Temporal Indexing

MVBT

Temporal Indexing

- Transaction time databases : update the last version, query all versions
- Queries:
  - “Find all employees that worked in the company on 09/12/98”
  - “Find the employees with salary between 35K and 50K on 04/21/99”
- Multiversion B-tree: answers efficiently both queries

Basics

- A data structure is called :
  - Ephemeral: updates create a new version and the old version cannot be queried
  - Persistent: updates can be applied at any version and any version can be queried
  - Partially Persistent: updates applied to the last version and any version can be queried
- Transaction time fits with partially persistence

MVBT – The idea

- Store all versions of the state of a B+-tree which evolved over time, i.e. multiple “snapshots” of the tree
- Inserts, updates and deletes are applied to the present version of the tree and increase the version number of the whole tree
- Queries know which version of the tree they require the result(s) from

General method

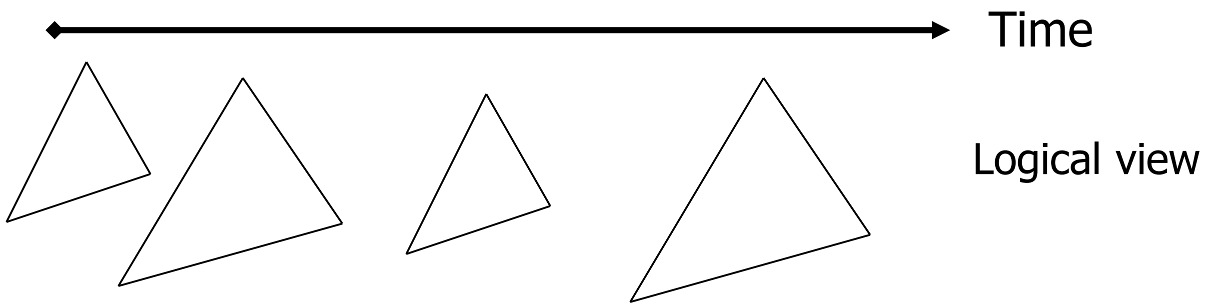
- Transform a single version external access structure (with high utilization of disk blocks) at the cost of a constant factor in time and space requirements compared to the original, single version structure
- Such increase is asymptotically optimal = worst-case bounds cannot decrease by adding multiversion capability to existing structure

Proposition Specifics

- Extend B+-tree to have multiversion capability
- Support operations:
  - Insert(key, info)
    Insert into current version, increase tree version
  - Delete(key)
    Delete from current version, increase tree version
  - Exact match query(key, version)
  - Range query(lowkey, highkey, version)

Overview

- The multiversion B-tree is a directed acyclic graph (DAG) of B-tree nodes that results from incremental changes to original B-tree
- It has a number of B+-tree root nodes which partition the versions from the first to the present one so that each B-tree root stands for an interval of versions



Blocks (pages)

- Contain b data items
- Live if it has not been copied, dead otherwise
- Weak version condition: for every version i and each block A, except the roots of versions, we require that the number of “alive” entries in A is either 0 or at least d, where $b=k\cdot d$ ($d=b/k$)

Data Items

- Leaf node of the tree
  - <key, in_version, del_version, info>
- Inner node of the tree
  - <router, in_version, del_version, info>
- Said to be of version **i** if its lifespan contains **i**
- In live block, deletion version * denotes that this entry has not been deleted at present, in a dead block it means that the entry has not yet been deleted before the block died

Updates

- Each update creates new version
- If no structural changes:
  - Insert: lifespan is [**i**, *****)
  - Delete: changes del_version from * to **i**
- A structural change is required if:
  - Block overflow: can only fit b entries in a block
  - Weak version underflow: if deletion in a block with exactly d current version entries

Structural Modification

- Copy the block and remove all but the present version entries from the copy
- If block consists of primarily present version entries, the copy will produce an almost full block, resulting in a split again after a few subsequent insertions
  - To avoid this, request that at least $\varepsilon d+1$ insert or delete operations are necessary for the next block overflow or version underflow in that block (ε will be defined later)

