R-trees with Update Memos

ICDE'06 paper by

Xiaopeng Xiong and Walid G. Aref
Purdue University

Presented by
Laurynas Biveinis

Talk Outline

- Motivation
- Example: Updates with R-tree
- Related work: Bottom-up Updates
- Contribution: RUM-tree
- Experimental Evaluation
- Strong and Weak Points
- Relation to my Project
- Conclusion

Motivation

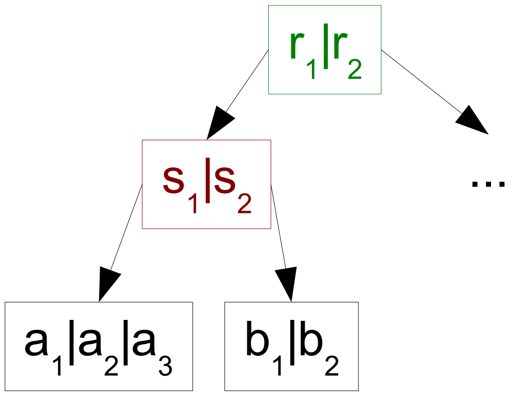
- Scenarios with continuous spatial data sampling are getting more and more common
  - 1 mln LBS users that send 1 update/hour
  - 280 updates/second!
  - Queries are relatively rare
- Wanted: a spatial disk-based index that can handle high volume of updates
- Is R-tree good enough?

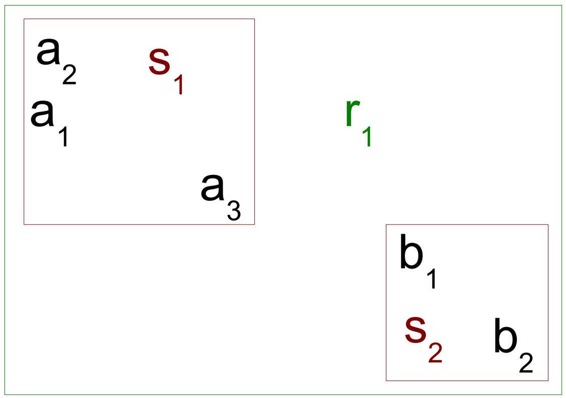
Talk Outline

- Motivation
- Example: Updates with R-tree
- Related work: Bottom-up Updates
- Contribution: RUM-tree
- Experimental Evaluation
- Strong and Weak Points
- Relation to my Project
- Conclusion

Example: Updates with R-tree

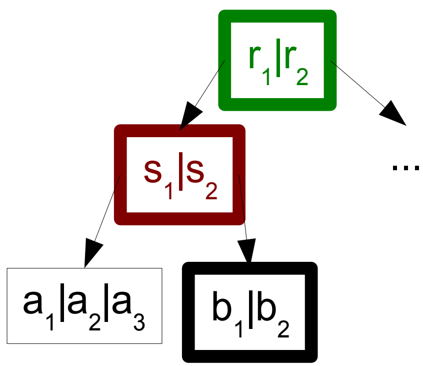
- R-tree: index of choice for low-dimensionality spatial data
- Index structure suited for efficient range queries on mostly static data

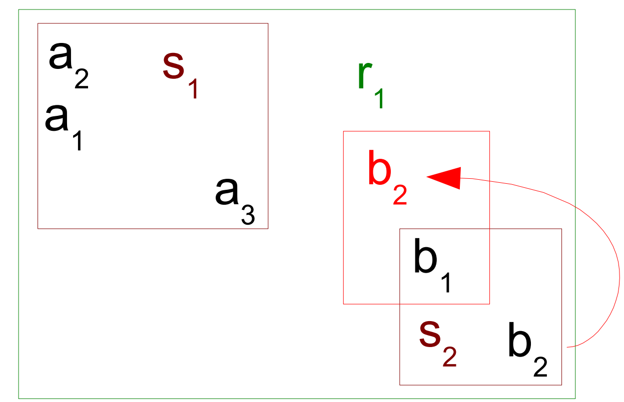




Example: Updates with R-tree

- Let's update position of b2
  1) **Delete** the old b_2: 2 traversals!
  2) **Insert** the new b_2: 2 traversals!

