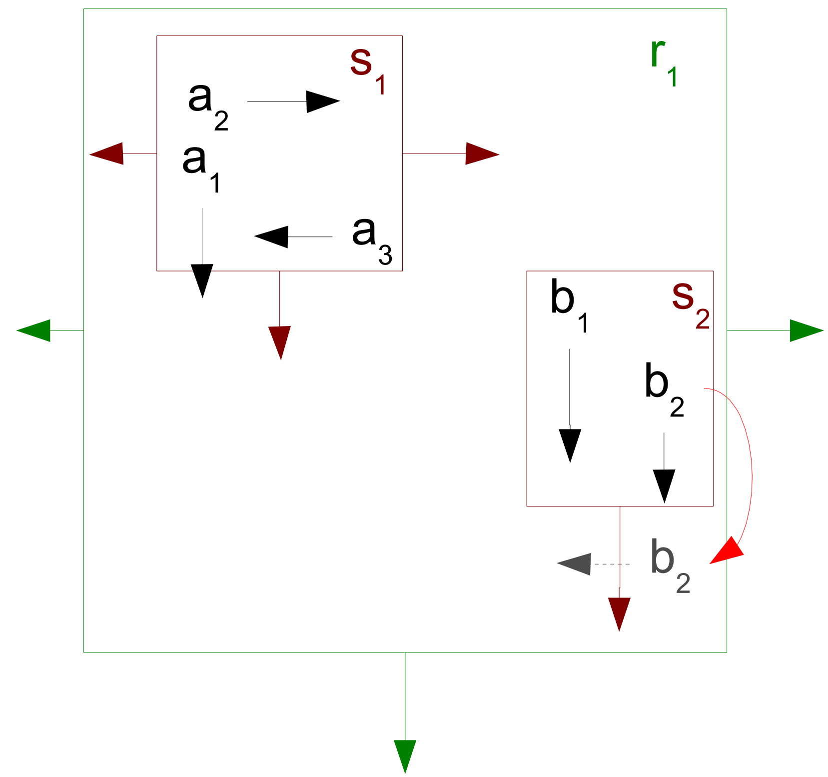




Example: update with TPR-tree

- Let's change the velocity of b_2! That means:
  1) **Delete** the old b_2: 2 traversals!
  2) **Insert** the new b_2: 2 traversals!





- 1 traversal = 3 I/Os
- 1 update = 12 I/Os!
- **Conclusion:** TPR-tree updates are expensive

Talk Outline

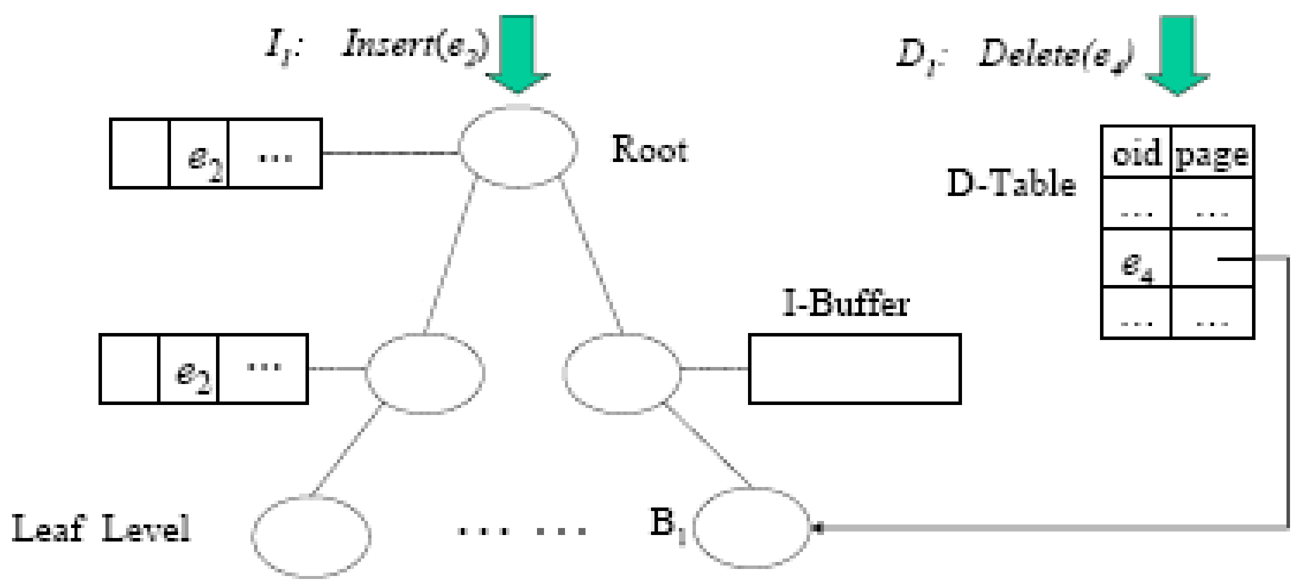
- Motivation
- Example: updates with TPR-tree
- Proposal: Lazy Group Update
- Updates with LGU
- Performance results
- Strong and weak points
- Relation to my project
- Conclusion

Why Lazy Group Update (LGU)?