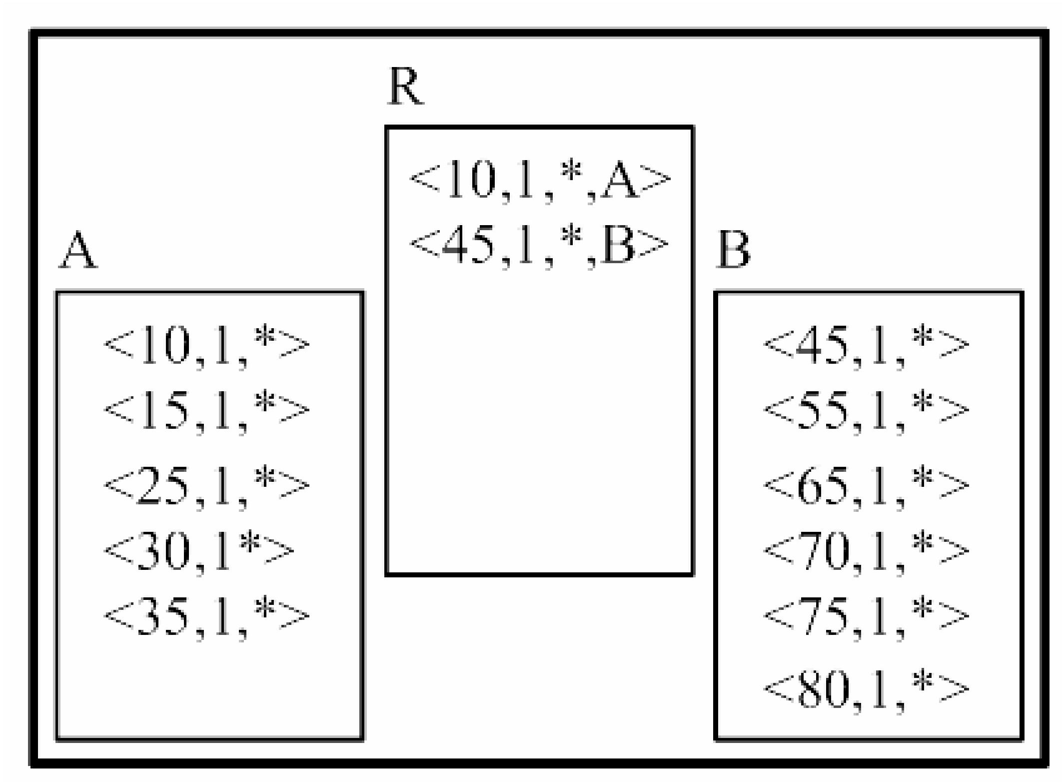
Strong Version Constraints

- Strong version condition: the number of present version entries after a version split must be in range from $(1+\varepsilon)d$ to $(k-\varepsilon)d$.
- Strong version underflow: result of version split leading to less than $(1+\varepsilon)d$ entries
  - Attempt to merge with a sibling block containing only its present version entries, if necessary followed by a version independent split by key values
- Strong version overflow: if a version split leads to more than $(k-\varepsilon)d$ entries in the block
  - Also perform a split by key values

Simple Example

b=6
d=2
ε=0.5

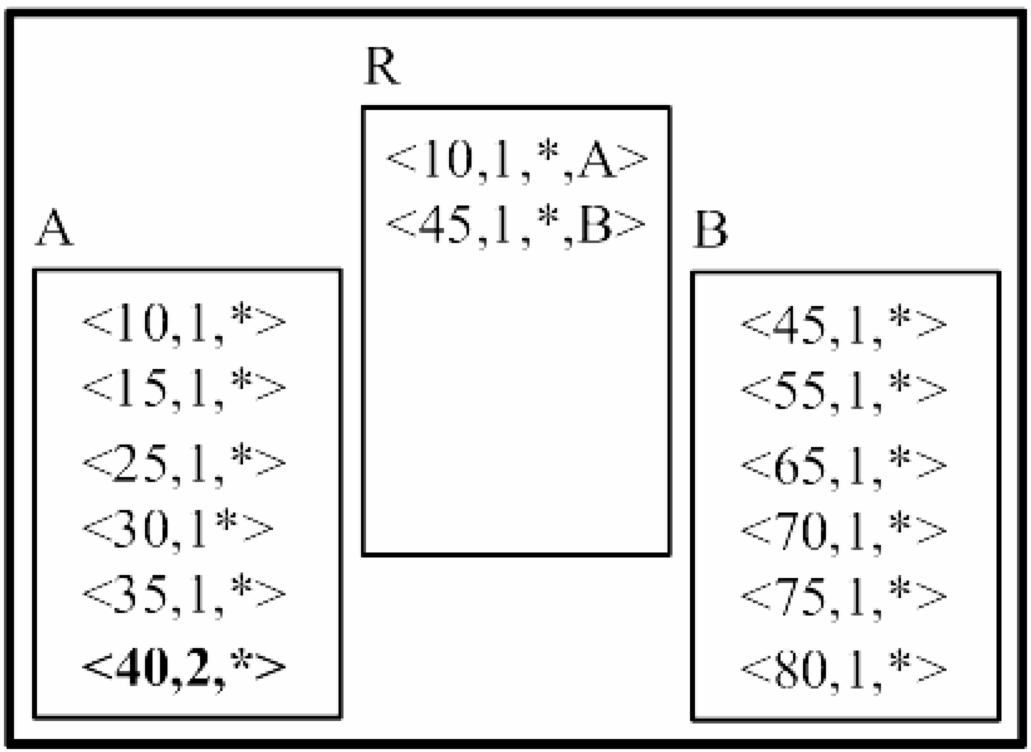
Original Tree (Version 1)



Insert(40)