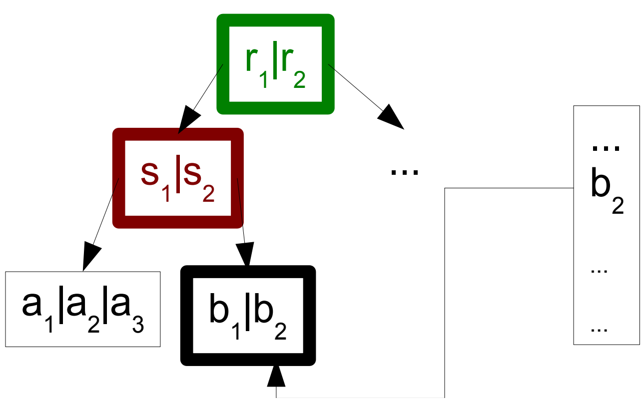




- 1 traversal = 3 I/Os
- 1 update = 12 I/Os!
- **Conclusion:** R-tree updates are expensive

How to Make Updates Cheaper?

- Top-down traversals do not do anything useful on upper tree levels if new object position is close to the old one
- Top-down traversal during deletion is redudant if leaf level can be accessed directly



Related Work: Bottom-up Updates

- FUR-tree by Lee et al in VLDB 2003
- Updates are processed bottom-up as locally as possible
  - If new position is close to the old one: update leaf
  - If not so close: traverse tree bottom-up as little as possible
- Performance is unstable and depends on characteristics of updates

A Different Approach: RUM-tree

- RUM-tree – „R-tree with Update Memo“
- Skip performing deletions altogether!
  - Store deletions in main memory – „Update Memo“
  - No top-down or bottom-up traversals at all
  - Let obsolete entries stay in the tree
  - But clean the tree periodically from them – „Garbage Cleaner“
- Perform insertions as for ordinary R-tree
- Enhance query algorithm to filter obsolete entries

RUM-tree: the Data Structure

- Leaf entries are timestamped to differentiate between up to date and obsolete entries:
  - <MBR, oid, **stamp**>
- Update Memo structure:
  - Entry format:
  - <object-id, latest-timestamp, max-num-of-obsolete>
  - Primary access on object-id
  - Invariant max-num-of-obsolete > 0
  - Requires very little amount of main memory

RUM-tree: Deletions

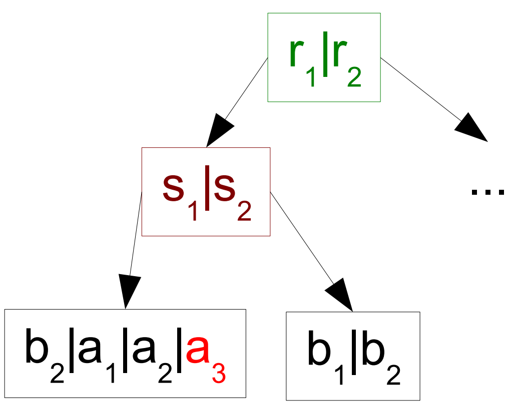
- Let's delete the old position of a_3
- No obsolete a_3 entries in the tree yet
- No disk I/O!

Update Memo

Object	Time	Max Old
b_2	1	2

Update Memo

Object	Time	Max Old
a_3	2	1
b_2	1	2



RUM-tree: Deletions, cont.

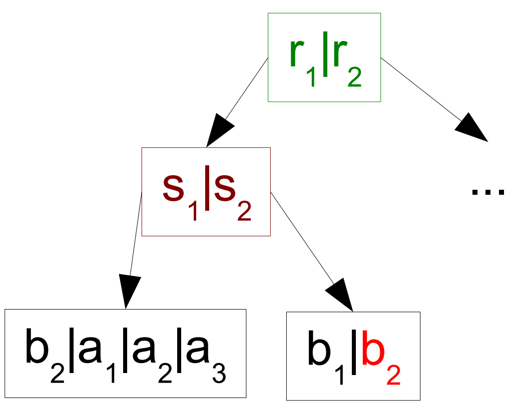
- Let's delete the old position of b2
- One old position of b2 already in the tree
- No disk I/O

