Seminar ”Cloud Computing“

Dr. Katja Hose, Dr. Klaus Berberich, Jörg Schad

# *“Efficient B-tree Based Indexing for Cloud Data Processing“*

S. Wu et al., VLDB '10

*presented by Frederic Raber*

UNIVERSITÄT
DES
SAARLANDES

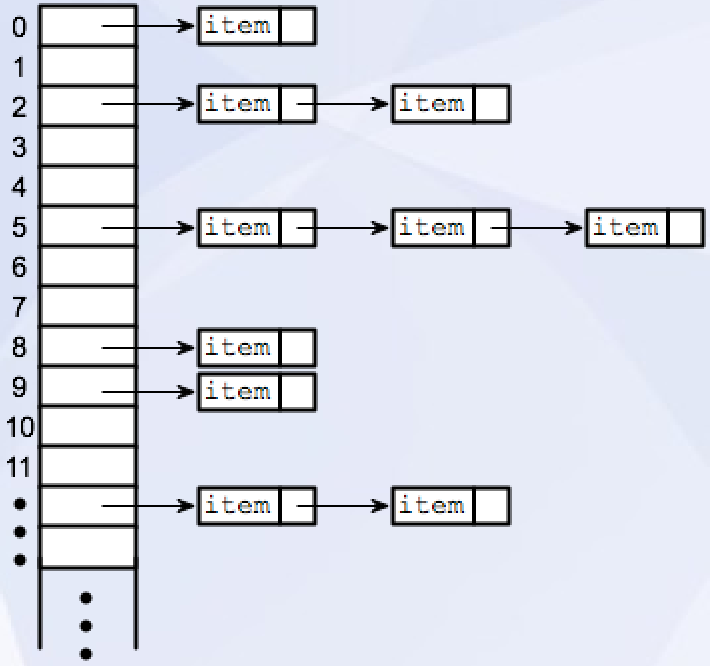
# Motivation: Indexing

- For quickly retrieving data, we need an Index
- Example: Youtube

  → find Video by its ID

# Motivation: Solutions

- Mostly used solution: Hash-Table (Key-Value-Pairs)
- In Youtube example: Map from ID to movie file



# Motivation: Problems

- Problem: Often need secondary index
- Youtube videos are mostly not searched by ID, but by name or author

→ need to generate secondary index

# Motivation: Secondary indices

- Solution: Generate the secondary index by a map-reduce job
- Execute this as a batch job repeatedly after a certain interval

→ high overhead for recreating the index

→ updates are not propagated directly, only after index recreation

# Motivation: Secondary indices

- Better: create a global B-Tree for the data on a server

→ Index is updated directly

→ No overhead for batching a map-reduce job

→ Central server is a bottleneck, high risk of failure

→ Not scalable

# Motivation: What we need

- So we need a solution which is
  - Providing instant updates
  - Fast
  - Highly scalable
  - Fault tolerant

