AALBORG UNIVERSITY

# Main-Memory Operation Buffering for Efficient R-Tree Update

***Simonas Šaltenis***

joint work with

Laurynas Biveinis and Christian S. Jensen

presented at VLDB 2007, Vienna

Department of Computer Science

Aalborg University

**daisy.aau.dk**

# Outline

- Motivation
- Contributions
- Related work
- Assumptions
- Updates in R-trees
- The $R^R$-tree
  - Structure and algorithms
  - Results of performance studies
- Conclusions

# Motivation

- A typical pervasive computing scenario
  - Sampling of continuous processes via large numbers of sensors
  - Maintenance of an up-to-date current state of the processes
  - Query processing against the current state
- Example: moving objects
  - The current positions of moving objects
  - Large populations of objects are anticipated (mobile phone users)
  - Updates are very frequent.
    - E.g., 600,000 objects, each issuing an update every minute yields 10,000 updates per second.
- Indexing is essential to efficient query processing.
- The index must be stored on disk, at least in part.
- Existing indices do not  support massive update loads.

# Contributions

- The paper proposes a new main-memory buffering technique for R-tree-type indices.

- Index maintenance and querying algorithms are provided.
- Analysis
  - An accurate analytical cost model is provided.
  - An empirical I/O performance study is conducted.
- Properties
  - Assumes that index updates need not be written to disk immediately
  - Supports normal index operations, including instantaneous queries
  - Supports disk-based index storage
  - Works with any amount of main memory
  - Exploits all available main memory
  - Supports more frequent index updates than possible so far

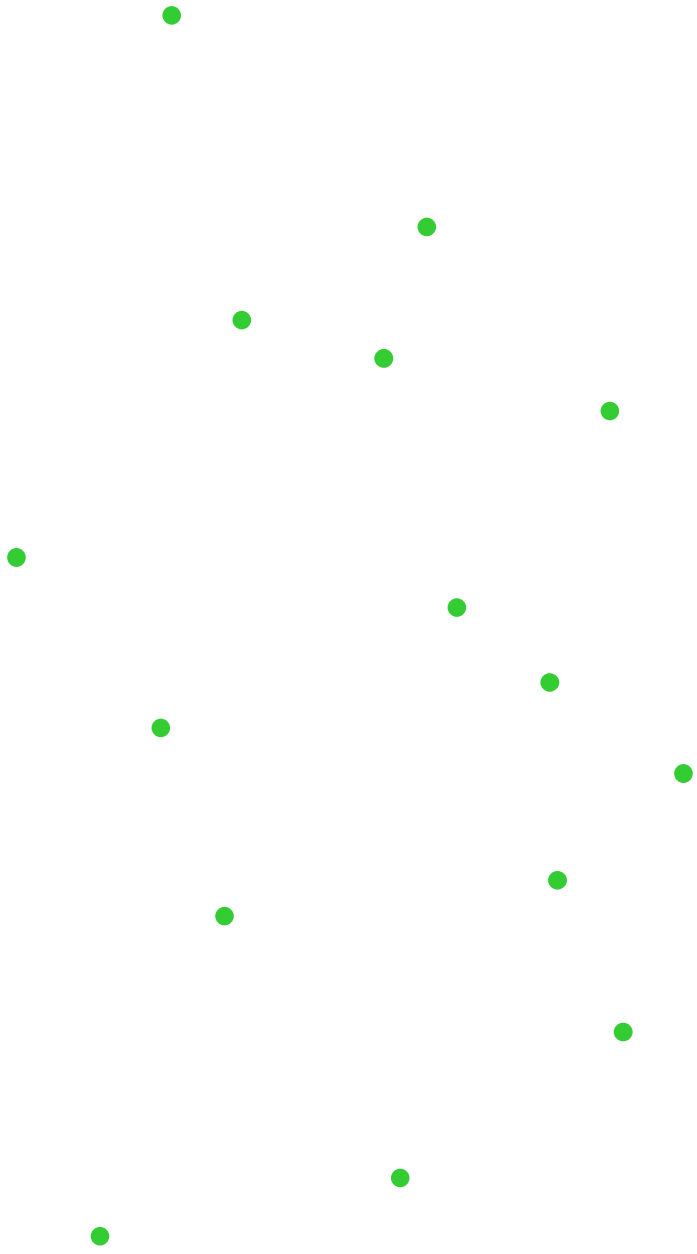
# Existing Solutions

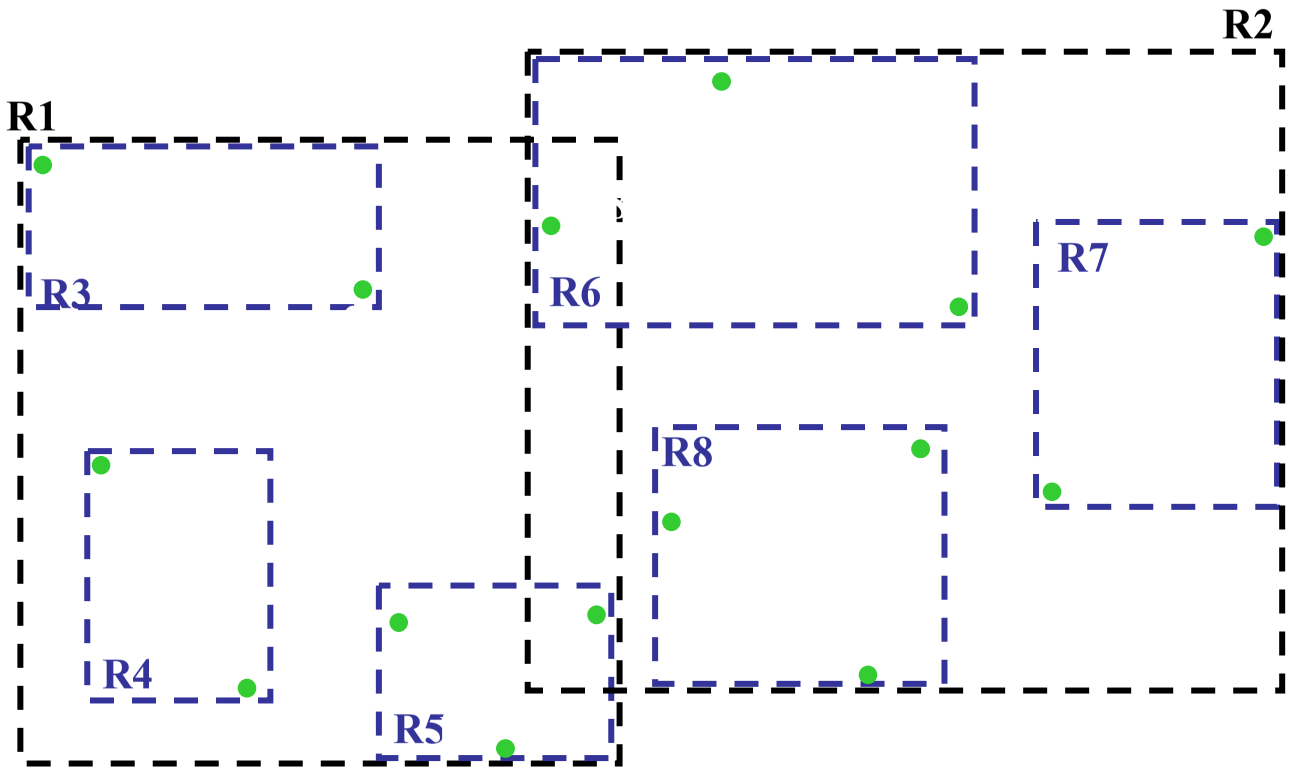
- Write-back LRU buffer
  - We compare with this.
- Modeling of data by functions of time
  - Orthogonal to this paper's proposal
- Bottom-up update techniques with direct leaf-level access
  - We need better performance.
- Buffer-tree-based solutions [e.g., Arge et al., 2002]
  - Complex; do not support instantaneous queries
- LGU [Lin and Su, 2005]
  - Complex; *requires* large amount of main-memory (relative to the data set size)
- The RUM-tree [Xiong and Aref, 2006]
  - We compare with this.
- LUGrid [Xiong et al., 2006]
  - The same persistency assumptions
- Various buffering techniques for B-trees
  - We build on these
  - Partial and selective buffer emptying not explored