Simple Example

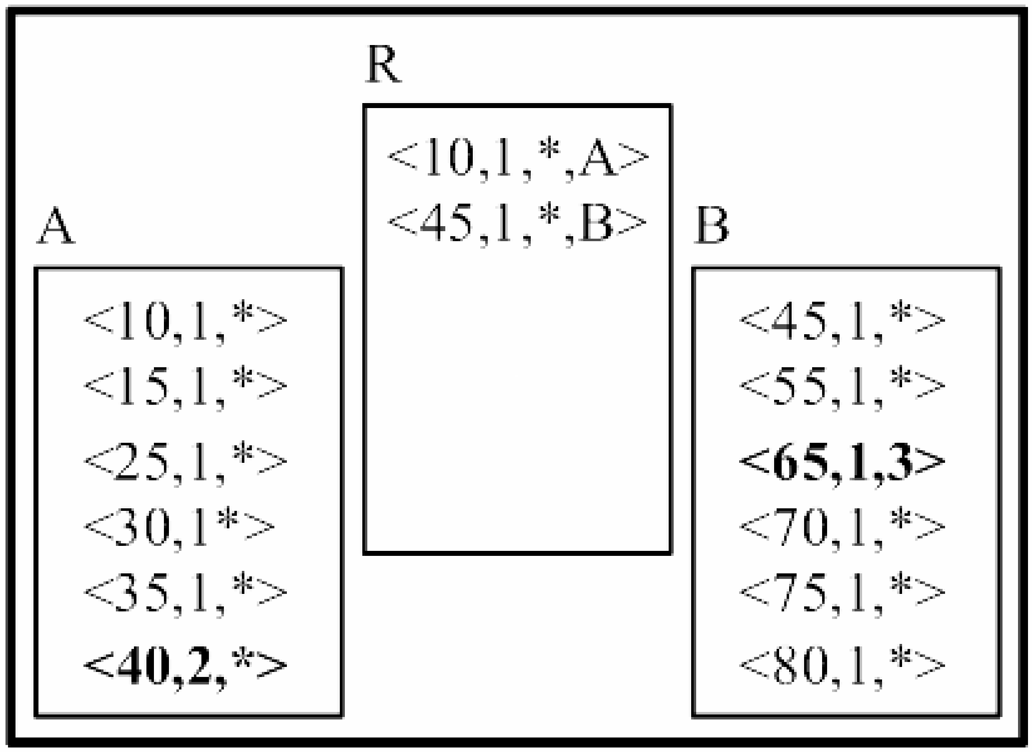
(Version 2)



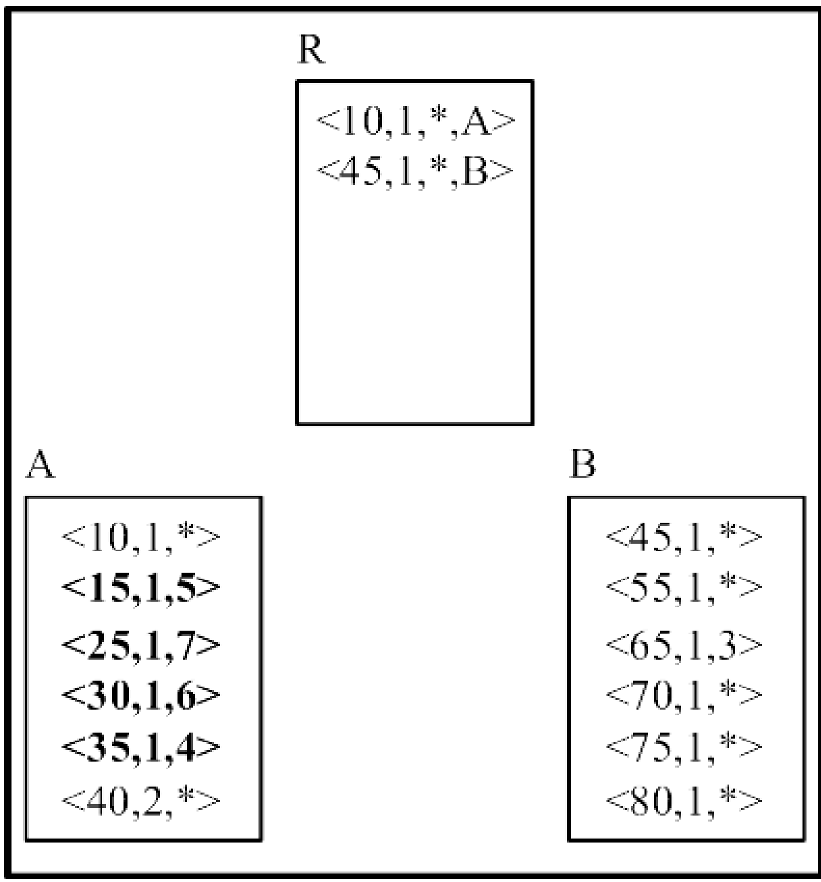
Delete(65)

Simple Example

(Version 3)



Version Split Example



(Version 7)