# A new approach

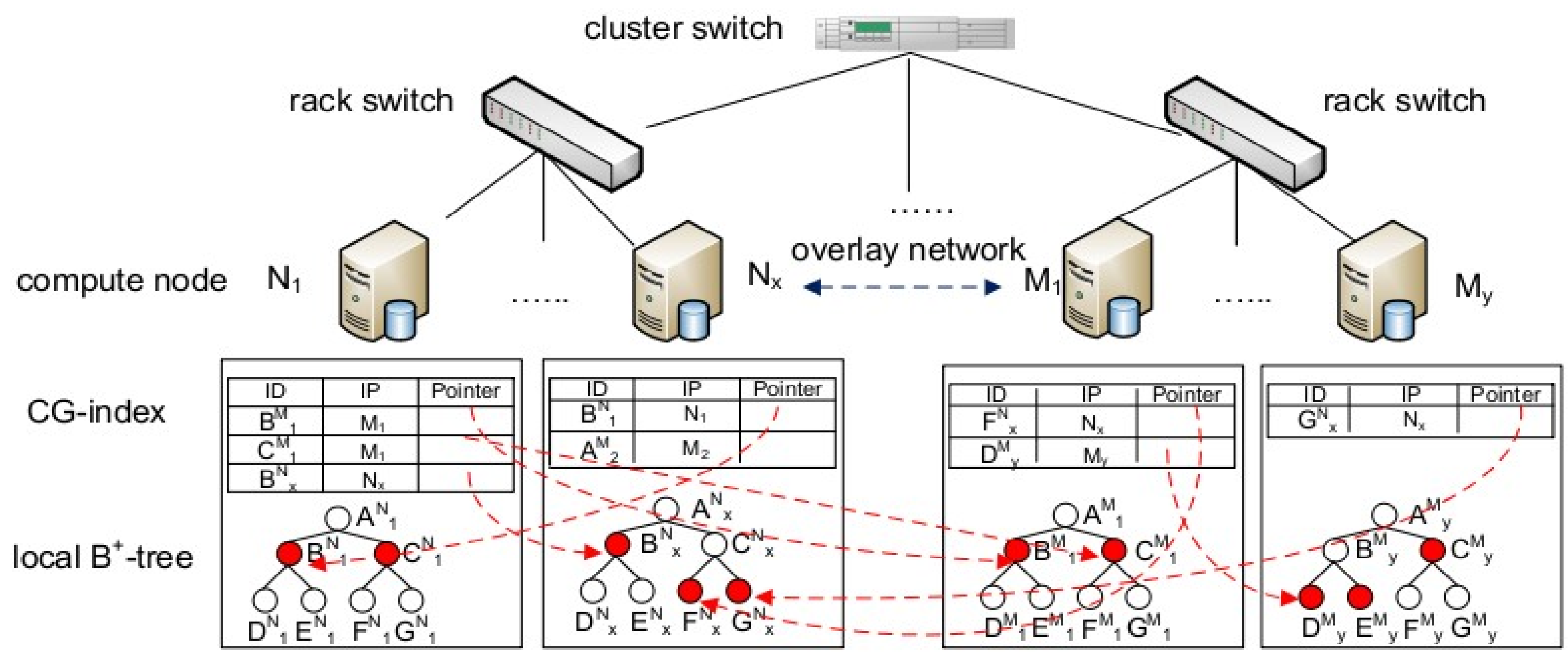
- „Index over the index“
- B+-tree on on intra-node level

➔ Global index (CG-index) for these local indices

  ➔ Clustered through the compute nodes

  ➔ Routing by BATON overlay protocol (last talk)

# A new approach



| ID | IP | Pointer |
|---|---|---|
| $B^M_1$ | $M_1$ | |
| $C^M_1$ | $M_1$ | |
| $B^N_x$ | $N_x$ | |

| ID | IP | Pointer |
|---|---|---|
| $B^N_1$ | $N_1$ | |
| $A^M_2$ | $M_2$ | |

| ID | IP | Pointer |
|---|---|---|
| $F^N_x$ | $N_x$ | |
| $D^M_y$ | $M_y$ | |

| ID | IP | Pointer |
|---|---|---|
| $G^N_x$ | $N_x$ | |

(a) System Architecture

# Open questions

1. Which local tree-nodes should be in the global index?
2. How is a B+tree-node indexed in the CG-index?
3. How is the data retrieved?
4. How are updates performed?
5. How is data consistency assured?