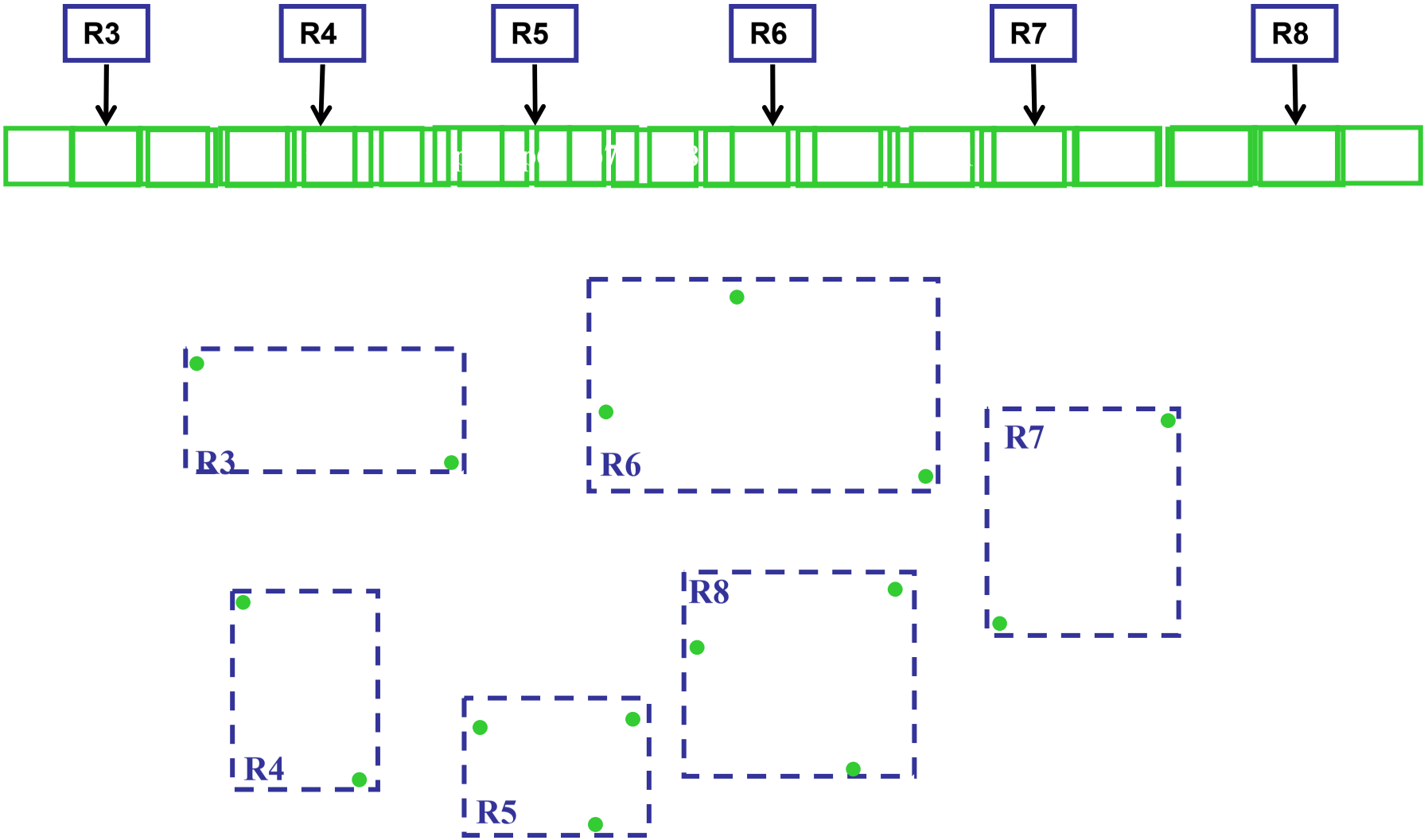
# Assumptions

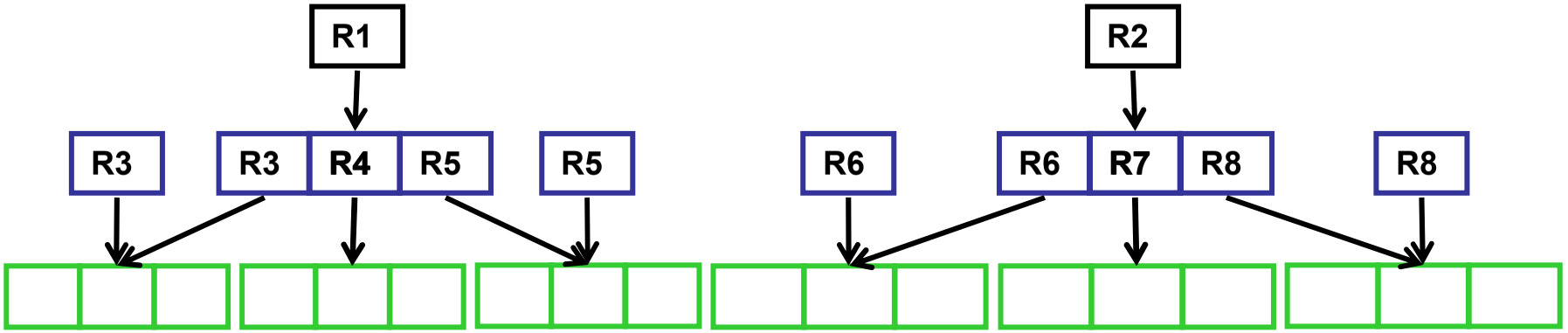
- The current state is a set of (id, key) pairs
  - For moving objects, key = (x, y)
- Each object sends a sequence of updates
  - *Update* (id, old_key, new_key)
    - Delete (id, old_key) + Insert (id, new_key)
- Softer persistency requirements
  - Not all of the index has to be committed on disk in-between operations
  - After a system crash
    - Wait (one minute) while the index is being populated
    - Alternatively, rebuild the index from the base data