- Are 4 traversals for a single update necessary? Observations:
  - The top-down traversal is redudant during deletion if we can access the leaf level directly
  - The deletion and insertion algorithms do not use the available main memory
  - Several updates to the same leaf could share a single traversal

LGU: the data structure

- Adaptable to most R-tree variants (R, TPR, ...)
- Memory-based D-Table
- Disk-based I-Buffers for each node
- Direct leaf level lookup table

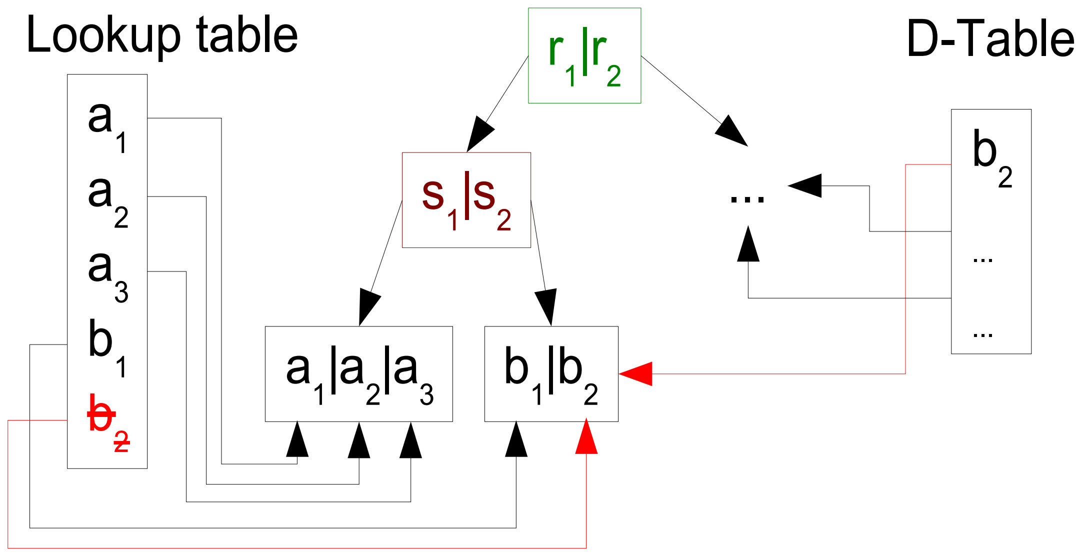


Talk Outline

- Motivation
- Example: updates with TPR-tree
- Proposal: Lazy Group Update
- Updates with LGU
- Performance results
- Strong and weak points
- Relation to my project
- Conclusion

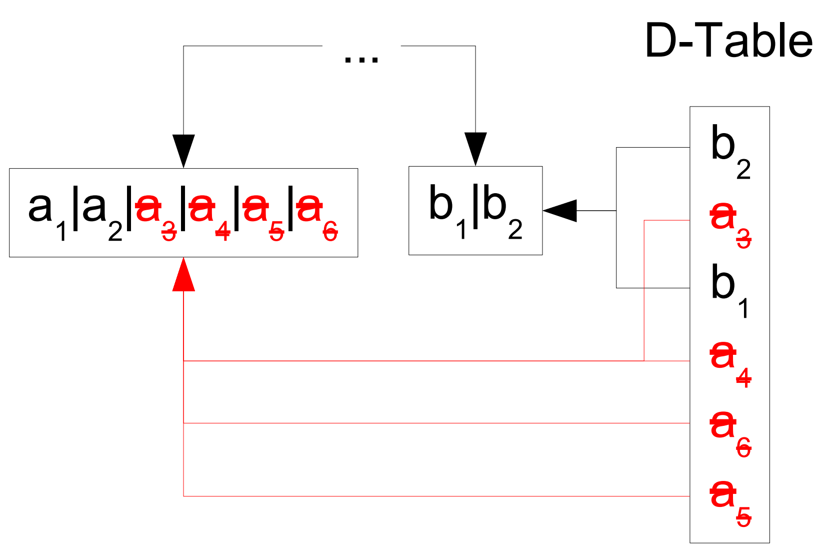
Example: deletions with LGU

- New deletions: (example: b_2)
  - Remove the entry from the lookup table
  - Add a deletion entry to the D-Table
  - No I/O