# Outline

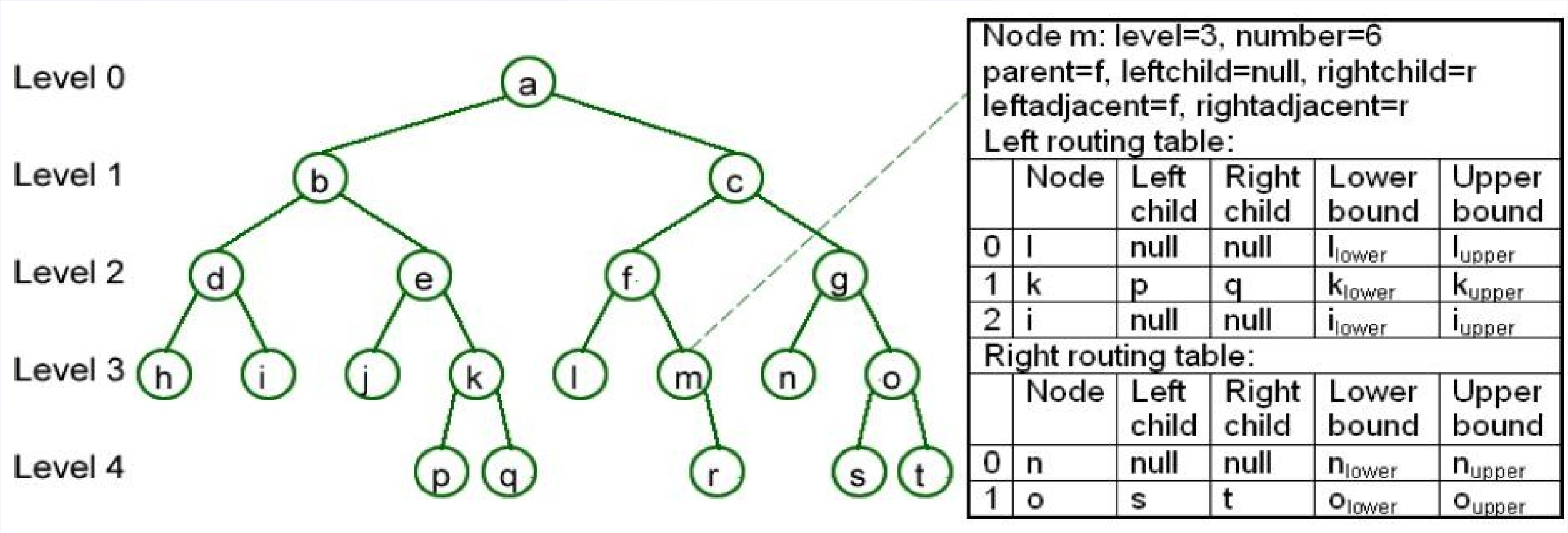
- Motivation
- **Solutions: creation & usage**
- Solutions: maintenance
- Tuning
- Evaluation
- Conclusion

# BATON

- **BA**lanced **T**ree **O**ver-lay **N**etwork
- Distrubuted tree structure for dynamic P2P-systems
- Based on B-tree
- Self-balancing, like AVL-tree
- Designed for handling dynamic node join and departure
- In this paper only used for routing purposes

# BATON

- Each tree node corresponds to a network node
- Additional links on each node to:
  - Adjacent nodes in-order
  - Nodes in the same routing level



Node m: level=3, number=6
parent=f, leftchild=null, rightchild=r
leftadjacent=f, rightadjacent=r

Left routing table:

| | Node | Left child | Right child | Lower bound | Upper bound |
|---|---|---|---|---|---|
| 0 | l | null | null | $l_{lower}$ | $l_{upper}$ |
| 1 | k | p | q | $k_{lower}$ | $k_{upper}$ |
| 2 | i | null | null | $i_{lower}$ | $i_{upper}$ |

Right routing table:

| | Node | Left child | Right child | Lower bound | Upper bound |
|---|---|---|---|---|---|
| 0 | n | null | null | $n_{lower}$ | $n_{upper}$ |
| 1 | o | s | t | $o_{lower}$ | $o_{upper}$ |

# Index selection

1. Which local tree-nodes should be in the global index?

All of them?

→ No, would take too much space

→ Select only some

# Index selection

1. Which local tree-nodes should be in the global index?

- Don't index root or leaf nodes
- If a node is indexed,
  its direct children must not be indexed
- Only index nodes, if benefit is greater
  than maintenance cost

# Index selection algorithm

## *Expand*

1. Start with the root node as actual node
2. Compute if it is beneficial to index the child nodes
3. If yes, index them and remove actual node from the index. Goto 2