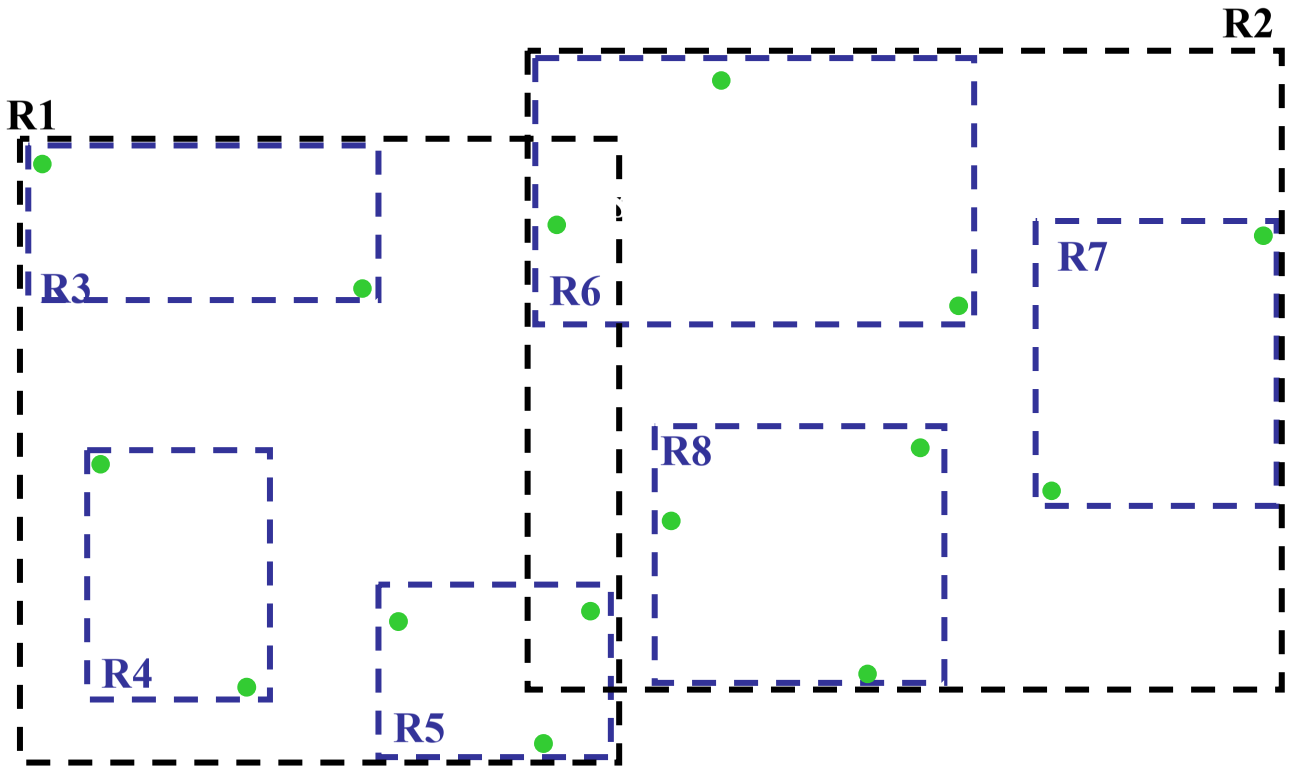
# Background: R-trees

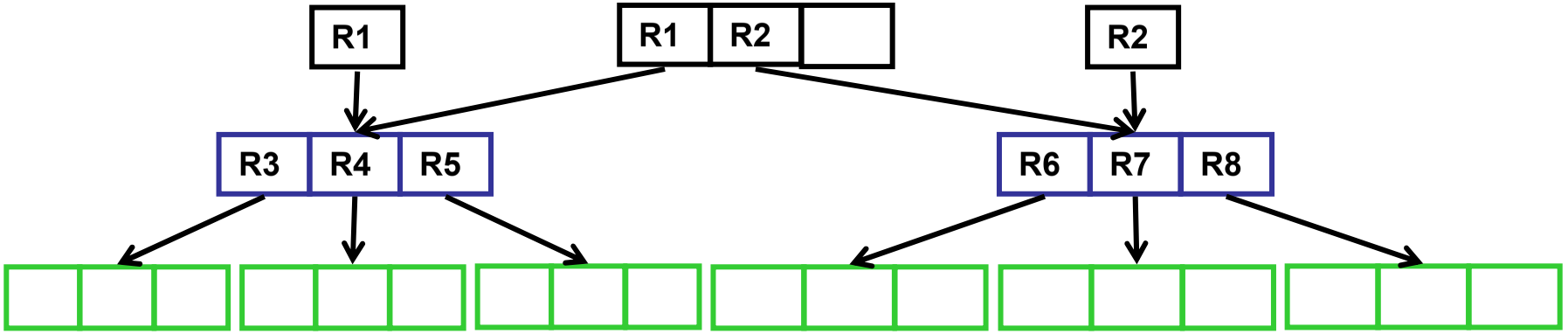


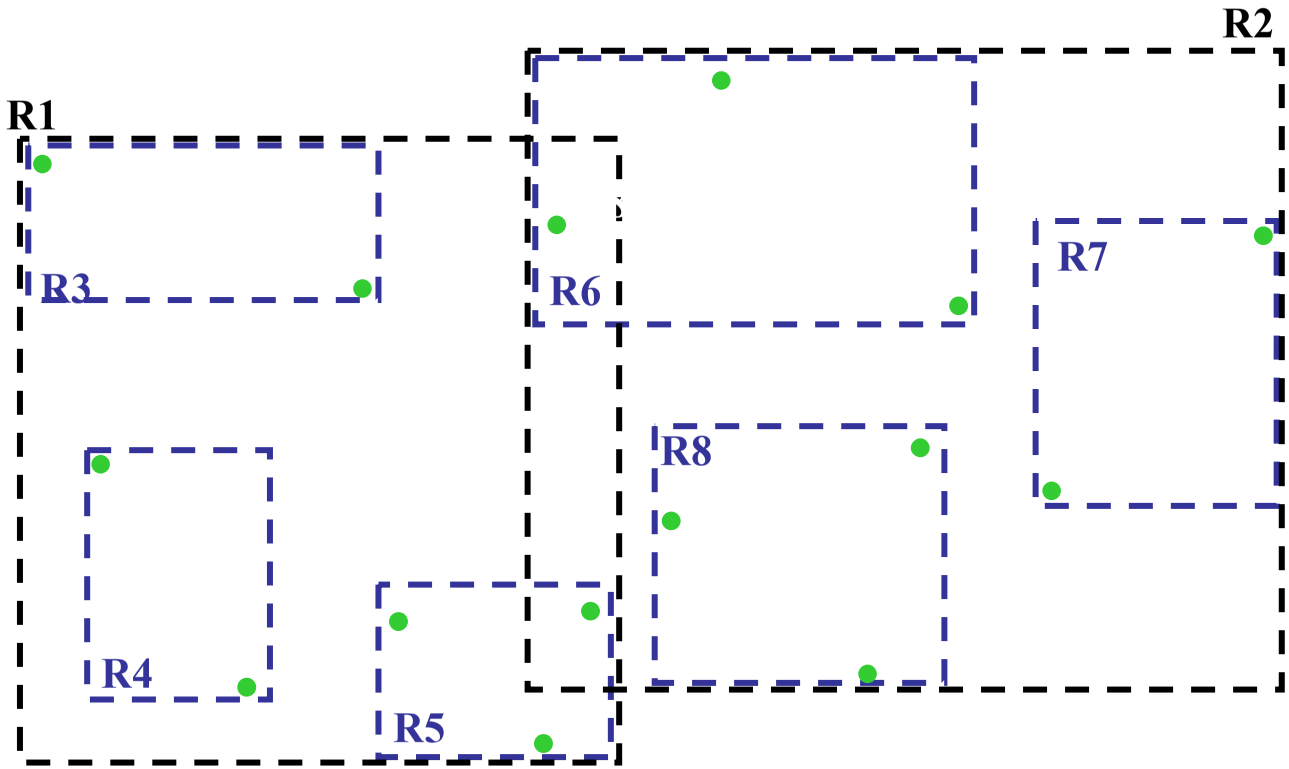

R1
R2
R3
R4
R5
R6
R7
R8


R3
R4
R5
R6
R7
R8
R3
R6
R7
R8
R4
R5


R1
R2
R3
R3
R4
R5
R5
R6
R6
R7
R8
R8

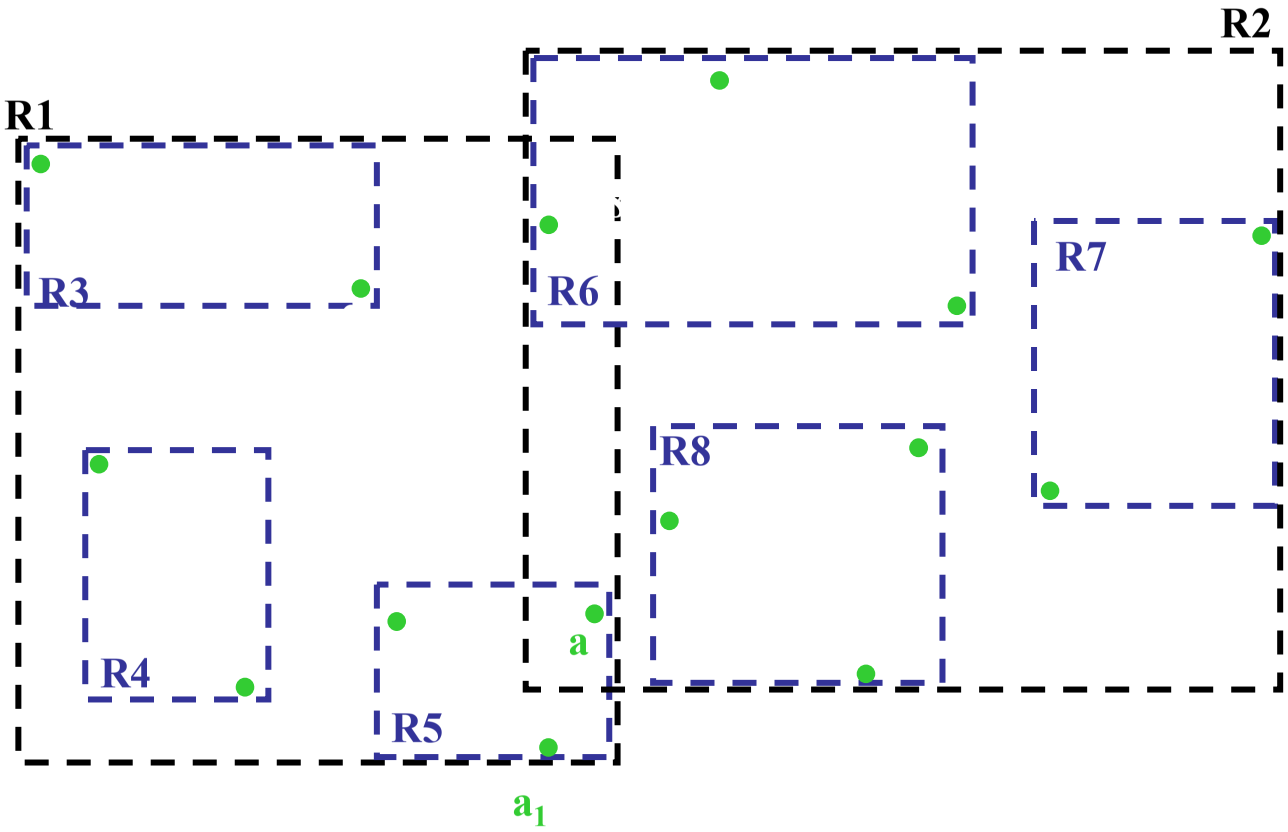

R1
R2
R3
R4
R5
R6
R7
R8


R1
R1
R2
R2
R3
R4
R5
R6
R7
R8

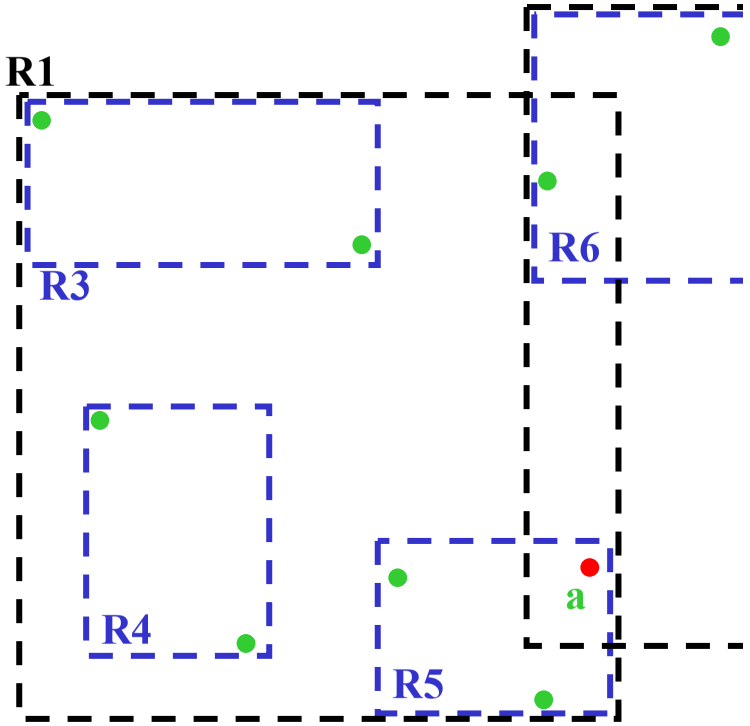

R2
R1
R3
R6
R7
R8
R4
R5

# Update operations in R-trees

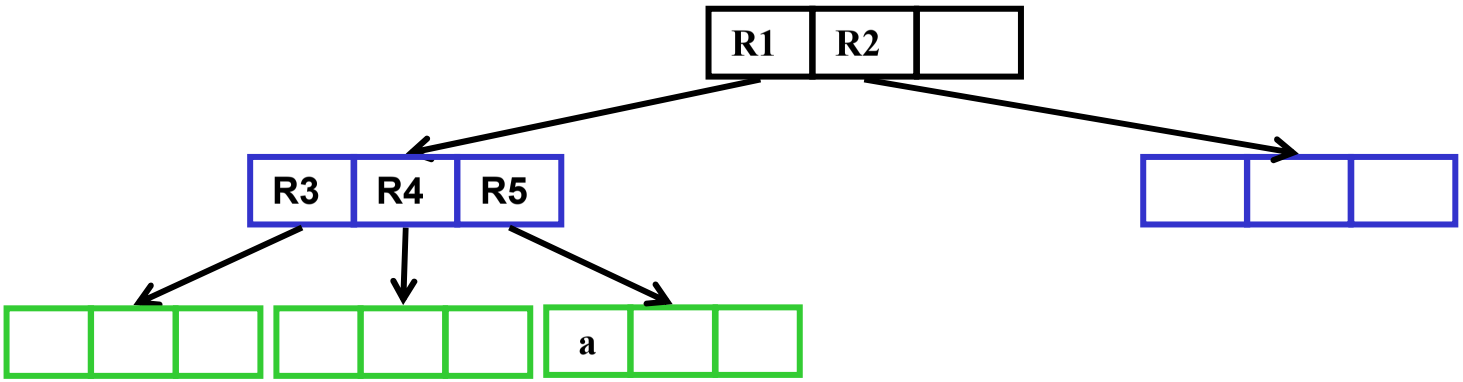
# Operations: $a^-$ $a_1{}^+$



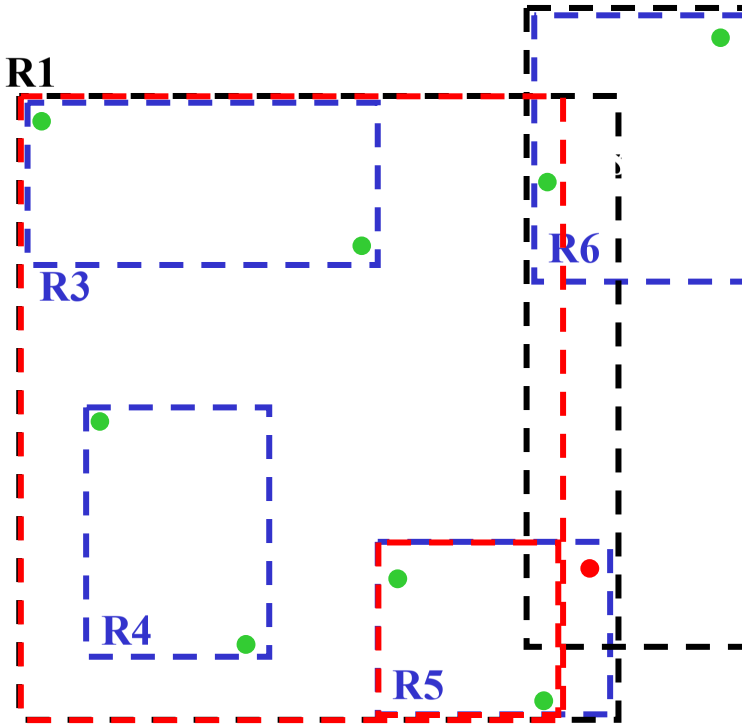
# Operations: $a^-$



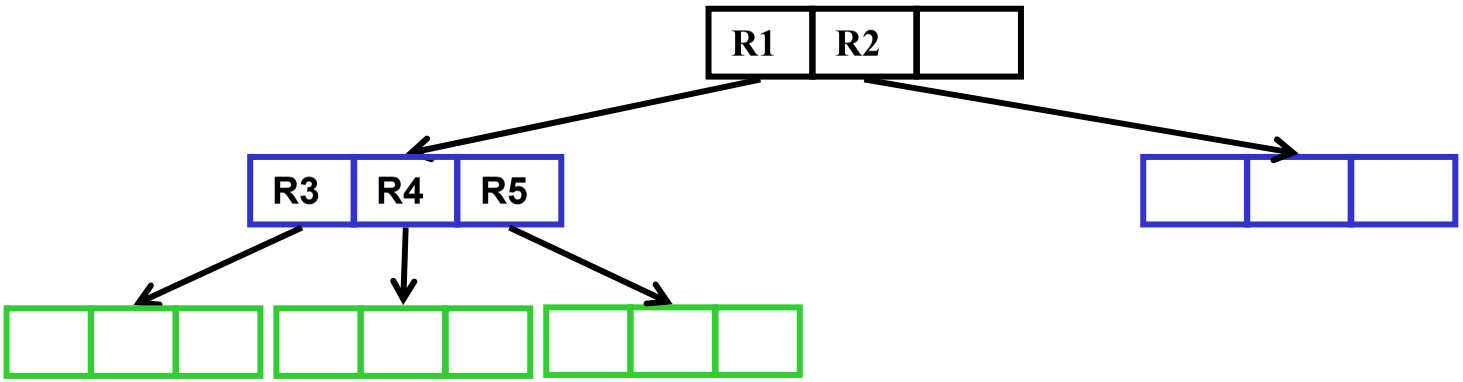

- **R-tree deletion, 1st step**:
  - Find the object – several partial paths may be followed
  - Remove the object from the leaf node



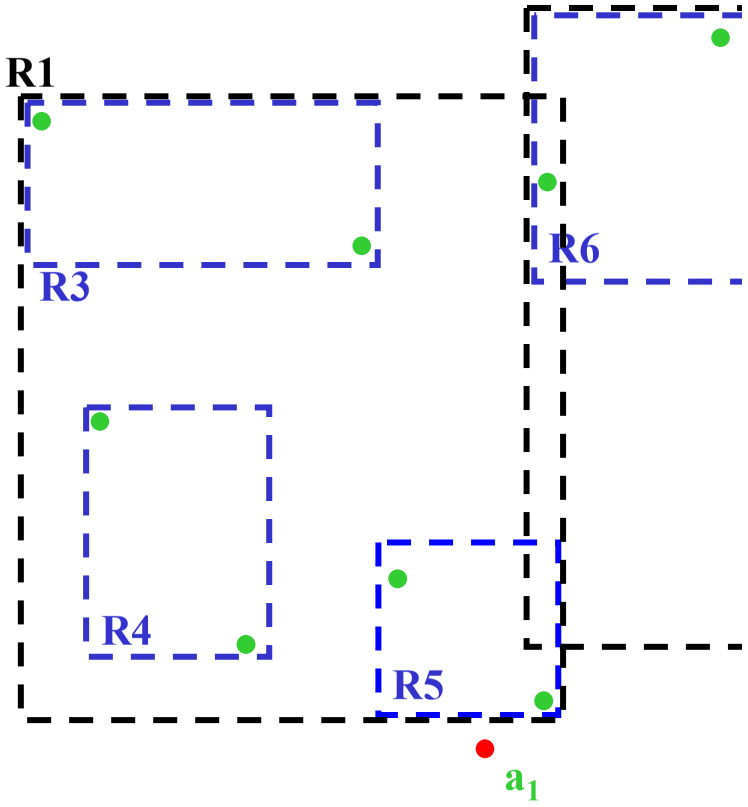
# Operations: $a^-$



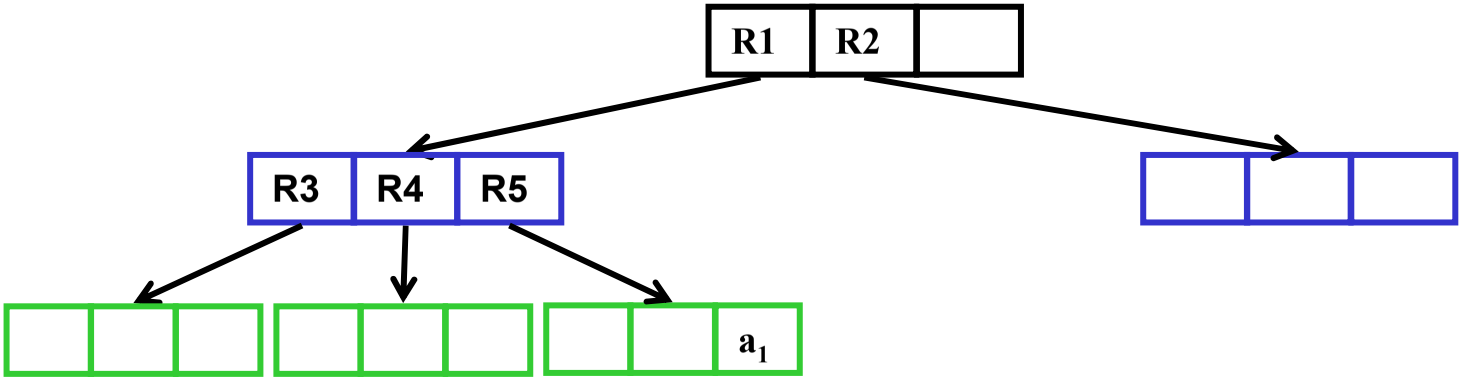

- **R-tree deletion, 2nd step**:
  - Traverse up the tree and update bounding rectangles