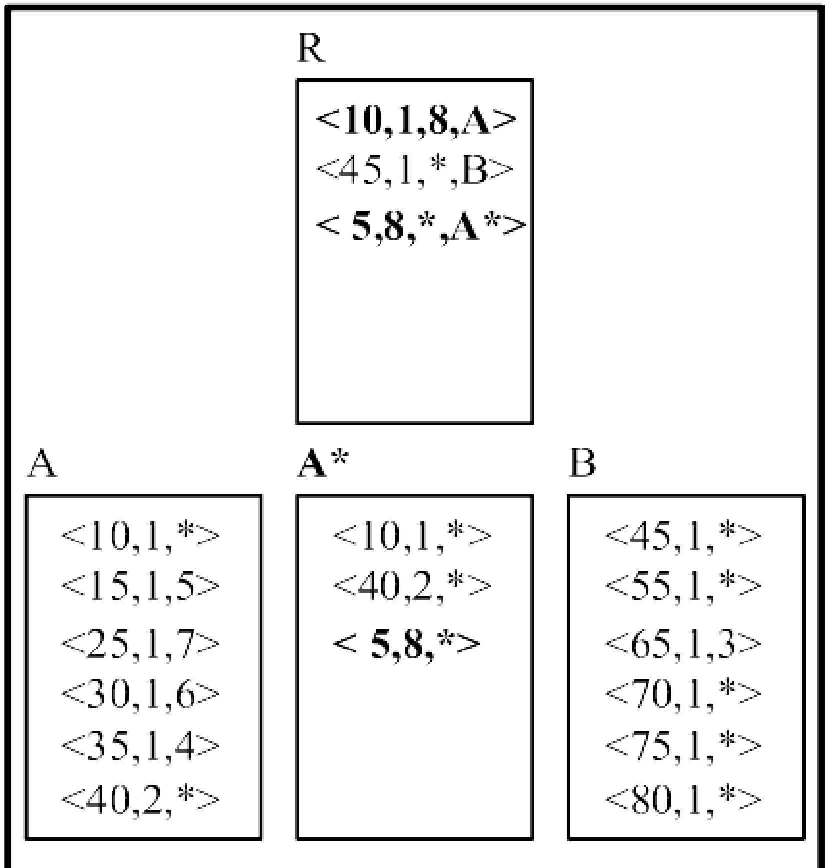
Insert(5) creates a block overflow

All currently live entries copied to the new live block A*, old block A marked dead

Also, the root block is updated to show that entity 10 was alive in the dead block A until version 8

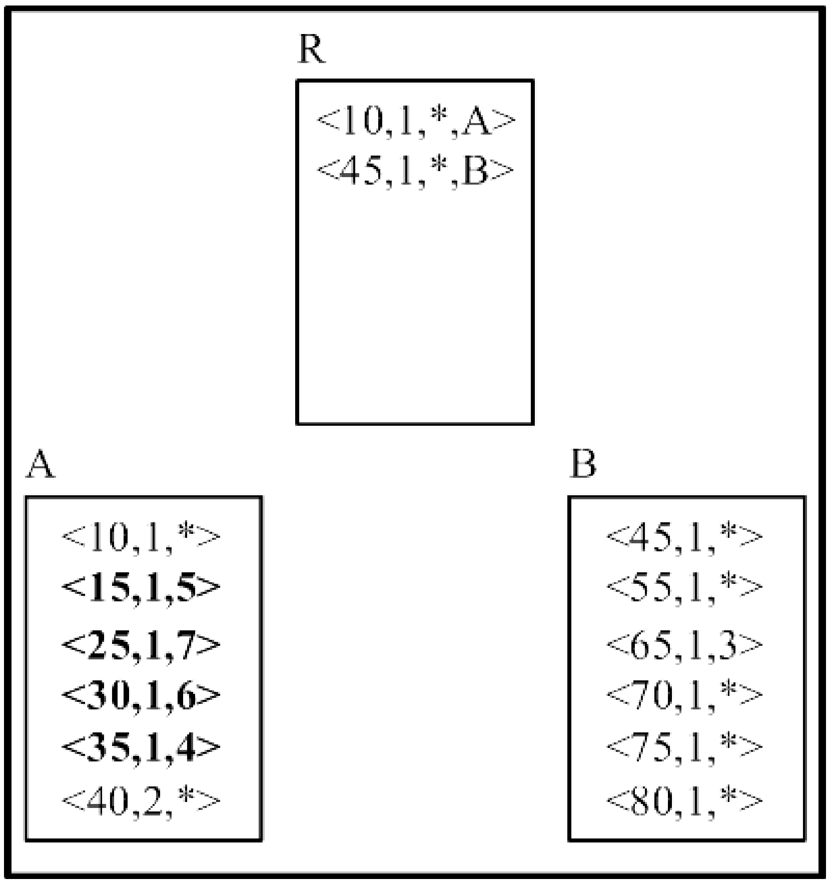
Version Split Example

Resulting “tree”



(Version 8)

Weak V. Underflow Example



(Version 7)