# Index selection algorithm (2)

*Collapse*

1. Check if (maintenance cost for indexing child nodes > benefit)

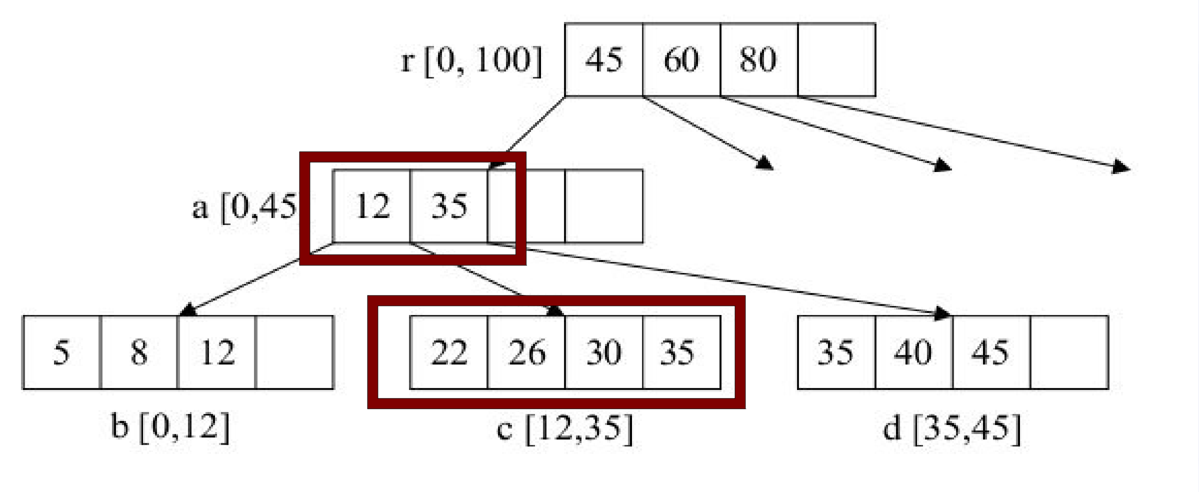
2. If yes, remove their index and index the parent node