Update Memo

Object	Time	Max Old
a_3	2	1
b_2	1	2

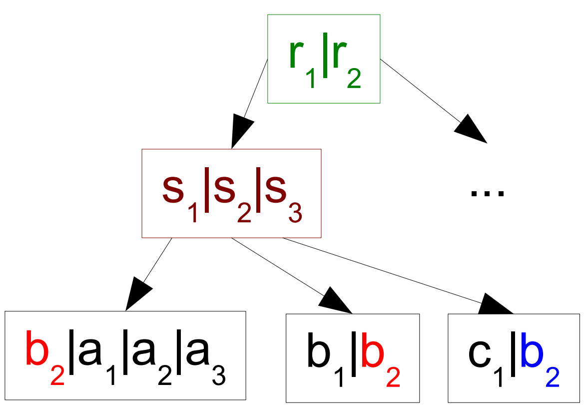
Update Memo

Object	Time	Max Old
a_3	2	1
b_2	3	3



RUM-tree: Insertions

- Let's insert a new position of b2
- Ordinary R-tree insertion
- Update Memo update
- If no old entry in Update Memo: create new one



Update Memo

Object	Time	Max Old
a_3	2	1
b_2	3	3

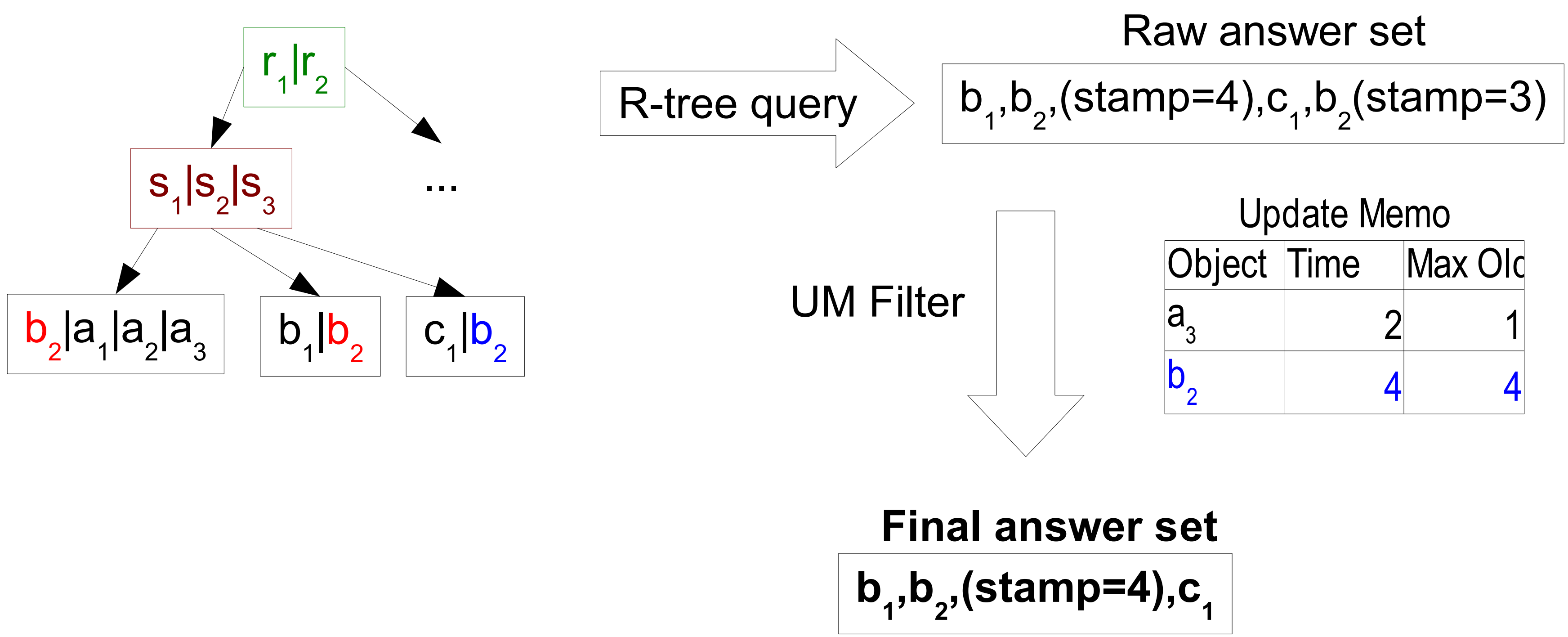
⇒

Update Memo

Object	Time	Max Old
a_3	2	1
b_2	4	4

RUM-tree: Queries

- Ordinary R-tree query with Update Memo filter
- Intuition: the bigger UM, the slower the query
- Example: range query with $MBR(s_2)\ U\ MBR(s_3)$



RUM-tree: Garbage Cleaning

- With previous algorithms:
  - Disk tree only grows with time
  - Update Memo only grows with time
  - Performance, esp. of queries, drops with time
- So, sometimes the garbage must be disposed