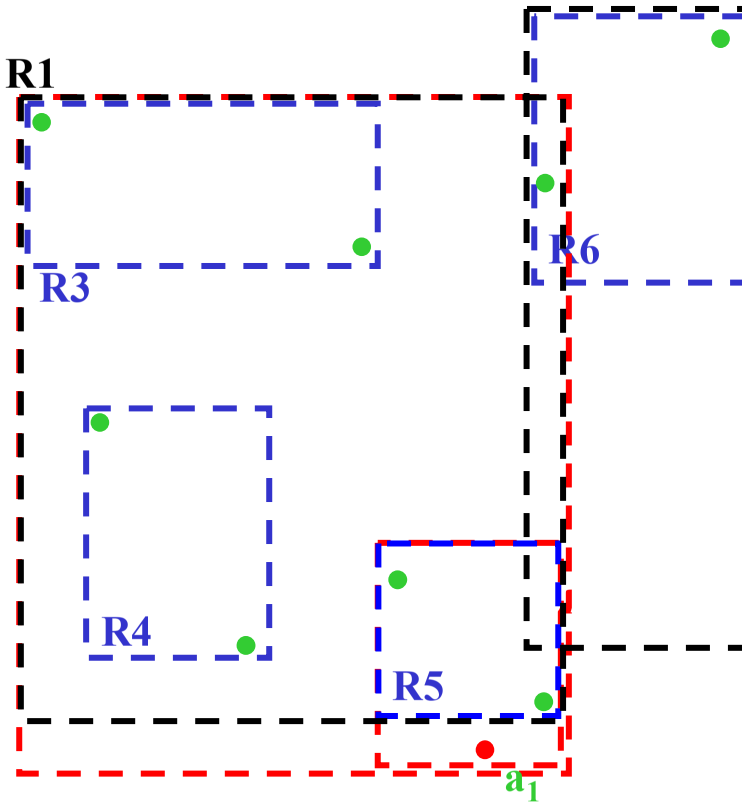
# Operations: $a^-$ $a_1^+$



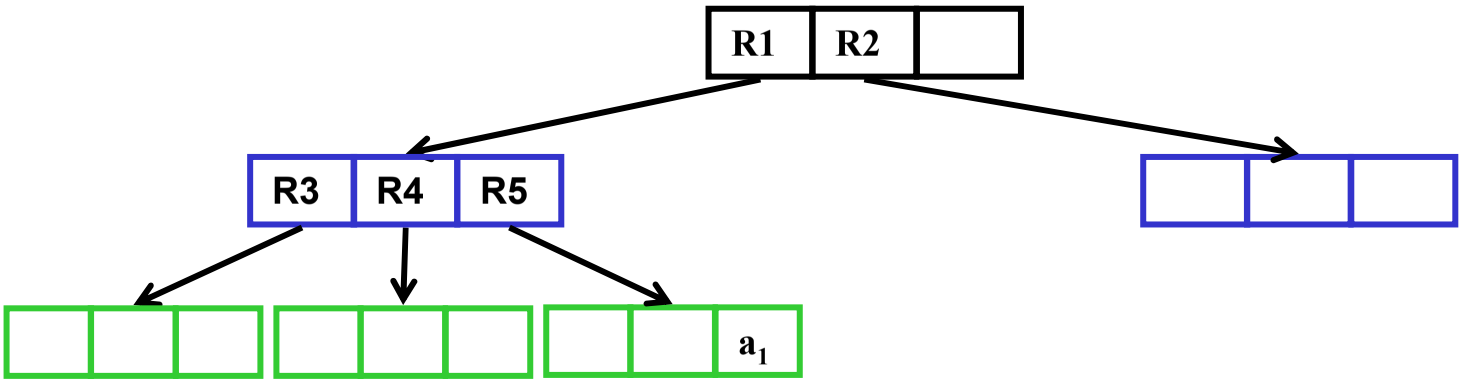

- **R-tree insertion, 1st step**:
  - Use R-tree insertion heuristics to find a path to a leaf
  - Insert *a1* to the leaf node.



# Operations: $a^-$ $a_1^+$

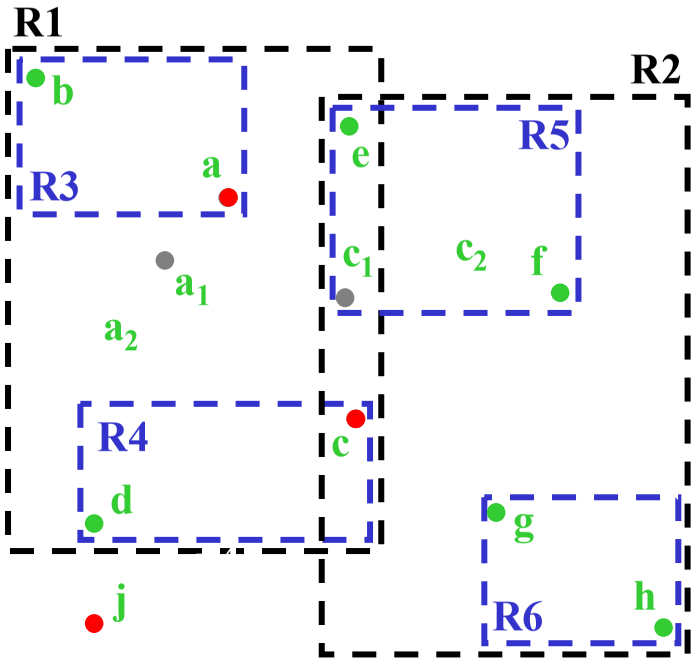



- **R-tree insertion, 2nd step**:
  - Traverse up the tree and update bounding rectangles (and possibly split nodes)
- In total: 4 traversals for an update.

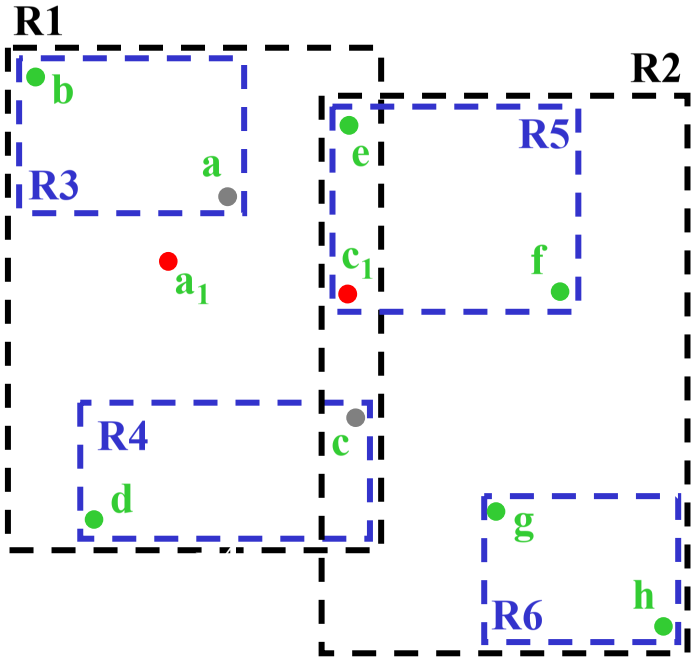


# Update operations in $R^R$-trees

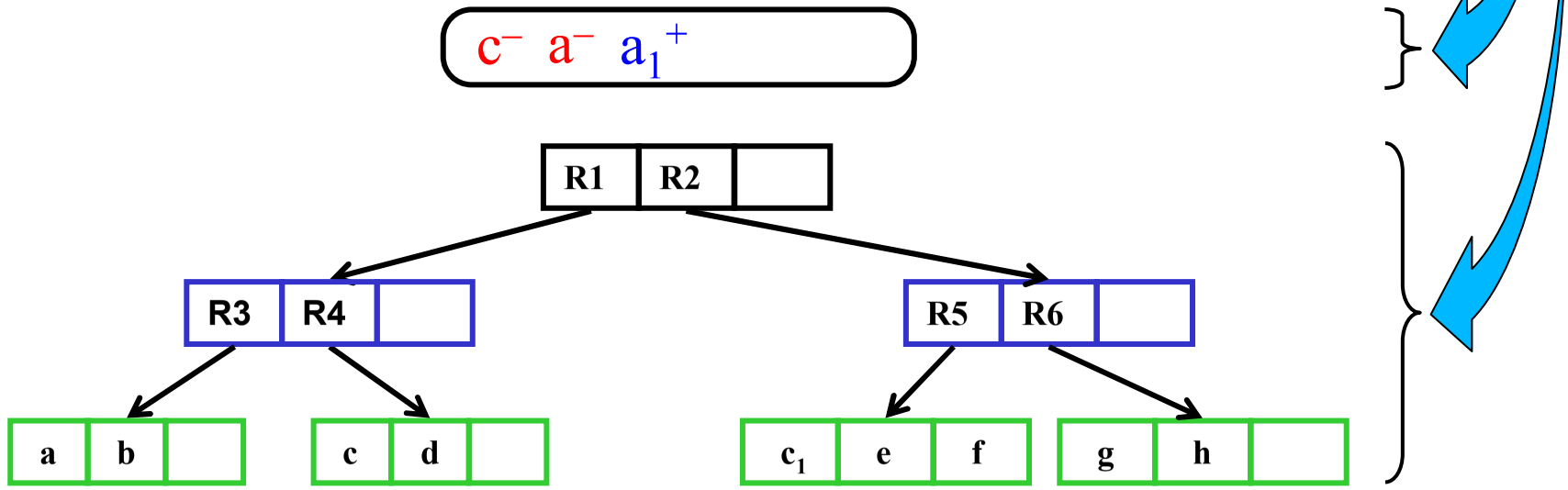
Operations: $c^-$ $c_1^+$ $a^-$ $a_1^+$ $a_1^-$ $a_2^+$ $c_1^-$ $c_2^+$ $j^+$