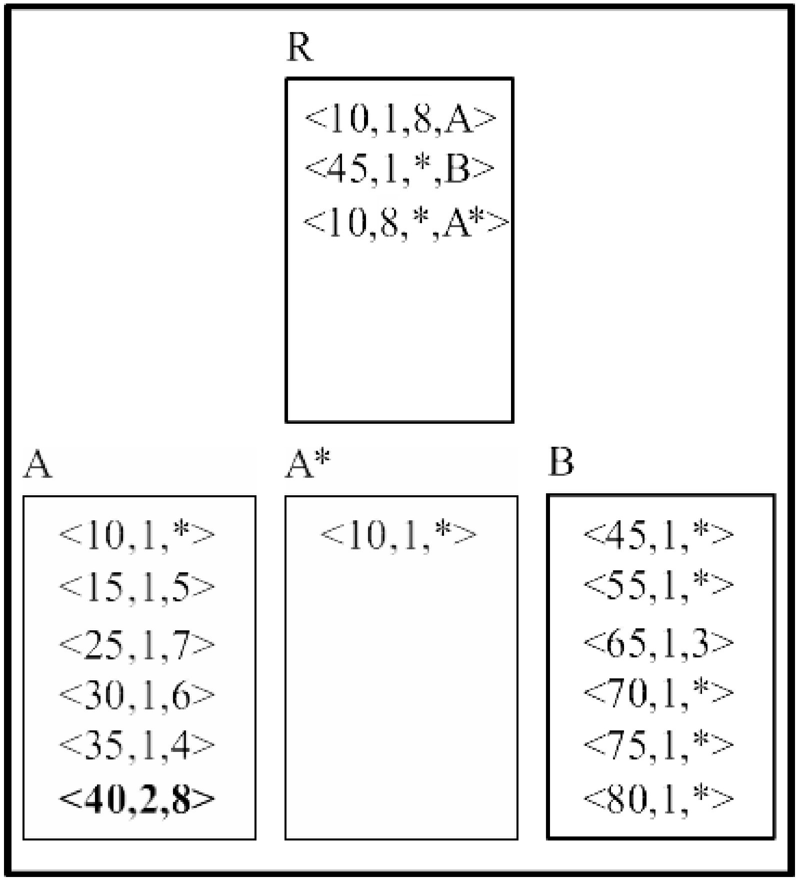
Suppose b=6, d=2, ε=0.5

(d: the minimum # of current version entries in the block)

Delete(40) results in block A only having 1 current entry

1<d: weak underflow, split A

Strong V. Underflow Example

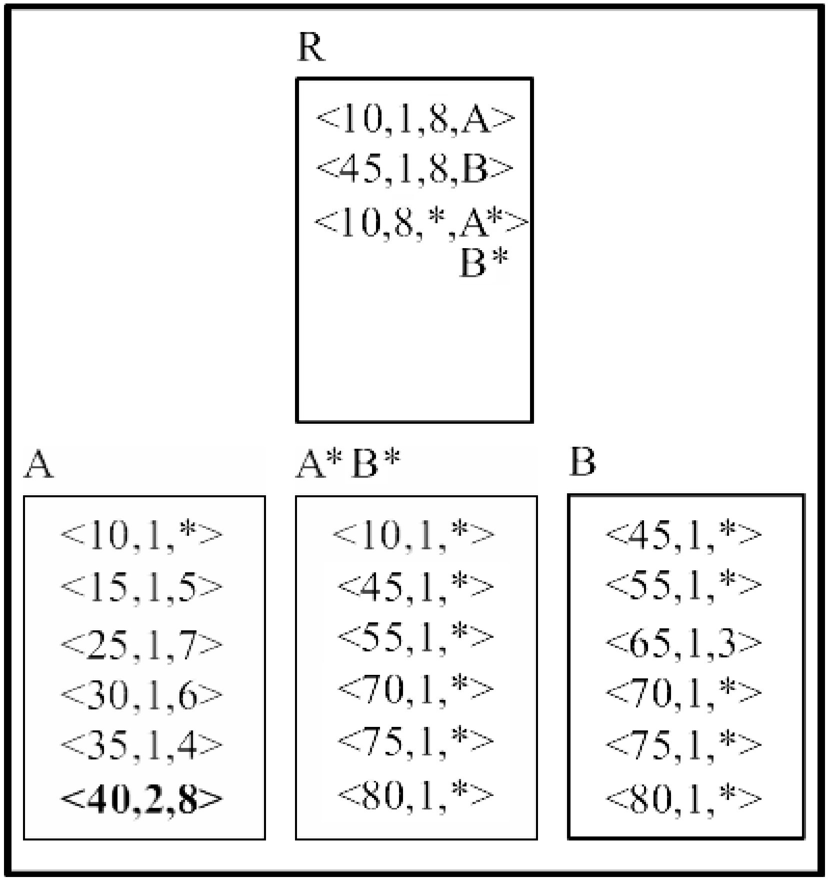


(Processing: Version 7→8)

The version split of A has led to less than (1+ε)d=1.5*2=3 entries in the new node →strong version underflow

Seek a sibling of A* (in our case, B)
Version split it (to create B*)
Merge B* and A*
to produce block A*B*