Update Memo

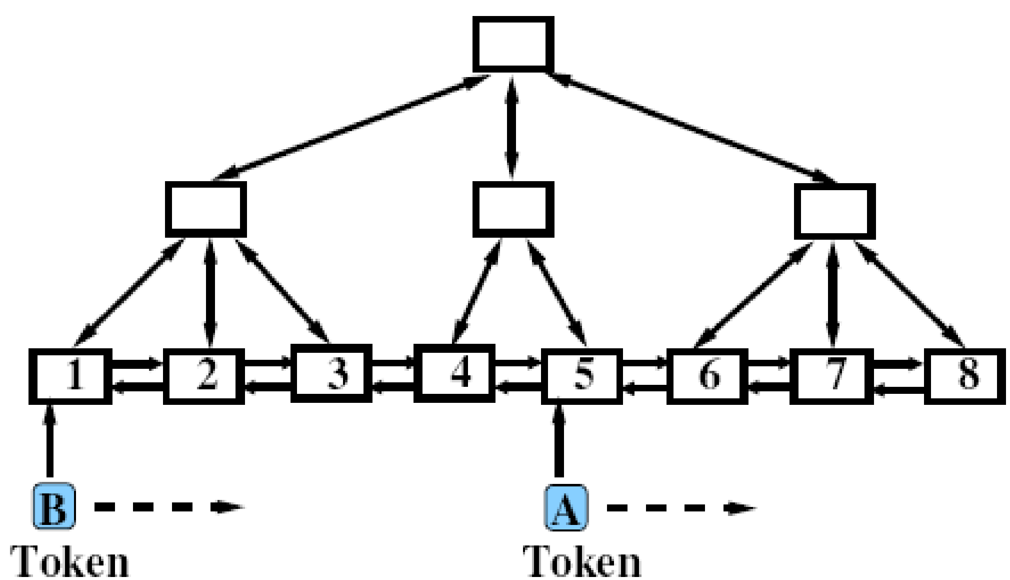
Object	Time	Max Old
a_3	2	1
b_2	4	4

Update Memo

Object	Time	Max Old

RUM-tree: Garbage Cleaning, cont.

- Leaf level nodes linked to a list
- All obsolete entries from each node are cleaned by a so-called token
- After *I* updates token is passed to the next node



RUM-tree: Garbage Cleaning, cont.

- Another way: to clean garbage whenever node is touched
- Combined with cleaning token method
- Useful definitions to measure GC effectiveness
  - Garbage ratio (*gr*): number of obsolete entries divided by total number of objects
  - Inspection ratio (*ir*):number of GC-inspected nodes divided by number of updates
- We want to minimize both *gr* and *ir*.

Talk Outline

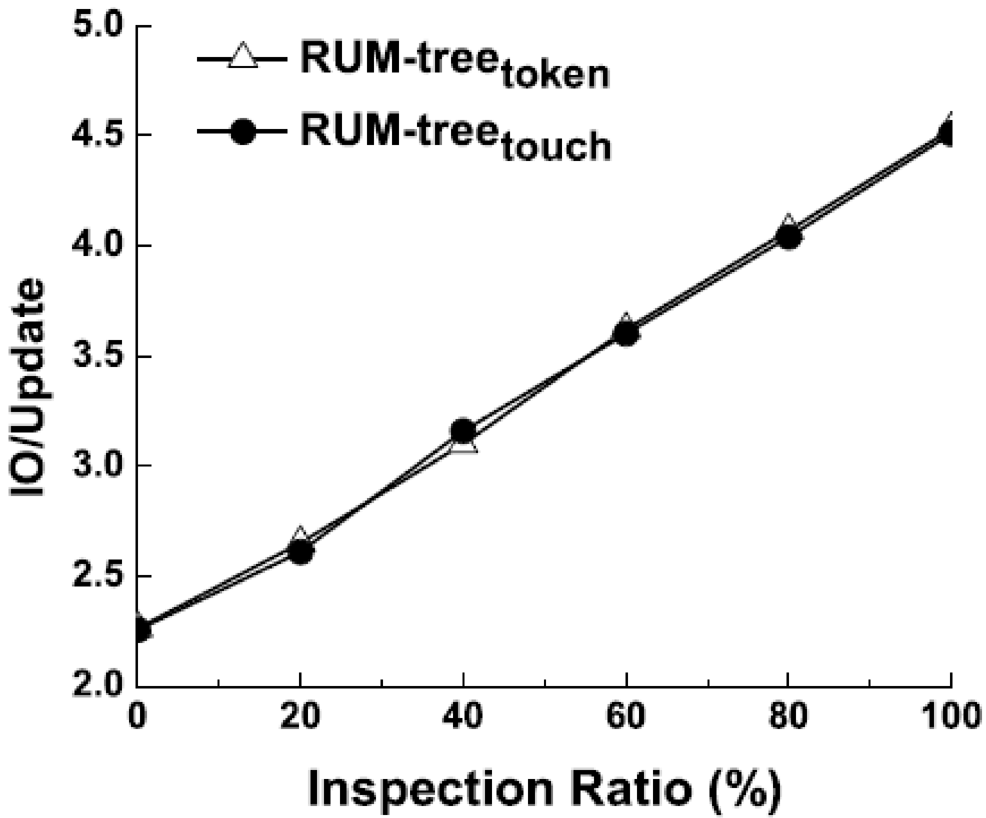
- Motivation
- Example: Updates with R-tree
- Related work: Bottom-up Updates
- Contribution: RUM-tree
- Experimental Evaluation
- Strong and Weak Points
- Relation to my Project
- Conclusion