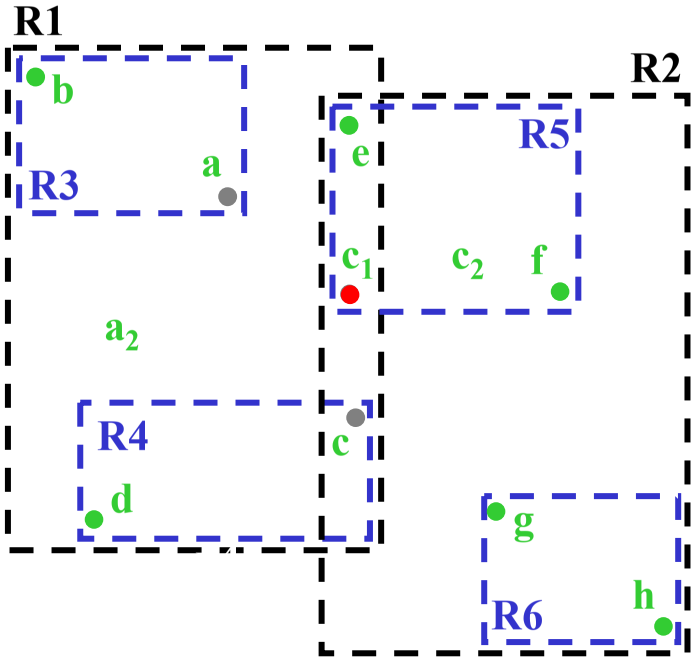
Operations: $c^-$ $c_1^+$ $a^-$ $a_1^+$



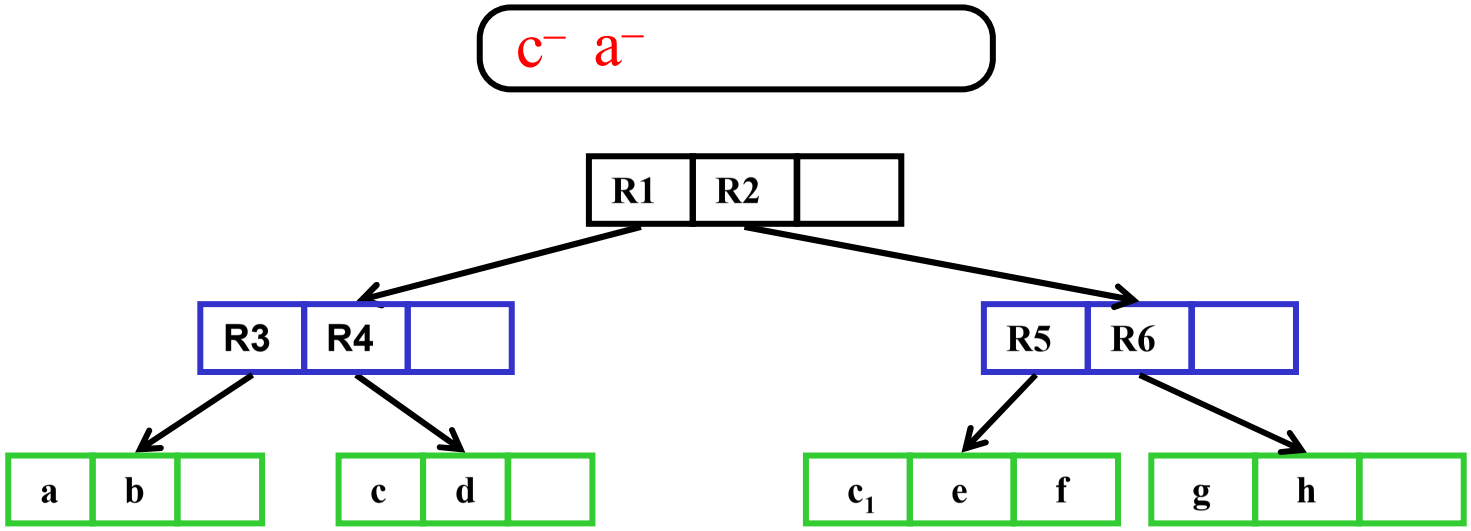

- **Structure of the $R^R$-tree:**
  - A regular disk-based R-tree
  - Main-memory *operations buffer,* storing at most $C_{max}$ pending insertions and deletions.
  - The operations buffer is an R-tree, with a *deletion flag* in leaf entries.



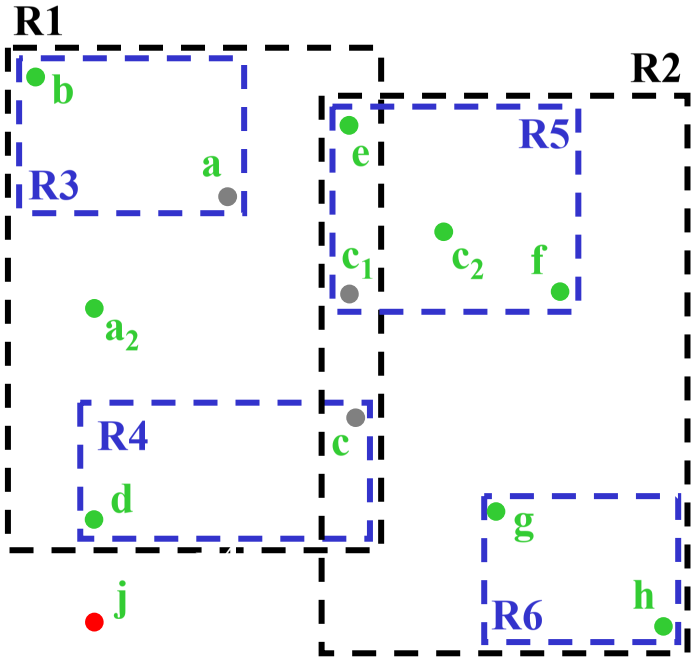
# Operations: $c^-$ $c_1^+$ $a^-$ $a_1^+$ $a_1^-$



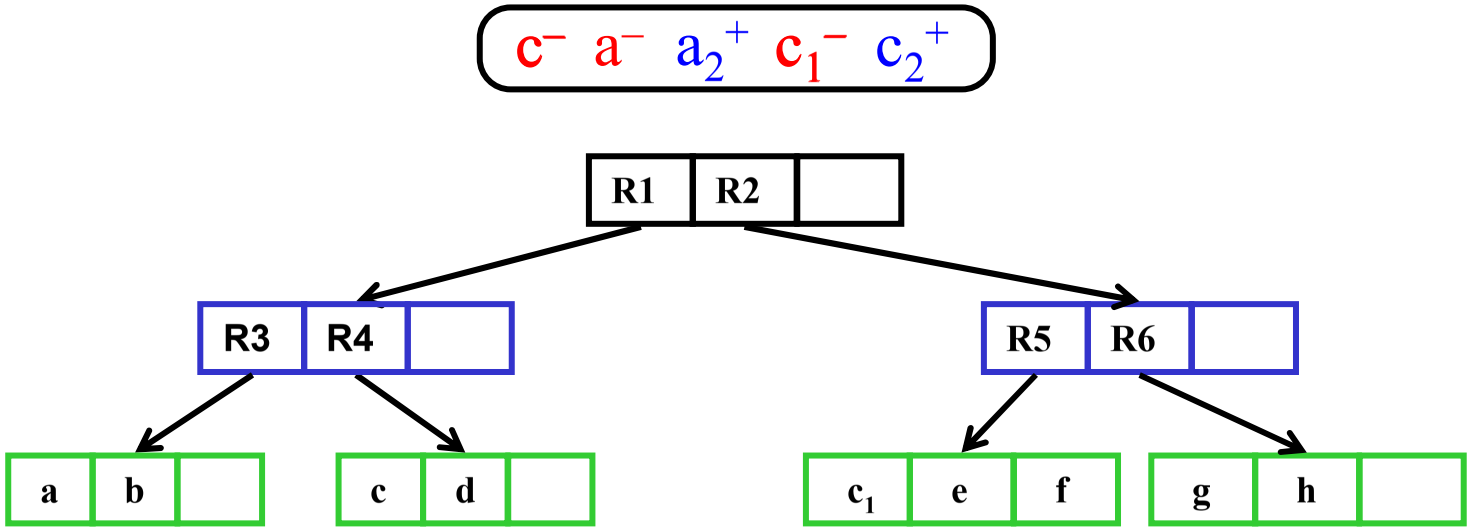

- **Insertions and deletions**
  - Insertions and deletions are put into operations buffer, if it is not full
  - Incoming deletion *annihilates* the matching insertion in the buffer (and vice versa).



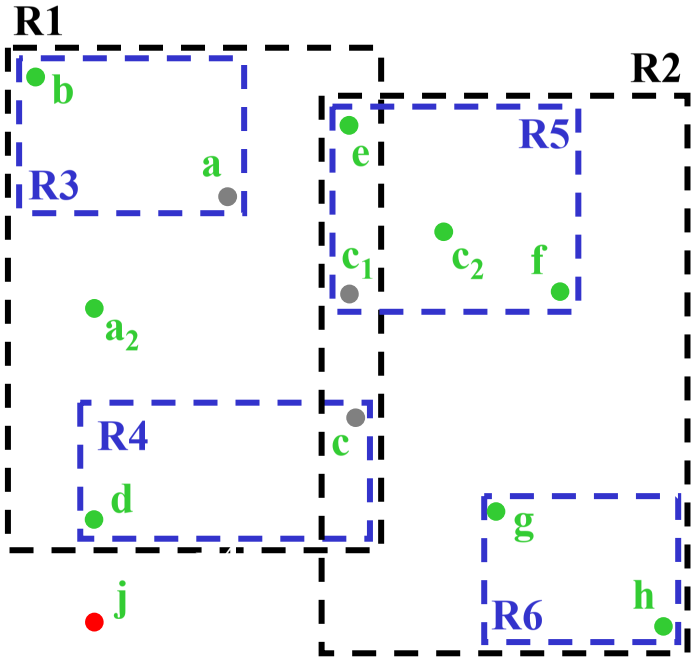
Operations: $c^-$ $c_1^+$ $a^-$ $a_1^+$ $a_1^-$ $a_2^+$ $c_1^-$ $c_2^+$ $j^+$



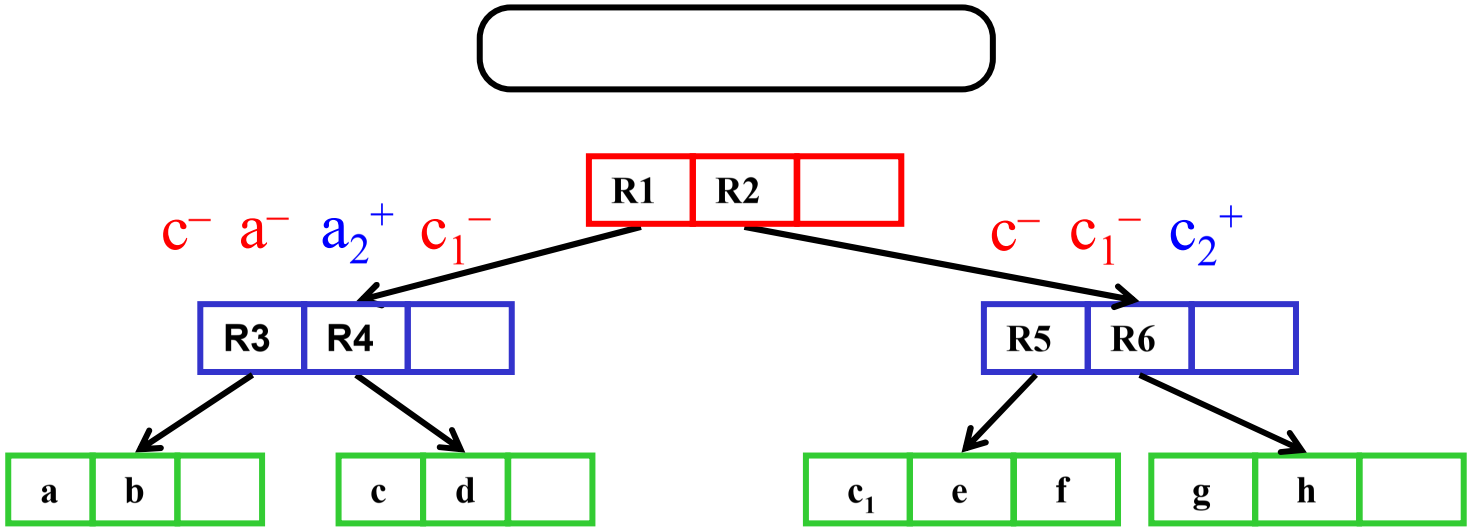

- **Partial buffer emptying**
  - Regular R-tree heuristics are used to direct insertions.
  - Copies of deletions are directed into the subtrees that overlap them.
  - Operations go down the tree in groups, sharing the I/O.