# Index creation

*2. How is a B+tree-node indexed in the CG-index?*

1. Compute the key range for the node

→ look up in the parent node

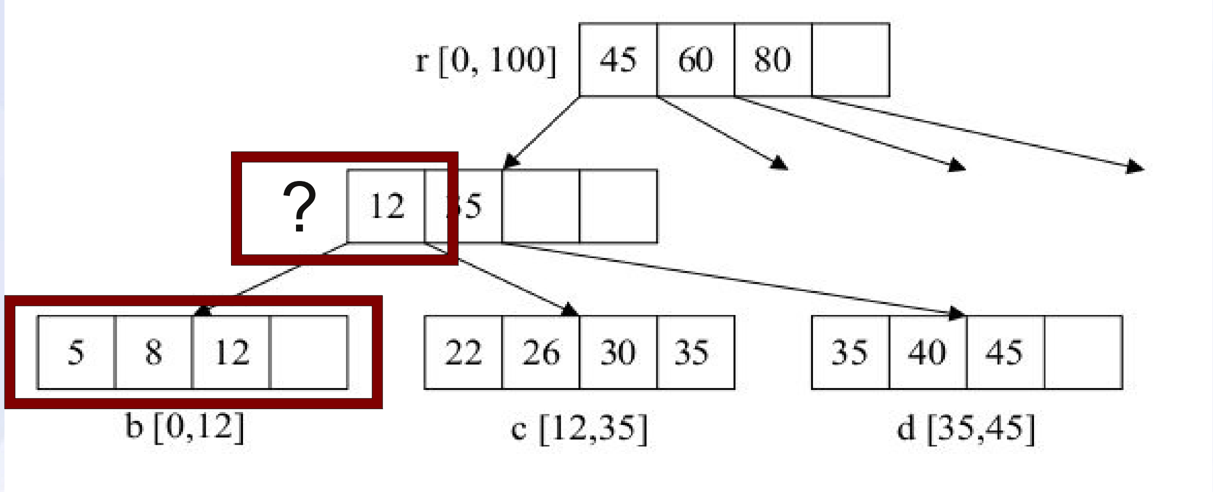


# Index creation

*2. How is a B+tree-node indexed in the CG-index?*

1. Compute the key range for the node

→ look up in the parent node

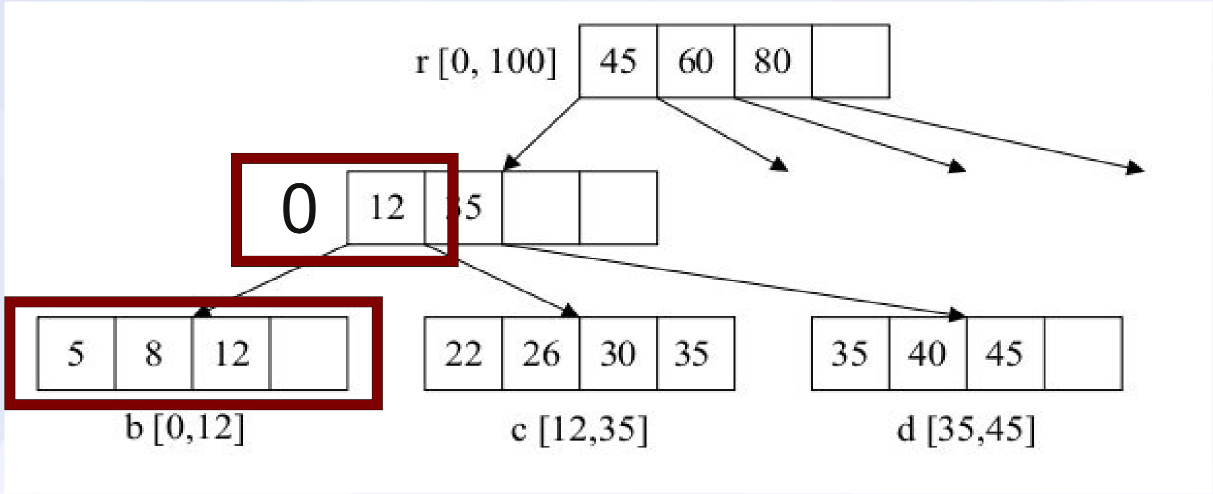