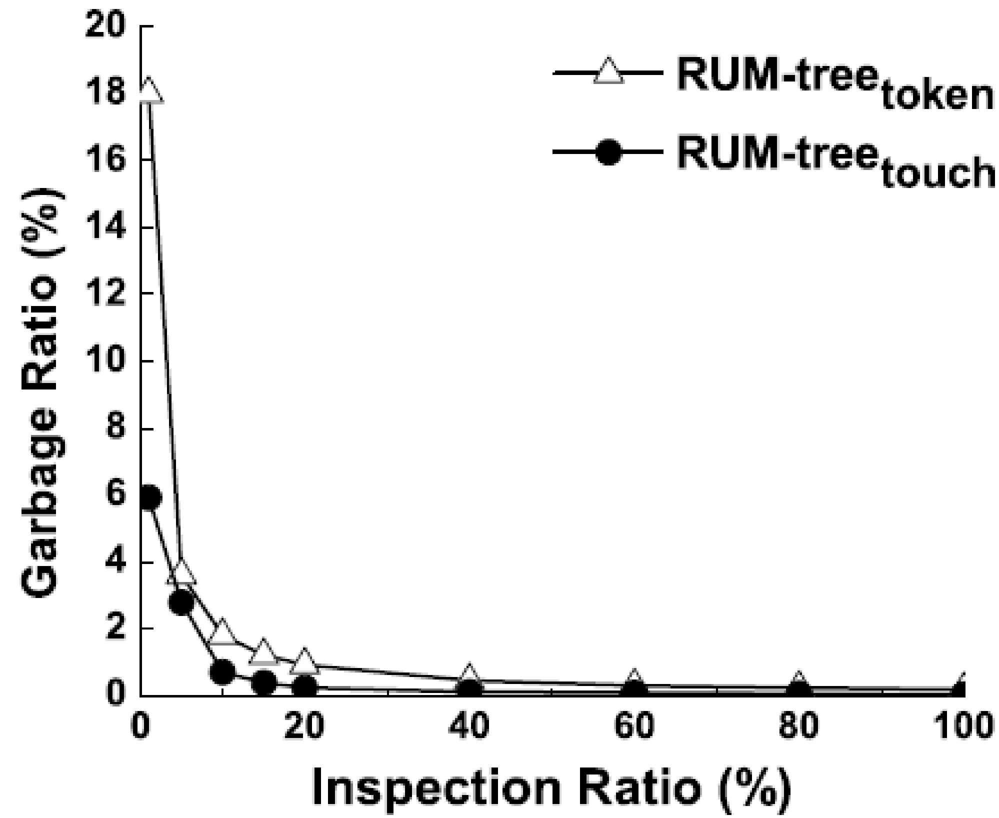
Experimental Evaluation

- Los Angeles street network
- Objects moving along the network generated by Brinkhoff generator

PARAMETERS	VALUES USED
Number of objects	**2M**, 2M$\sim$20M
Moving distance between updates	**0.01**, 0$\sim$0.01
Extent of objects	**0**, 0$\sim$0.01
Node size (bytes)	1024, 2048, 4096, **8192**
Inspection Ratio of RUM-tree	**20%**, 0%$\sim$100%