Strong V. Underflow Condition Violation

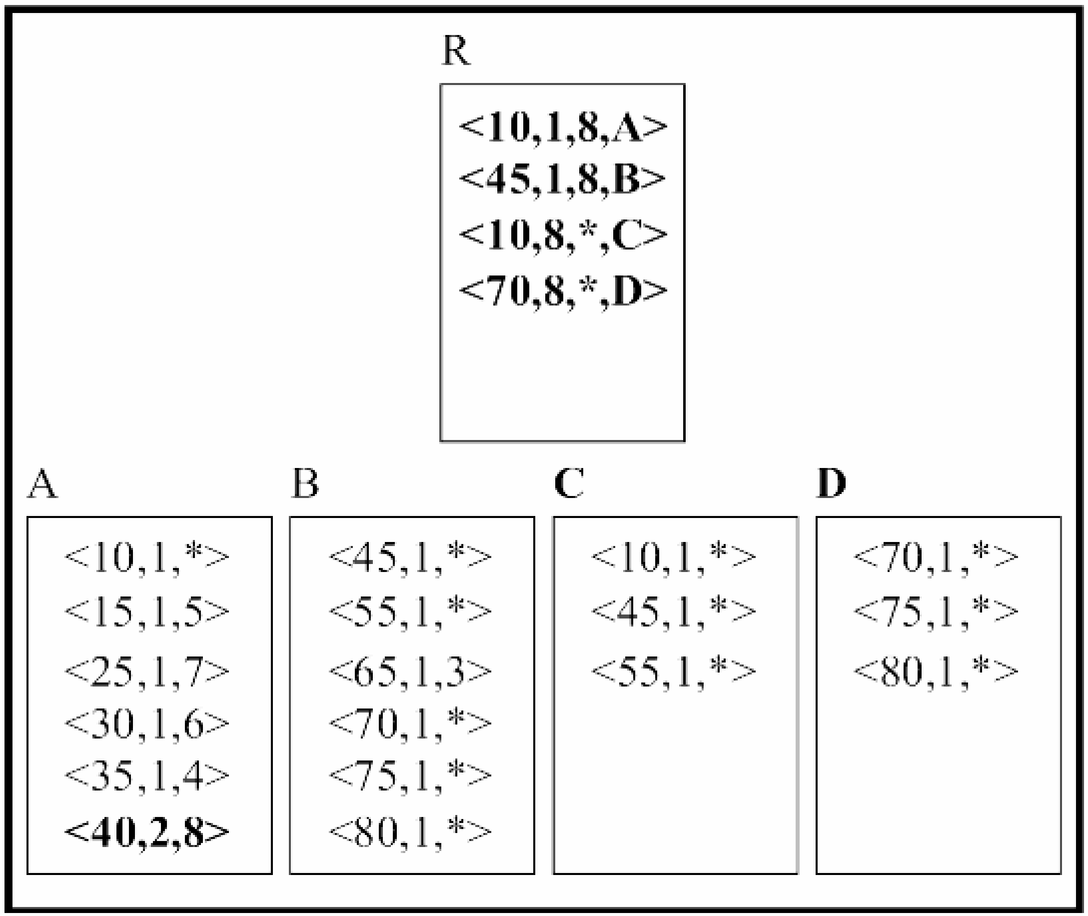


(Processing: Version 7→8)

But now, the node A*B* violates the strong ersion overflow condition and must be split by key into nodes C and D