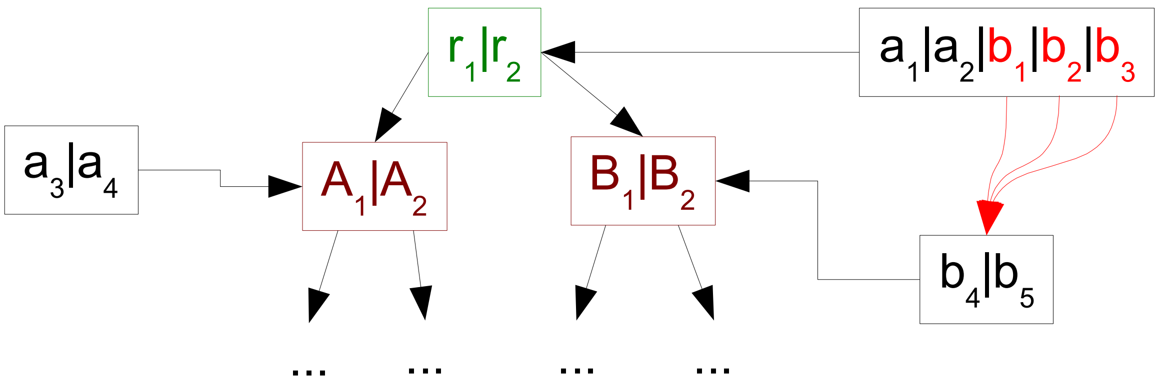
Example: deletions with LGU, cont.

- When the D-Table is full:
  - Execute the largest group of deletions going to a single page
  - 2 I/Os for *n* deletions



Example: insertions with LGU

- New insertions are added to the root node I-Buffer in main memory
- When any I-Buffer gets full:
  - Part of it is pushed down to the I-Buffers or leaf nodes below



LGU insertion: push down strategies

- What part of a full I-Buffer should be pushed down?
  - Whole buffer
    - Naive choice
    - Very small groups are pushed too; 1 I/O per 1 insertion possible
  - The largest group
    - No I/O wastage for small groups!
  - The "youngest" group
    - Reasoning: old entries may become obsolete soon

LGU updates: summary

- Insertions:
  - Processed lazily in batches top-down
  - It will take several buffer push-downs for any given insertion to reach leaf level
- Deletions:
  - Processed lazily in batches bottom-up
- Less than 1 I/O per 1 update on average
- Lazy updates: additional burden for queries

Talk Outline

- Motivation
- Example: updates with TPR-tree
- Proposal: Lazy Group Update
- Updates with LGU
- Performance results
- Strong and weak points
- Relation to my project
- Conclusion

Performance results

- In a nutshell: TPR-based LGU outperforms TPR by up to 2 orders of magnitude
- Similarly, R-based LGU outperforms related work by order of magnitude
- Main measure of performance:
  - Workload (combined update/query) throughput
  - A representative example:
    - TPR-based LGU: ~ 250 operations per second
    - TPR: ~15 operations per second

Performance results, cont.

- Empirical comparison of push-down strategies:
  - Both largest-group and youngest-group strategies significantly outperform whole-buffer strategy
  - Largest-group is the best

Talk Outline

- Motivation
- Example: updates with TPR-tree
- Proposal: Lazy Group Update
- Updates with LGU
- Performance results
- Strong and weak points
- Relation to my project
- Conclusion

Strong points

- Tackles a very important scenario of frequent updates
- Provides quite accurate analytical model
- Extensive performance studies
  - Several related work data structures compared
  - Uniform, skewed, network datasets
  - Various workload sizes

Weak points

- ✗ Not enough algorithmic pseudocode
  - ✗ In particular, no exact delete algorithm
  - ✗ Contradictions between textual description and figures
- ✗ Lookup table requires a large amount of main memory which depends on dataset size
  - ✗ 60% or more of dataset!
- ✗ Youngest-group push-down strategy requires timestamps for data, but space overhead is not discussed
- ✗ Chaotic style

Talk Outline

- Motivation
- Example: updates with TPR-tree
- Proposal: Lazy Group Update
- Updates with LGU
- Performance results
- Strong and weak points
- Relation to my project
- Conclusion

Relation to my project

- The same setting – we are solving the same problem
- Both are based on the Buffer Tree idea
  - Both process updates lazily in batches
  - Different buffer data structures
- Different delete processing
  - LGU: bottom-up, additional data structure
  - My project: top-down, processed together with insertions

Talk Outline

- Motivation
- Example: updates with TPR-tree
- Proposal: Lazy Group Update
- Updates with LGU
- Performance results
- Strong and weak points
- Relation to my project
- Conclusion

Conclusion

- A relevant contribution to an active research area
- Demonstrates a convincing performance improvement and analytical model
- However, skimps over very important details