GC Parameter Evaluation and Tuning

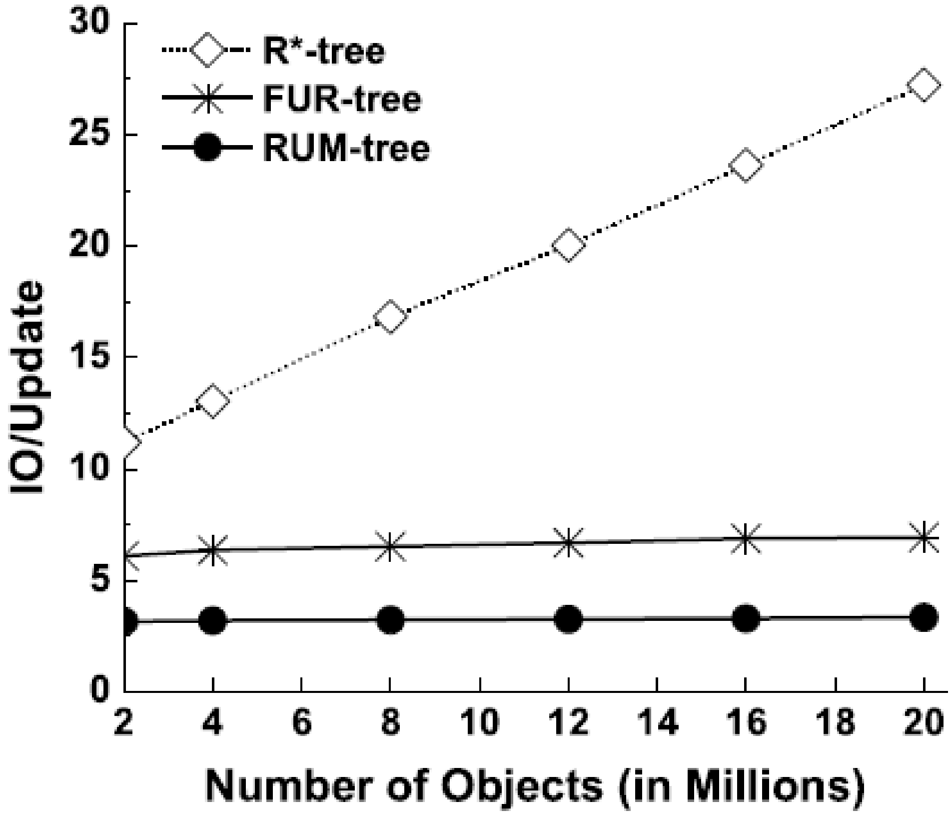


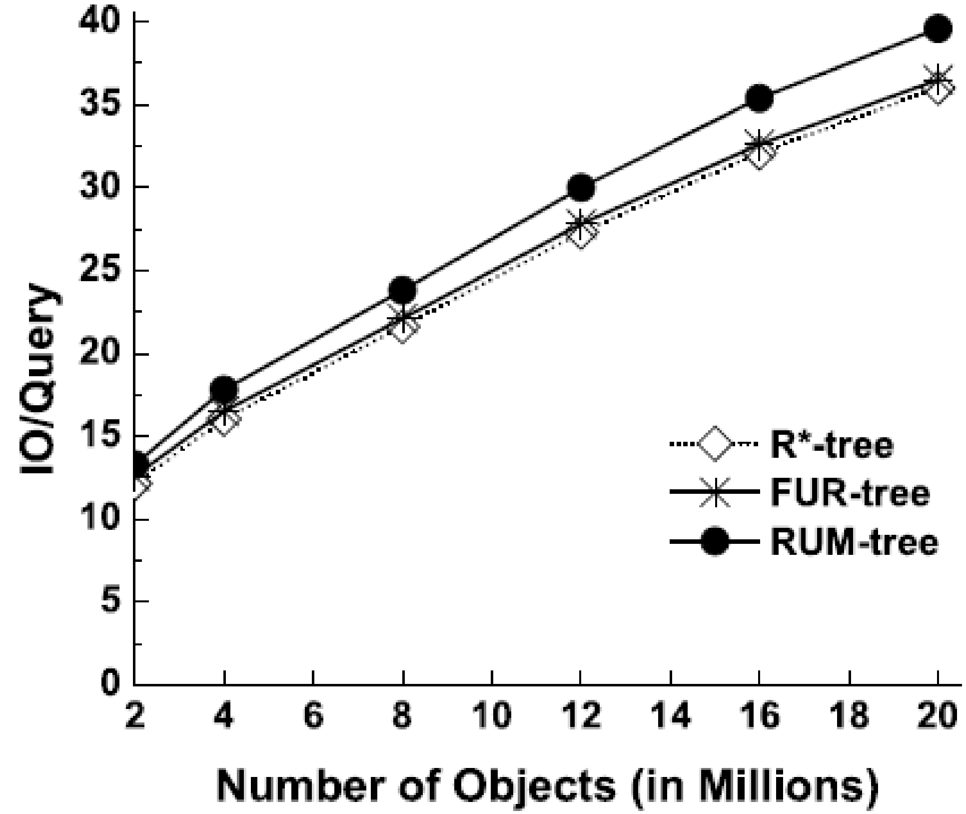


Performance Comparison

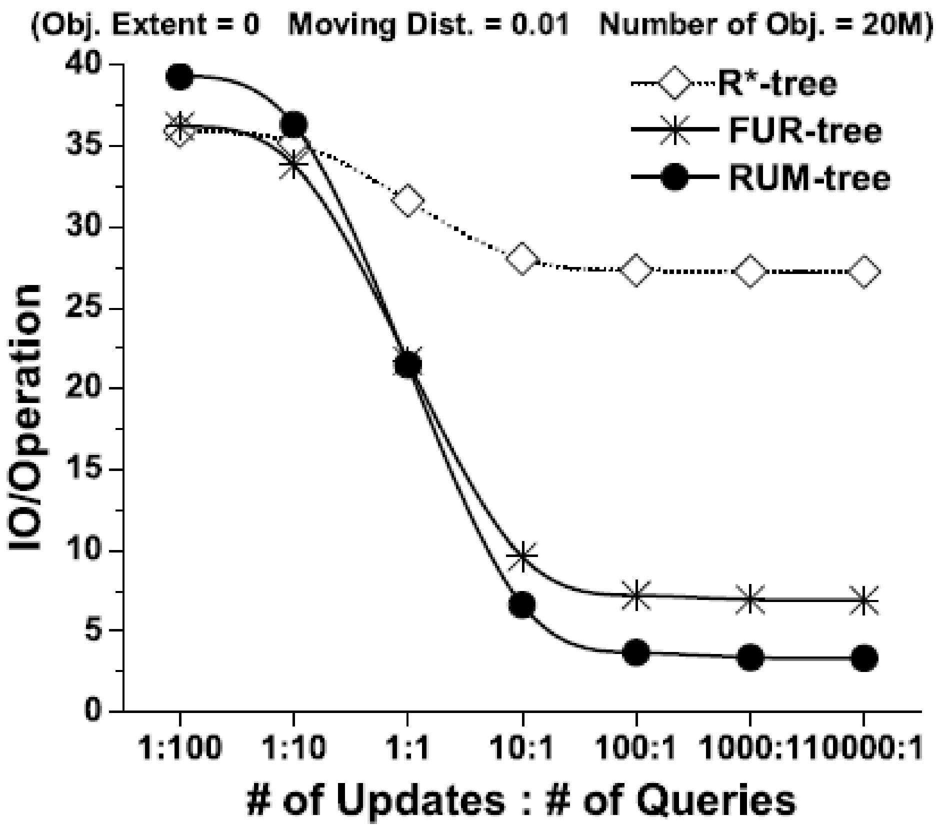
- Trees compared:
  - R*-tree
  - FUR-tree
    - Previously discussed related work: bottom-up updates
  - RUM-tree
- All internal tree nodes stored in main memory