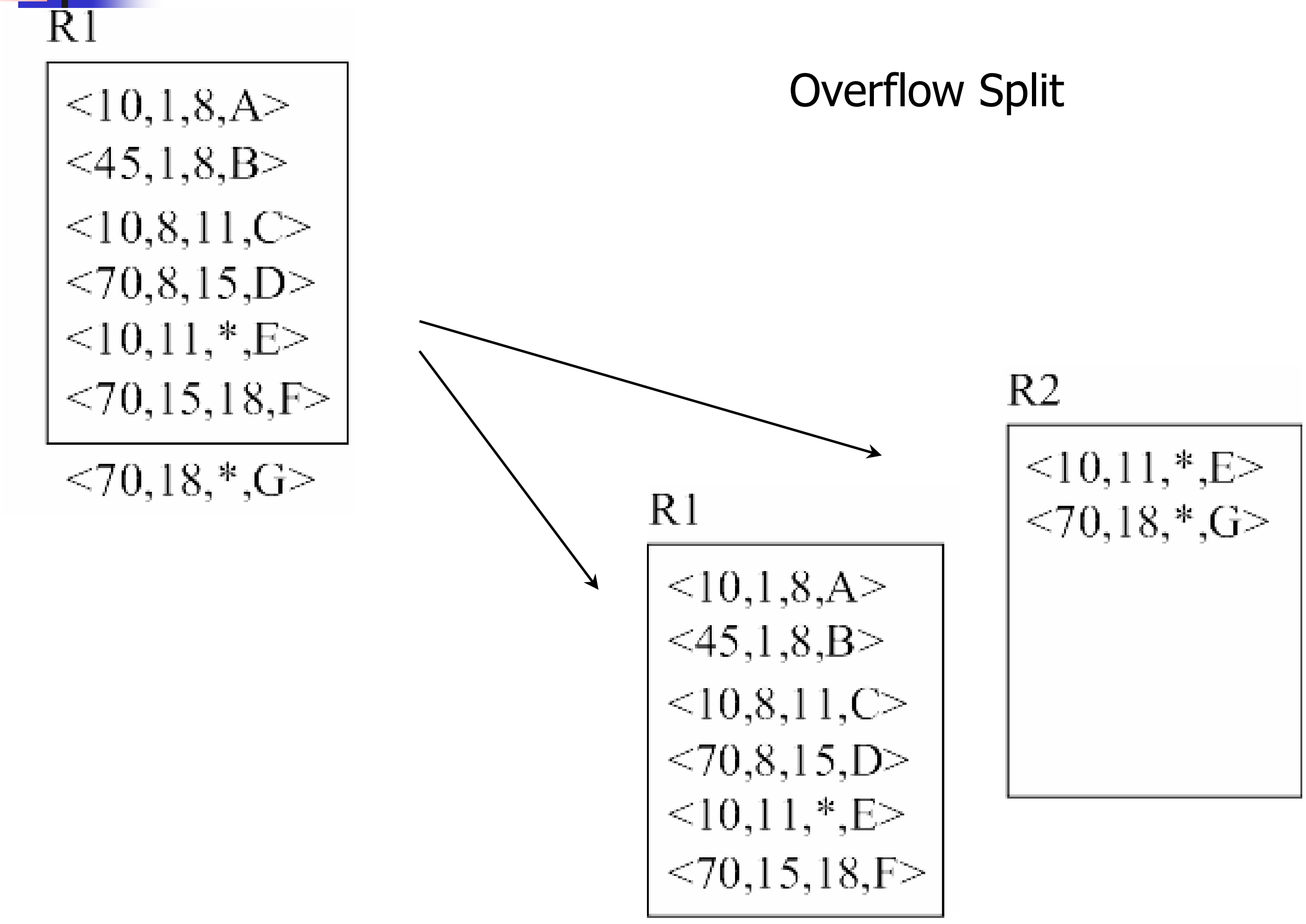
Resulting Tree

Example Query: (25, 5)



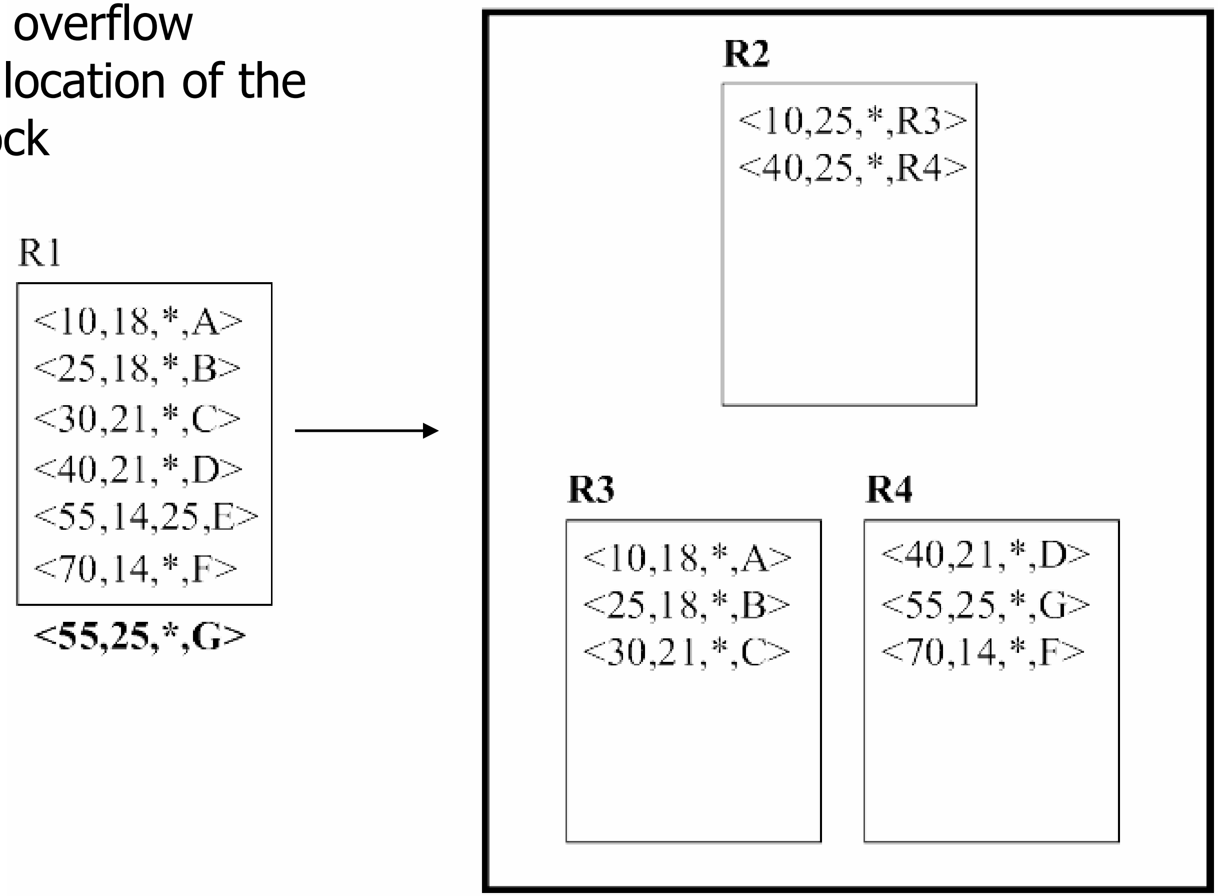
(Version 8)