# Index creation

*2. How is a B+tree-node indexed in the CG-index?*

1. Compute the key range for the node

→ look up in the parent node



# Index creation

*2. How is a B+tree-node indexed in the CG-index?*

2. Find the corresponding CG-Node in BATON

→ go down the tree until lower bound of range is found

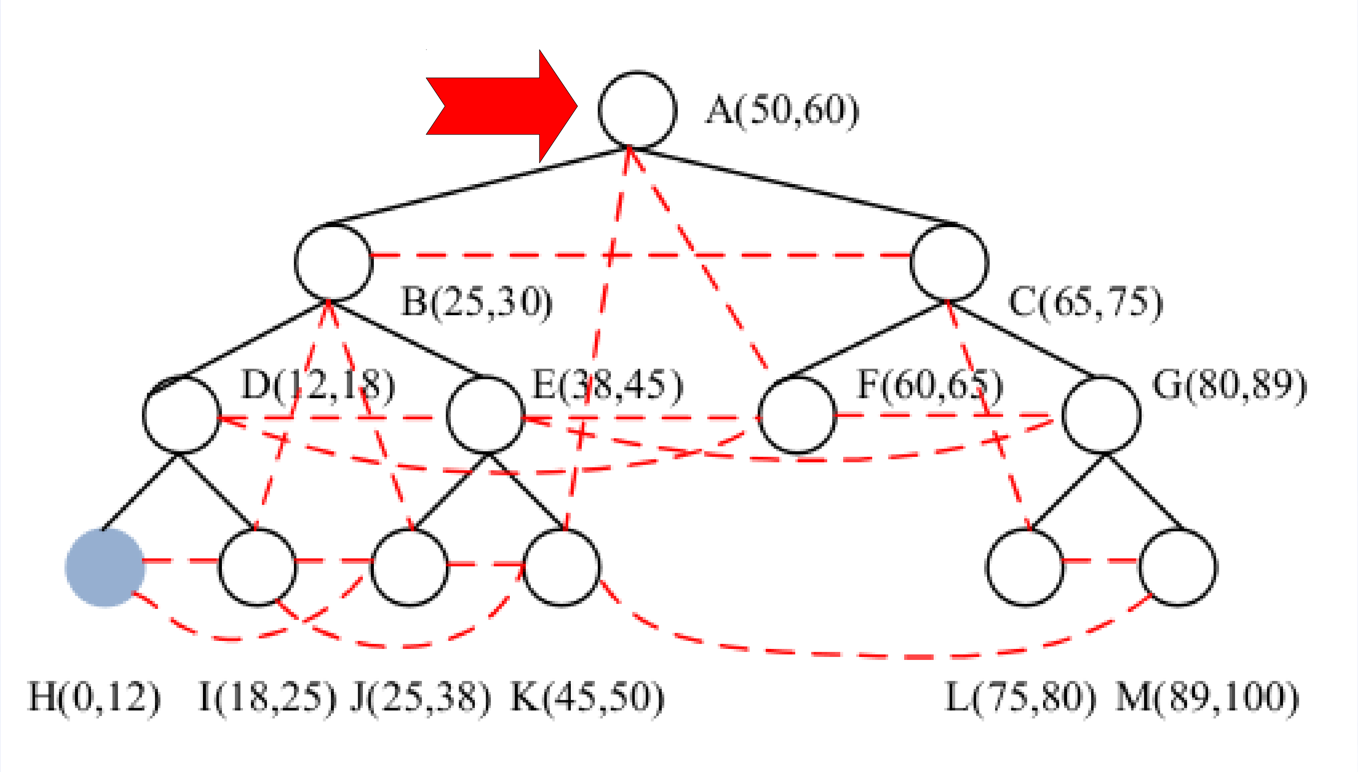
→ go up the tree until the complete range is covered by this subtree