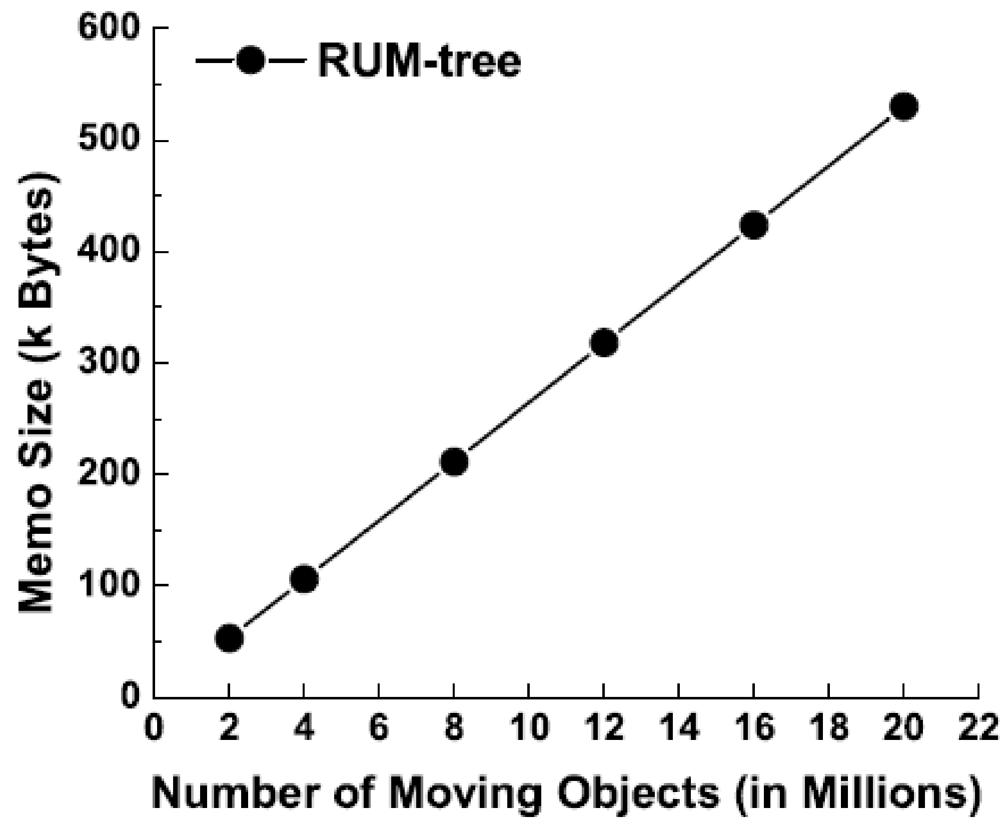
Performance Comparison Results





Performance Comparison Results, cont.





Performance Comparison Conclusion

- RUM-tree update cost: ~3 I/O
  - Twice better than FUR-tree
  - 3-10-... times better than R*-tree
  - Scales very well
- All trees have similar query cost

Talk Outline

- Motivation
- Example: Updates with R-tree
- Related work: Bottom-up Updates
- Contribution: RUM-tree
- Experimental Evaluation
- Strong and Weak Points
- Relation to my Project
- Conclusion

Strong Points

- An important problem setting
- Works with any amount of main memory
  - Update Memo is very small
- Stable performance
- Proposed solution discussed thouroughly
  - Correctness, crash recovery, cost model, concurrency control
- Comprehensive experimental evaluation
  - Although only with network dataset
- Clear and concise writing style

Weak Points

- Fails to consider garbage cleaning with only clean-on-touch
  - Much simpler data structures and algorithms
    - No leaf-level linked list, no parent pointers, no tokens
  - Garbage ratio = 6%, compared to ~1% in paper experiments
- Crash Recovery treatment has issues
  - It is possible to lose deletions
- Cost model falls apart with *ir = 0%*
- Performance evaluation with uniform and skewed datasets would add value

Talk Outline

- Motivation
- Example: Updates with R-tree
- Related work: Bottom-up Updates
- Contribution: RUM-tree
- Experimental Evaluation
- Strong and Weak Points
- Relation to my Project
- Conclusion

My Project (R^R-tree)

- The same setting, but persistence is not assumed
  - Frequent updates
- Disk-based R-tree
- Main memory buffer of incoming updates
- When buffer gets full, its updates are processed on the main tree in batch
  - Performance win by making lots of updates share same I/O operations

Relation to my Project

- Similar in that incoming deletions are procesed in memory, but data structures differ very much
- Different persistence assumptions, not really comparable performance
  - RUM-tree and related work: index is persistent
    - Each update costs at least 1 I/O by definition
  - R^R-tree: index is partially main-memory based
    - Each update costs ~ 0.1 I/O

Conclusion

- Well-written paper on important topic
- Contribution: an R-tree modification, that:
  - Supports frequent updates
  - Grounded by theoretical analysis
  - Convincingly outperforms related work
- Problem setting similar to my project
  - A key difference in persistence
  - Thus cannot be directly compared