Roots Can Split, Too

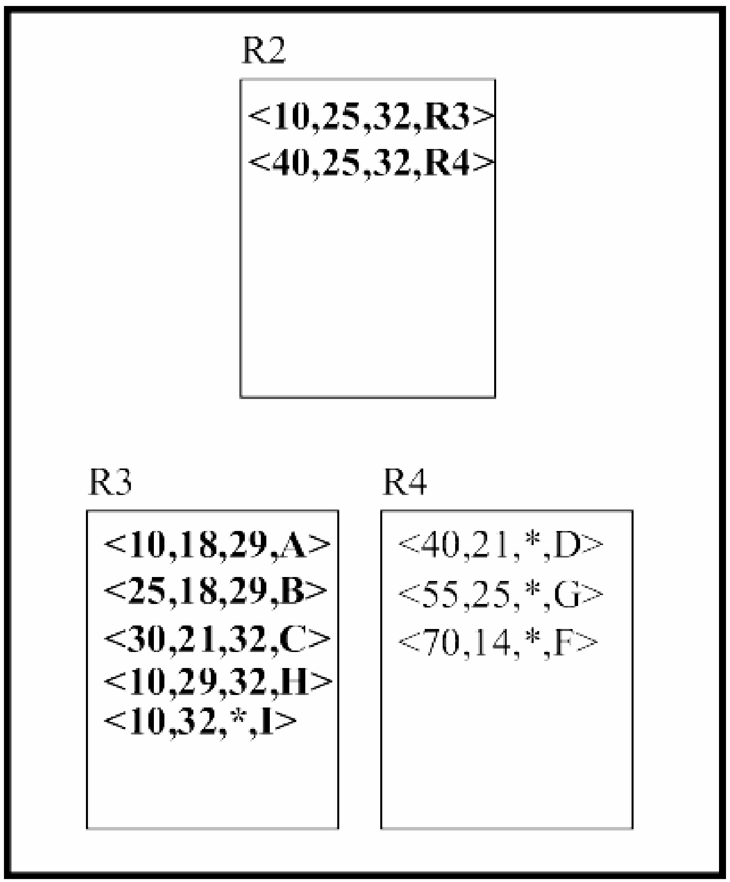


Roots Can Split, Too

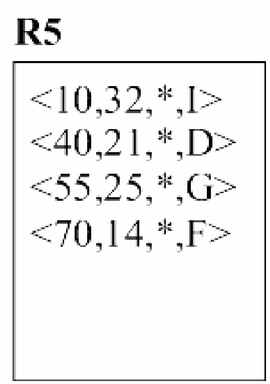
Strong version overflow
with key split and allocation of the
new block



Weak Version Underflow of the Root Node



R3 has shrunk→weak underflow
Block copies of R3 and R4 are created and merged into R5



This causes weak version underflow of R2, so R5 becomes new root block

Algorithms

- Insertion: Find the leaf node for the new key e, then call blockInsert (say A)
- blockInsert: enter e in A
  - If block overflow of A then
    - Version split, block insert
    - If strong version underflow then
      - Merge
    - Else if strong version overflow → key split

Algorithms

- Delete: blockDelete A
  - Check weak version underflow on A
  - If true, then merge with sibling
- Note that Deletion is easier than the insertion in the MVBT. What about the B+-tree?

Constraints on MVBT Parameters

- What are the restrictions on choices of k and ε?
  - $(k-\varepsilon)d+1 \geq (1/\alpha)(1+\varepsilon)d$
    - α is the fraction of the entries in a block guaranteed to be in a new node after a key split, 0.5 for B-trees
    - Before key split, A contains at least $(k-\varepsilon)d+1$ entries
    - After a key split, both blocks must contain at least $(1+\varepsilon)d$ entries.
  - $2d-1 \geq (1+\varepsilon)d$
    - Before merge is performed, together there are at least $2d-1$ present version entries in the blocks to be merged

Efficiency Analysis

- The big result is that the MVBT is asymptotically optimal to the B-Tree in the worst case in time and space for all considered operations
- Search time is in

$$O(\lceil \log_b m_i \rceil + \frac{r}{d})$$

- Space is O(n/b) and update $5\lceil \log_b m_i \rceil$

Additional Issues

- Can store the roots of version interval trees in a B-tree of their own; authors demonstrated that for most practical cases the depth of the root B-tree will never exceed two anyway.

Other Approach

- Overlapping B+-trees
- Idea: for each update store only the path from the root node to the leaf node, use the rest of the tree for the new version (since it is the same)
- Better approach, store only the new path
- Easier to implement but space is higher:
- $O(n/b \log_b n/b)$

Temporal Hashing

- Hashing is used to answer exact match queries
- Exact-match with time: “find if employee with id=23 was working in the company on 09/08/98”
- One approach: Use MVBT (4-5 I/Os)
- Temporal Hashing: can achieve 2 I/Os

Temporal Hashing

- Treat objects that fall into a given bucket B_i during the whole evolution as an <u>evolving set</u> $B_i(t)$
- To find if key k is in S(t), reconstruct bucket B_i and search for key k (B_i is the bucket that the key k should have been placed at t)
- Simply observe how an ephemeral hashing scheme would map the keys of S(t) in buckets and keep the history of each bucket

Temporal Hashing

- For each bucket use a Snapshot Index
- Using the SI we can reconstruct the bucket
- Note that when we split a bucket we treat the keys that moved to the new bucket as deleted for the old bucket, newly inserted for the new
- Another approach is to keep an evolving list

Bi-temporal Data Indexing

- Bi-temporal Index:
  - R-tree for valid time
  - Partial persistence for transaction time
- Partially persistent R-tree [Kumar at al. 97]
- What about the case that valid time has an open end_time (now)?
  - A special R-tree to handle that [Saltenis et al. ‘99]