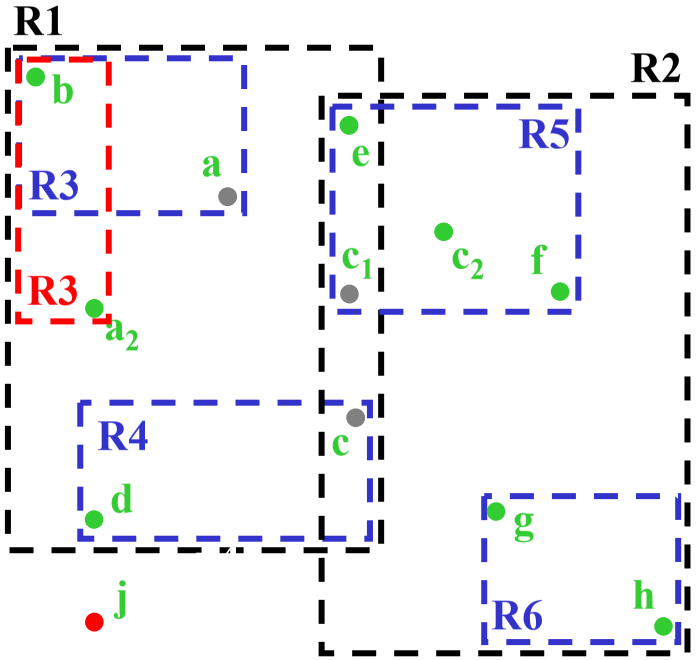
Operations: $c^-$ $c_1^+$ $a^-$ $a_1^+$ $a_1^-$ $a_2^+$ $c_1^-$ $c_2^+$ $j^+$



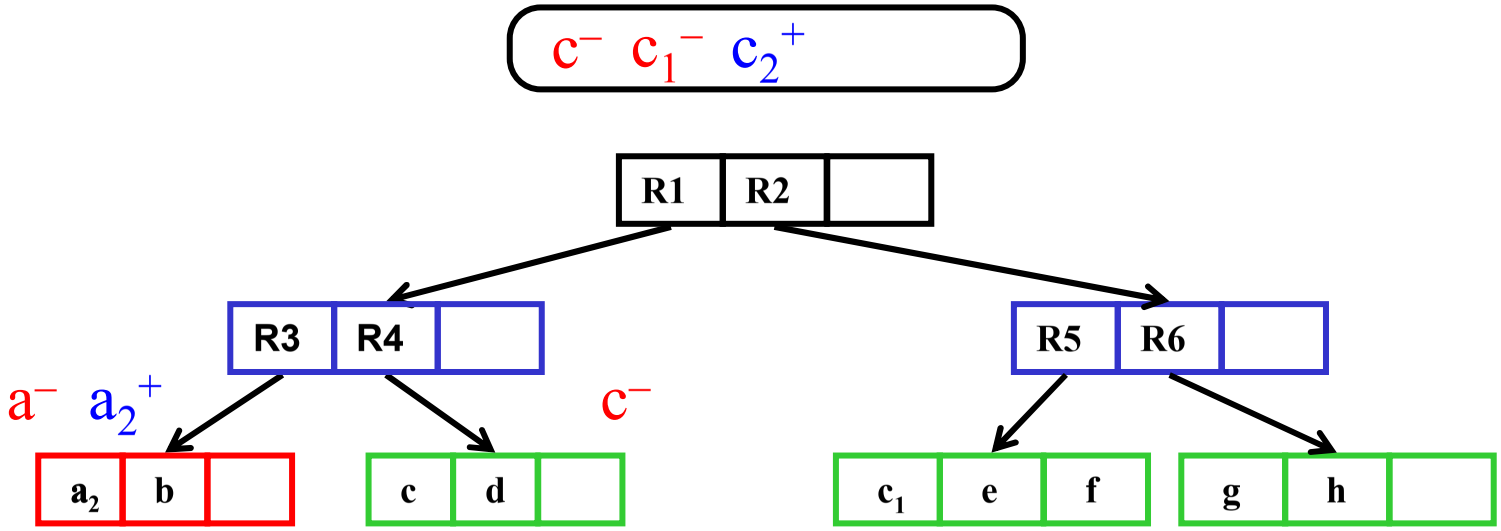

- **Partial buffer emptying**
  - Regular R-tree heuristics are used to direct insertions.
  - Copies of deletions are directed into the subtrees that overlap them.
  - Operations go down the tree in groups, sharing the I/O.
    - We avoid small groups – *at the root level*, all groups are put back to the buffer, except for the largest one.



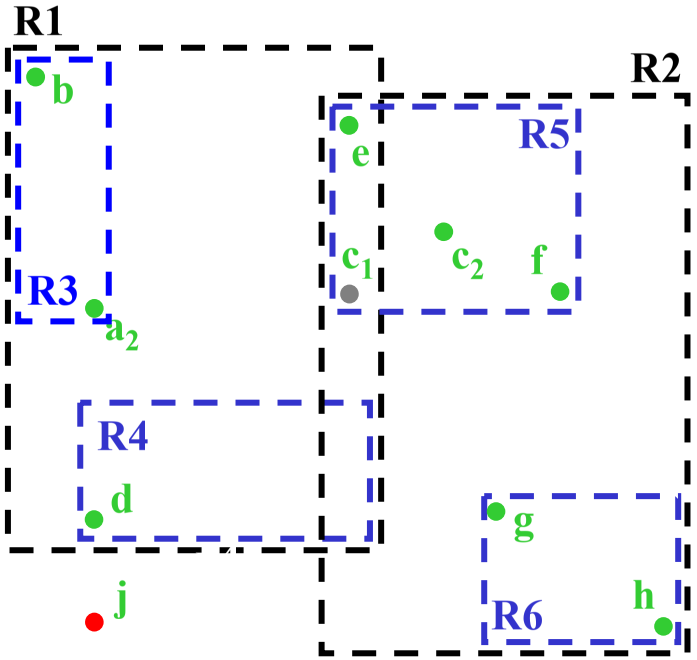
Operations: $c^-$ $c_1^+$ $a^-$ $a_1^+$ $a_1^-$ $a_2^+$ $c_1^-$ $c_2^+$ $j^+$



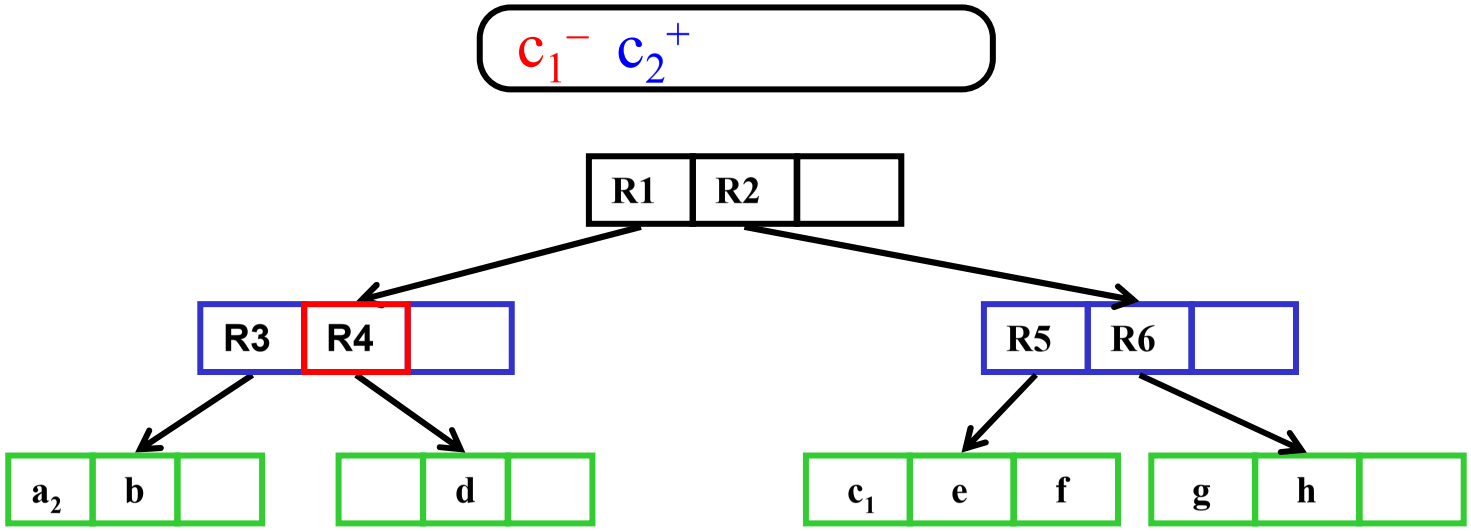

- **Partial buffer emptying**
  - When operations reach leaves they are performed.
  - A parent is written only when all its children are updated.
  - When a deletion is successfully performed, a corresponding entry from the buffer is removed.



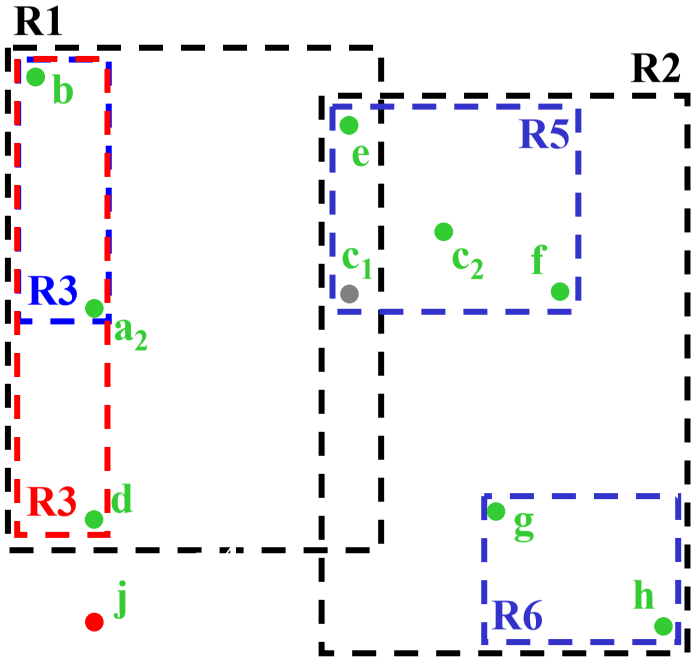
Operations: $c^-$ $c_1^+$ $a^-$ $a_1^+$ $a_1^-$ $a_2^+$ $c_1^-$ $c_2^+$ $j^+$



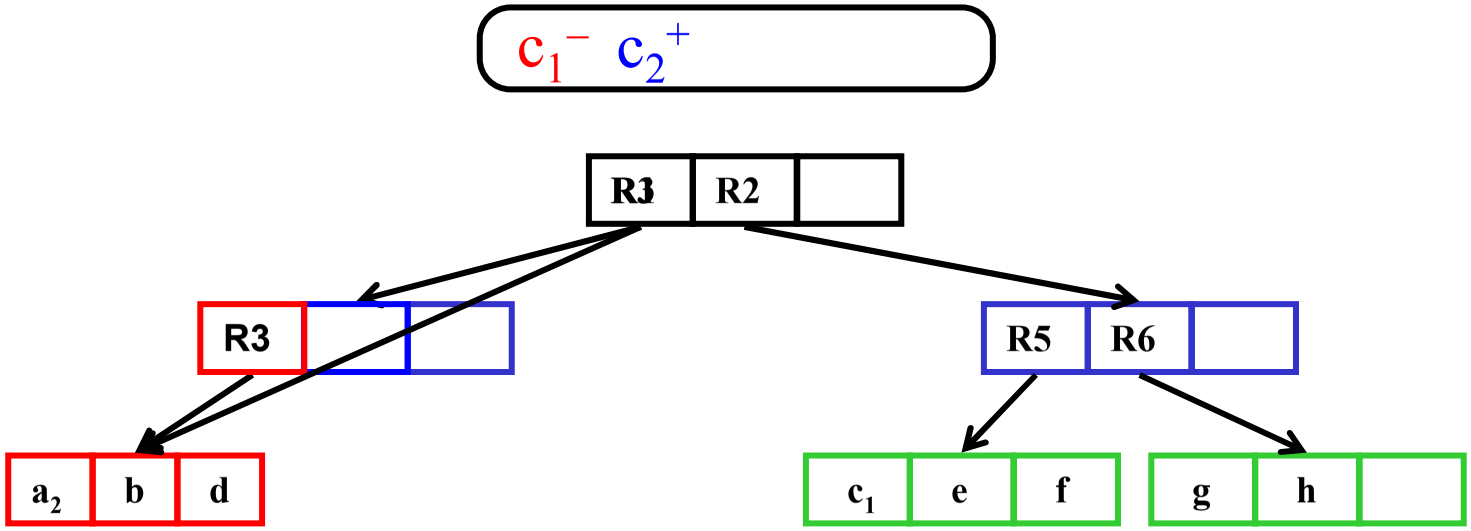

- **Bulk restructuring of the tree**
  - A child subtree may become
    - like *R4* – with an underfull subtree root: *merge with a sibling (single-entry non-leaf root is removed)*