3. Store the node index there

# Index creation - example

→ go down the tree until lower bound of range is found

→ go up the tree until the complete range is covered by this subtree

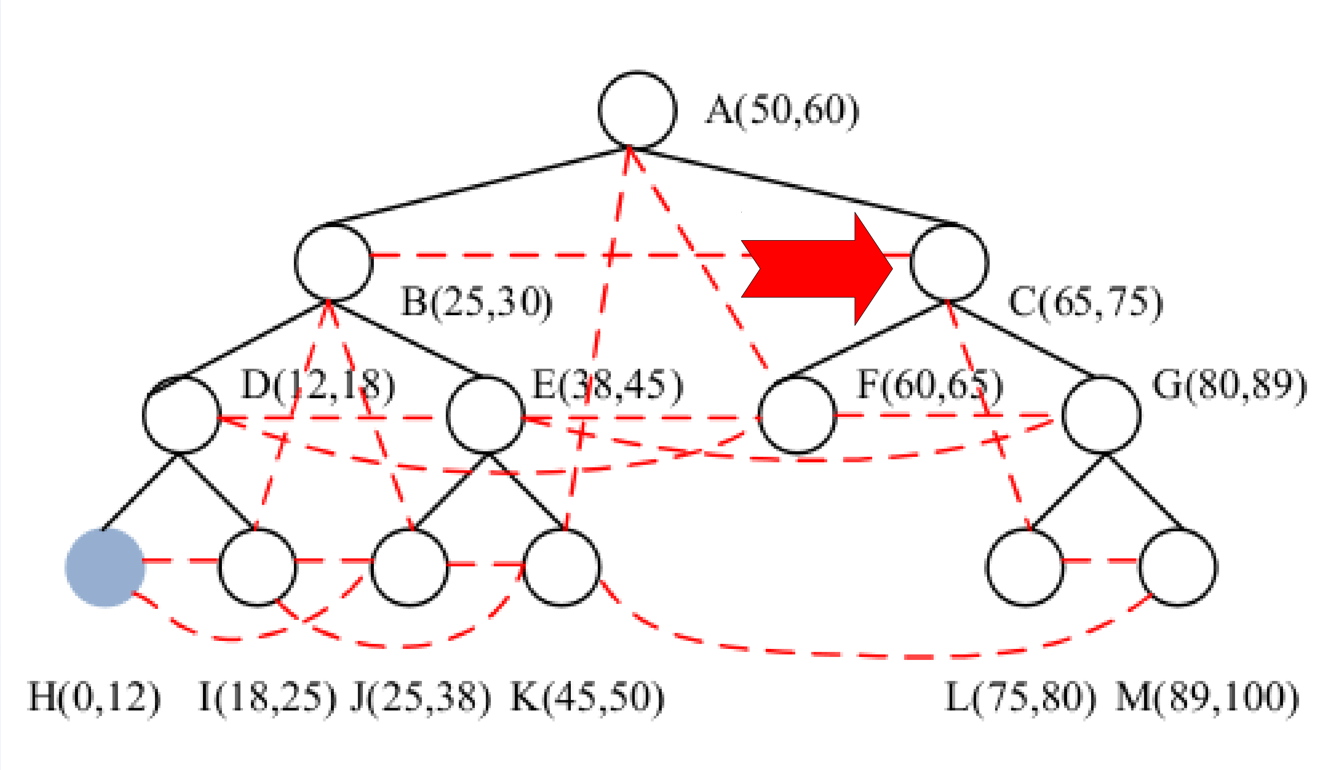


[76,96]

# Index creation - example

→ go down the tree until lower bound of range is found

→ go up the tree until the complete range is covered by this subtree

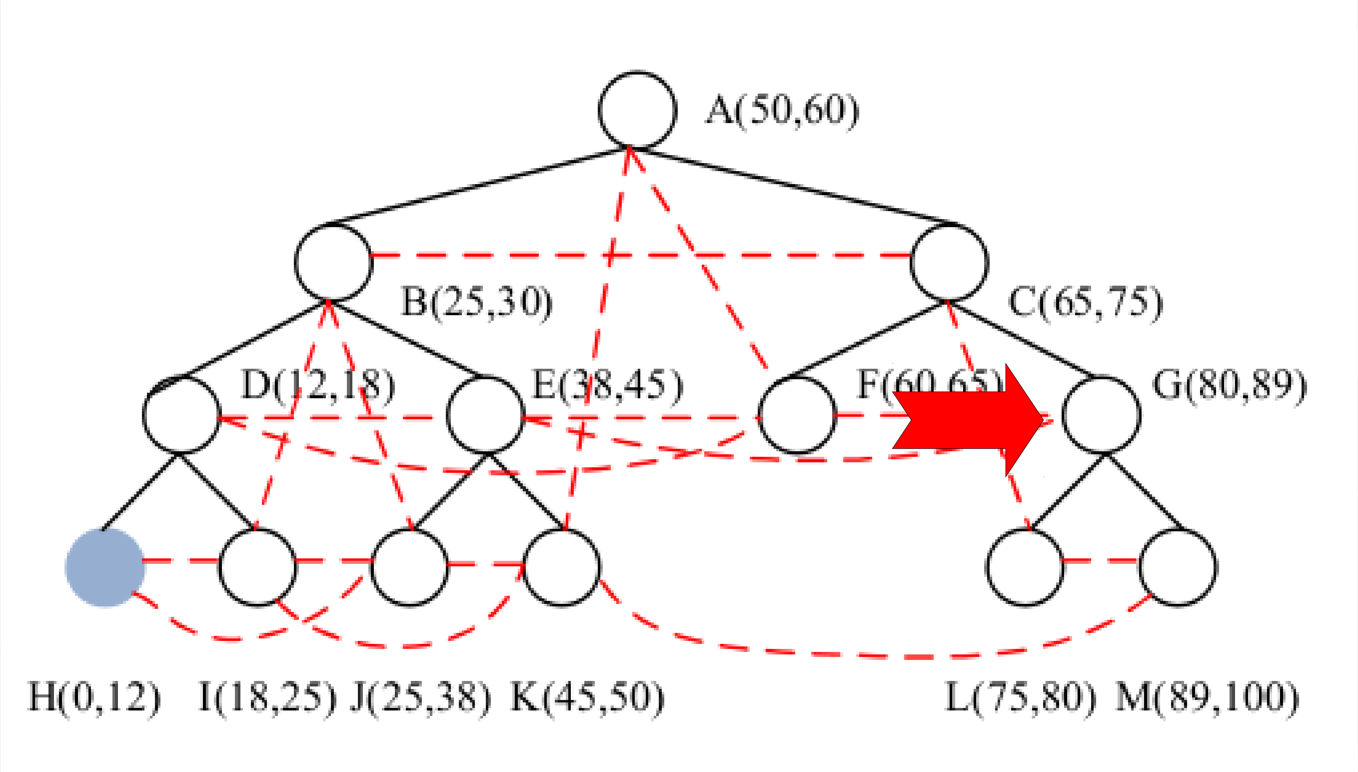


[76,96]