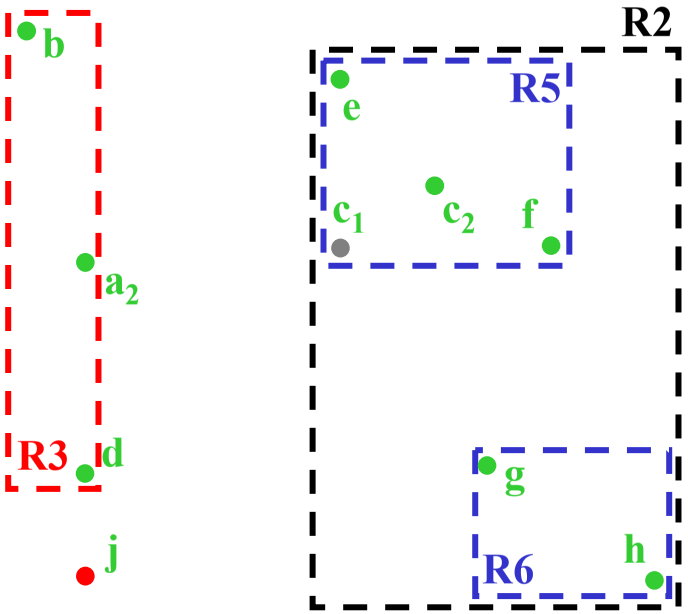
Operations: $c^-$ $c_1^+$ $a^-$ $a_1^+$ $a_1^-$ $a_2^+$ $c_1^-$ $c_2^+$ $j^+$



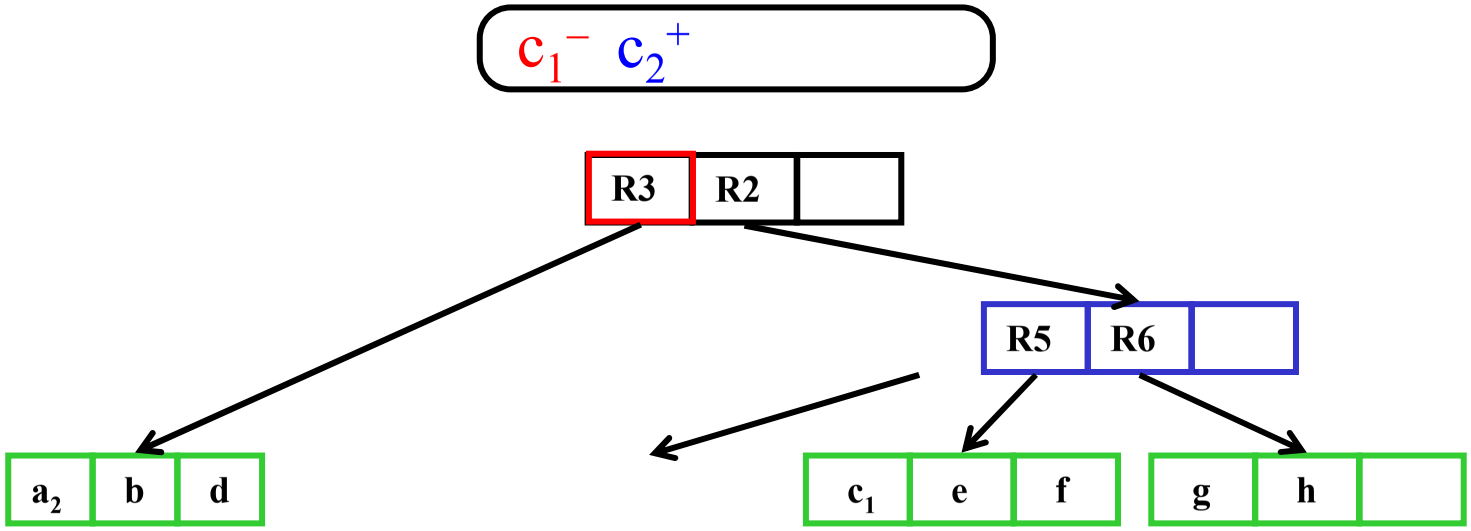

- **Bulk restructuring of the tree**
  - A child subtree may become
    - like *R4* – with an underfull subtree root: *merge with a sibling (single-entry non-leaf root is removed)*



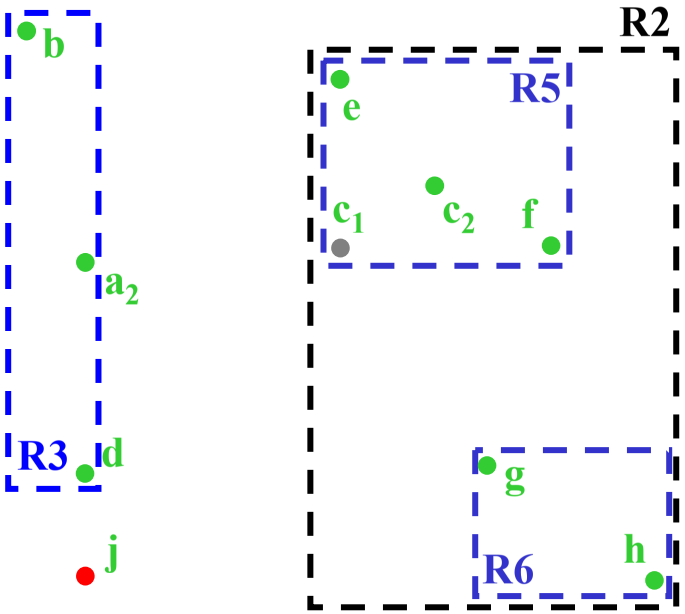
# Operations: $c^-$ $c_1^+$ $a^-$ $a_1^+$ $a_1^-$ $a_2^+$ $c_1^-$ $c_2^+$ $j^+$



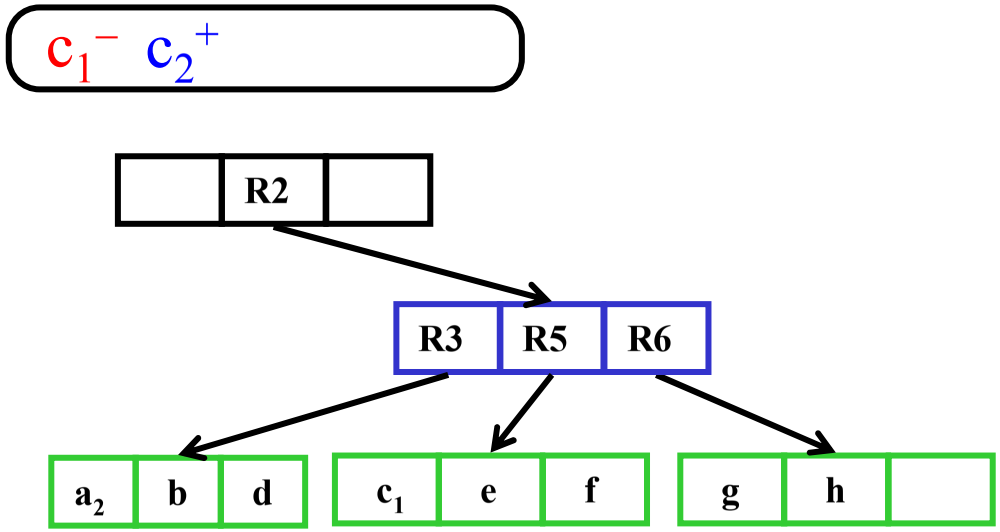

- **Bulk restructuring of the tree**
  - A child subtree may become
    - like *R4* – with an underfull subtree root: *merge with a sibling (single-entry non-leaf root is removed),*
    - like *R3* – of a smaller height: *traverse down the tree using R-tree heuristics and insert at the right level.*



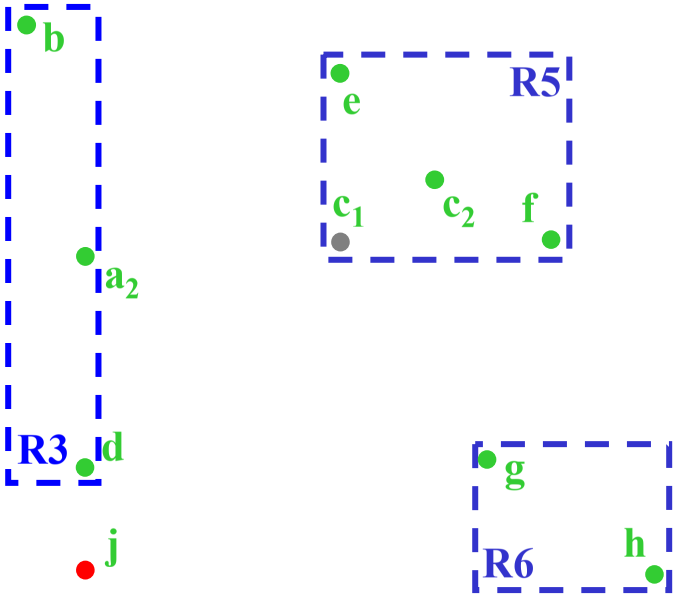
# Operations: $c^-$ $c_1^+$ $a^-$ $a_1^+$ $a_1^-$ $a_2^+$ $c_1^-$ $c_2^+$ $j^+$




- **Bulk restructuring of the tree**
  - A child subtree may become
    - like *R4* – with an underfull subtree root: *merge with a sibling (single-entry non-leaf root is removed),*
    - like *R3* – of a smaller height: *traverse down the tree using R-tree heuristics and insert at the right level*.
    - of a smaller height and with an underfull subtree root: *merge with a sibling at the right level*

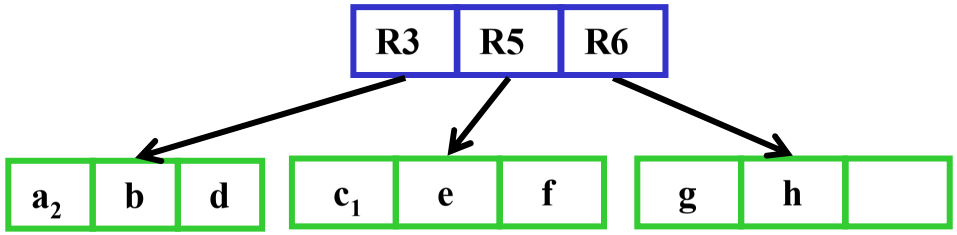


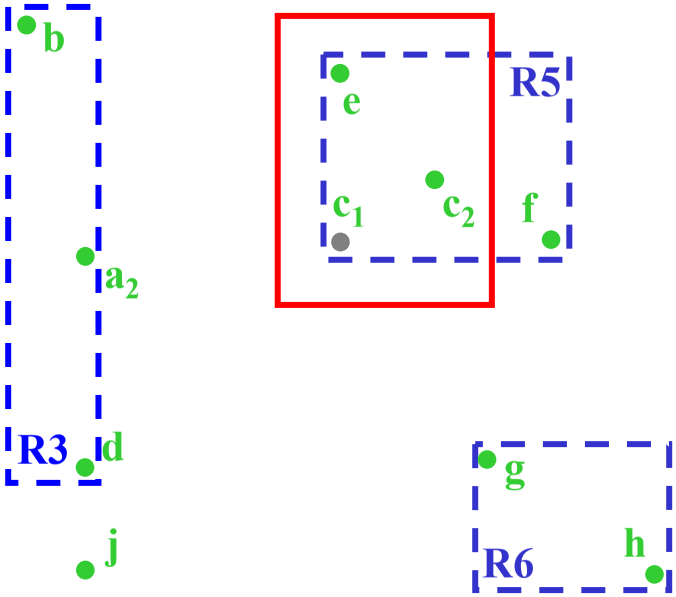
Operations: $c^-$ $c_1^+$ $a^-$ $a_1^+$ $a_1^-$ $a_2^+$ $c_1^-$ $c_2^+$ $j^+$




- **Bulk restructuring of the tree**
  - Overfull node may need to be split multiple times
- Finally put $j^+$ into the buffer!

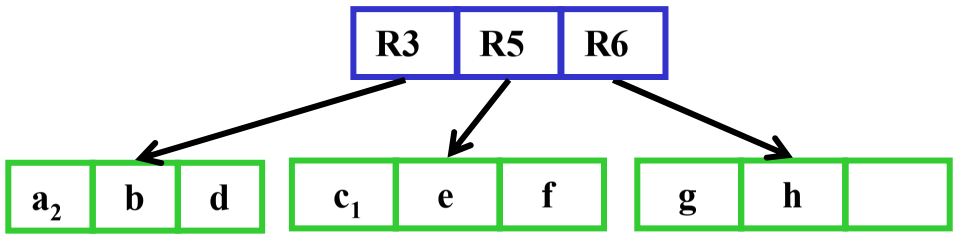
$c_1^-$ $c_2^+$






- **Querying**
  - First query the disk R-tree
  - Next, query the operations buffer
  - Finally, add the results and annihilate matching entries
  - Result: e $c_2$

$c_1^-$ $c_2^+$ $j^+$

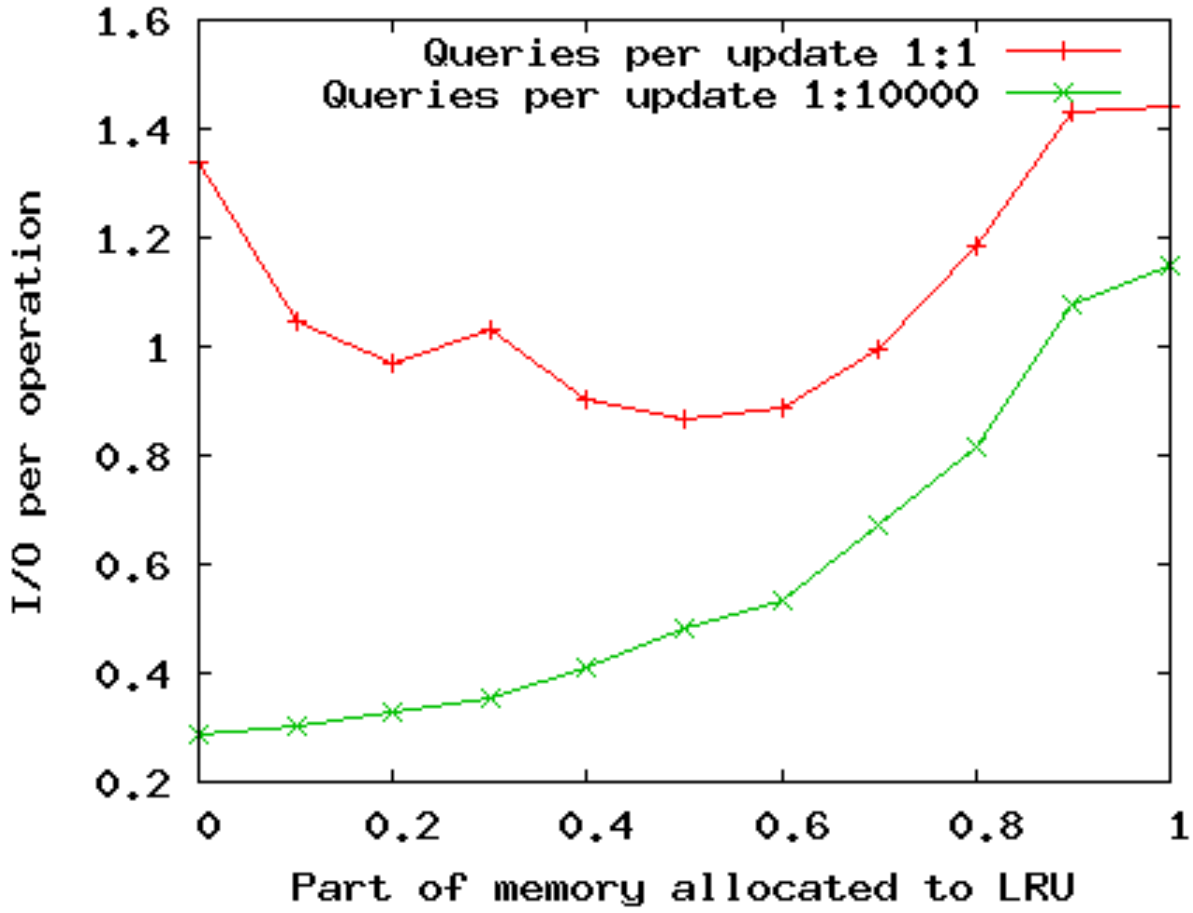


# Empirical study: setup

- Setting
  - Simulation of 100,000 cars in a road network
  - Updates occur whenever they travel 200 meters
- Comparisons
  - The $R^R$-tree (with the *largest-group* BufferEmpty strategy)
  - The RUM-tree (R-tree with update memos) with a write-back LRU cache
  - The R-tree with a write-back LRU cache
- Each structure is given exactly the same amount of main-memory

# Empirical study

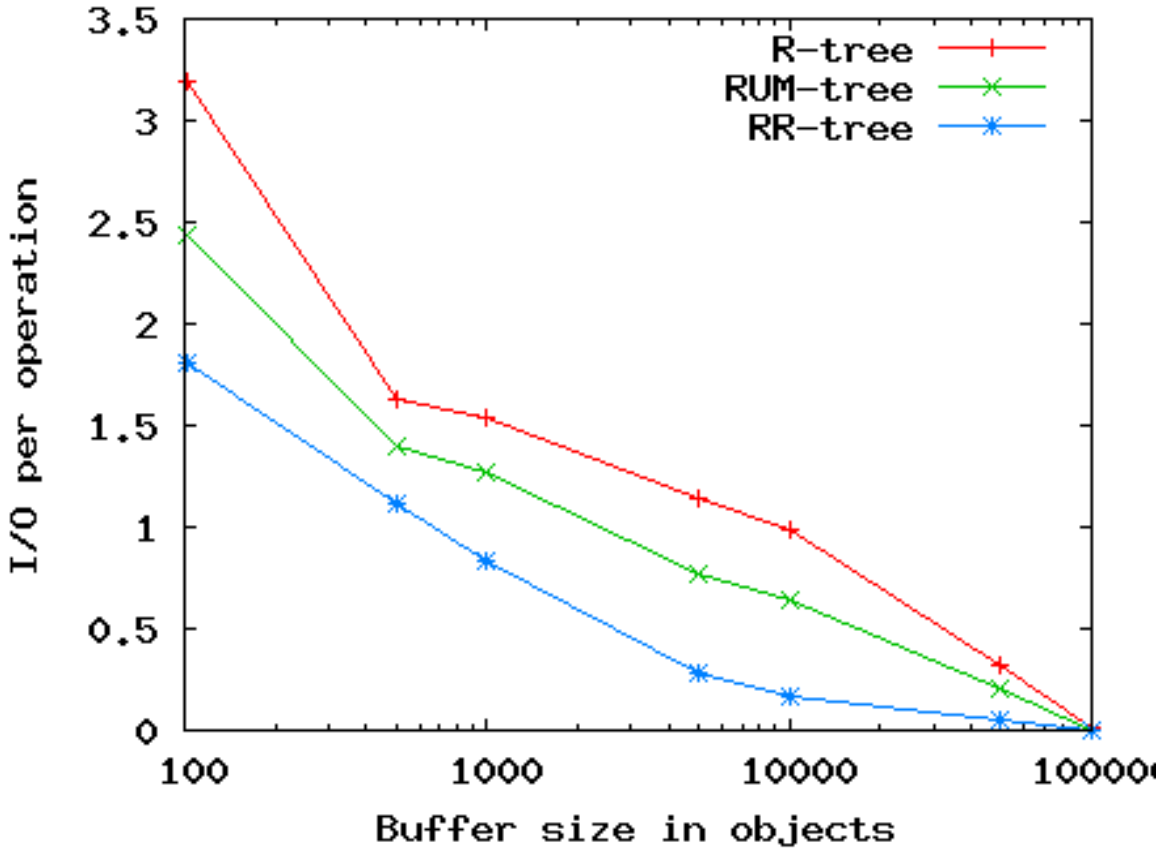
- How much main memory should be used for LRU buffering?




- For dynamic workloads, no LRU buffering should be used.

# Empirical study

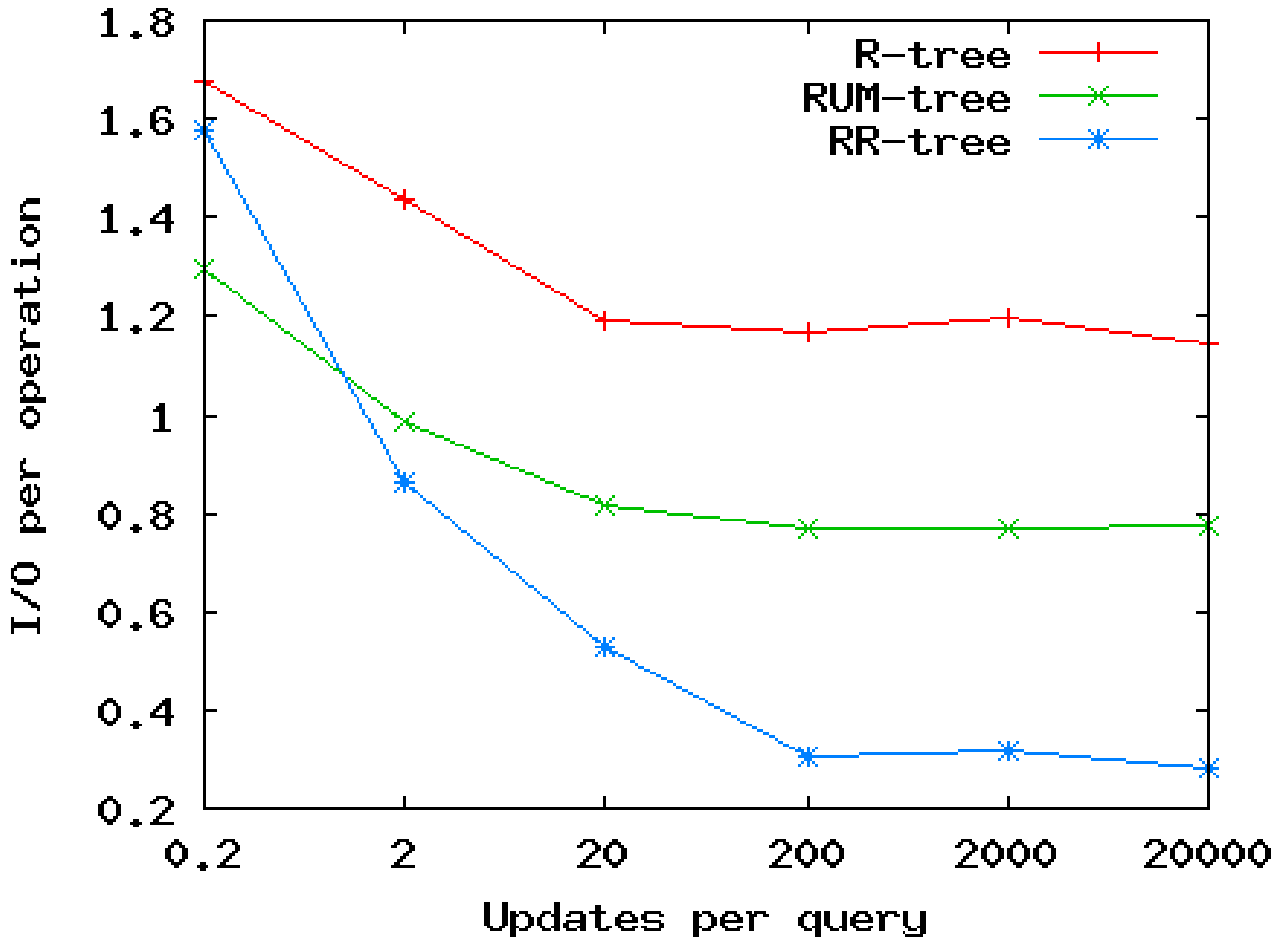
- How do the structures benefit from different amounts of main memory?




- Ex: memory is 10% of data set size: $R^R$-tree is more than 4 times faster than the RUM-tree and 7 times faster than the R-tree

# Empirical study

- How do the structures perform for varying ratios of updates and queries?



# Summary

- Presented a new main-memory buffering technique for R-tree type indices
- The general idea is to speed up updates by allowing these to share I/O.
- Uses partial buffer emptying
- Empirical studies show that the proposal improves on existing proposals.
- See the paper for the analytical study

- Future work
  - Application to other types of indices
  - Better main-memory indexing
  - Exploring the query performance/update performance trade-off
  - New recovery techniques