# Index creation - example

→ go down the tree until lower bound of range is found

→ go up the tree until the complete range is covered by this subtree

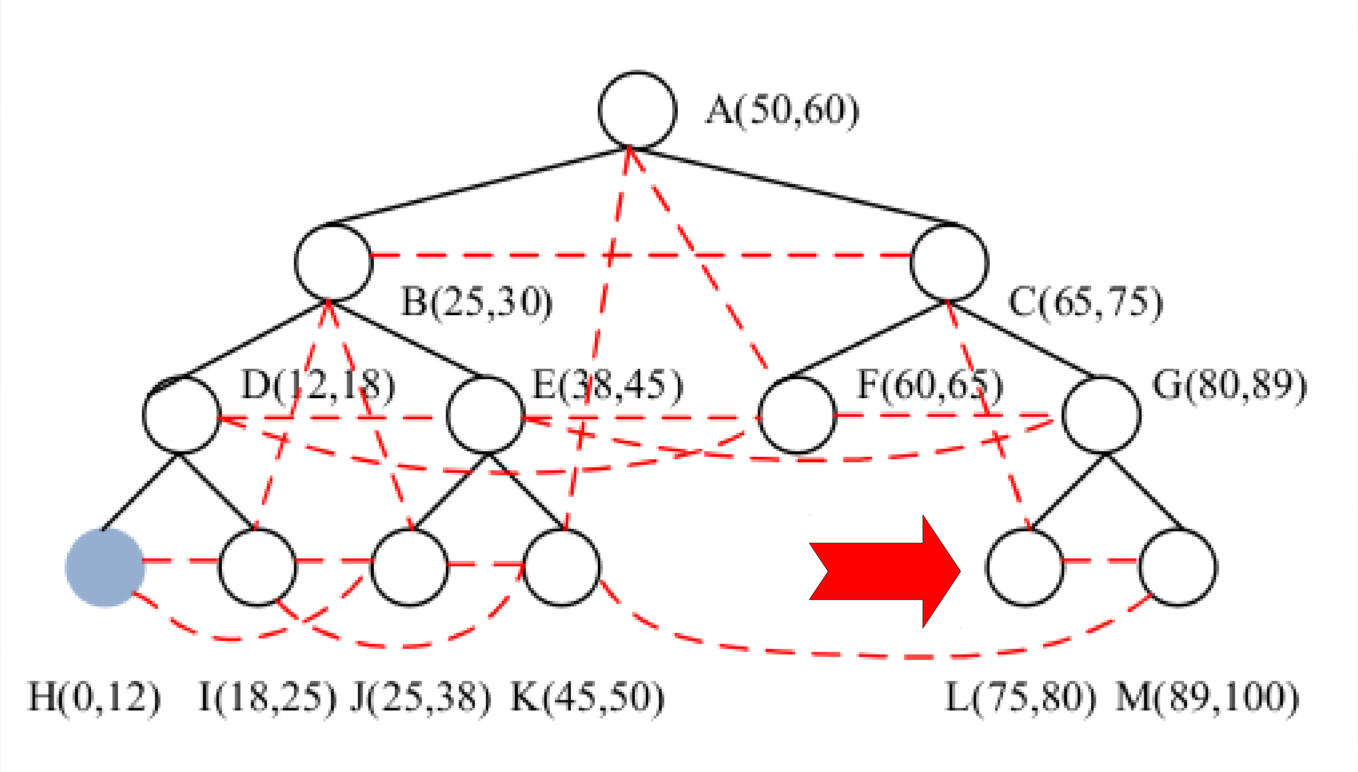


[76,96]

# Index creation - example

→ go down the tree until lower bound of range is found

→ go up the tree until the complete range is covered by this subtree

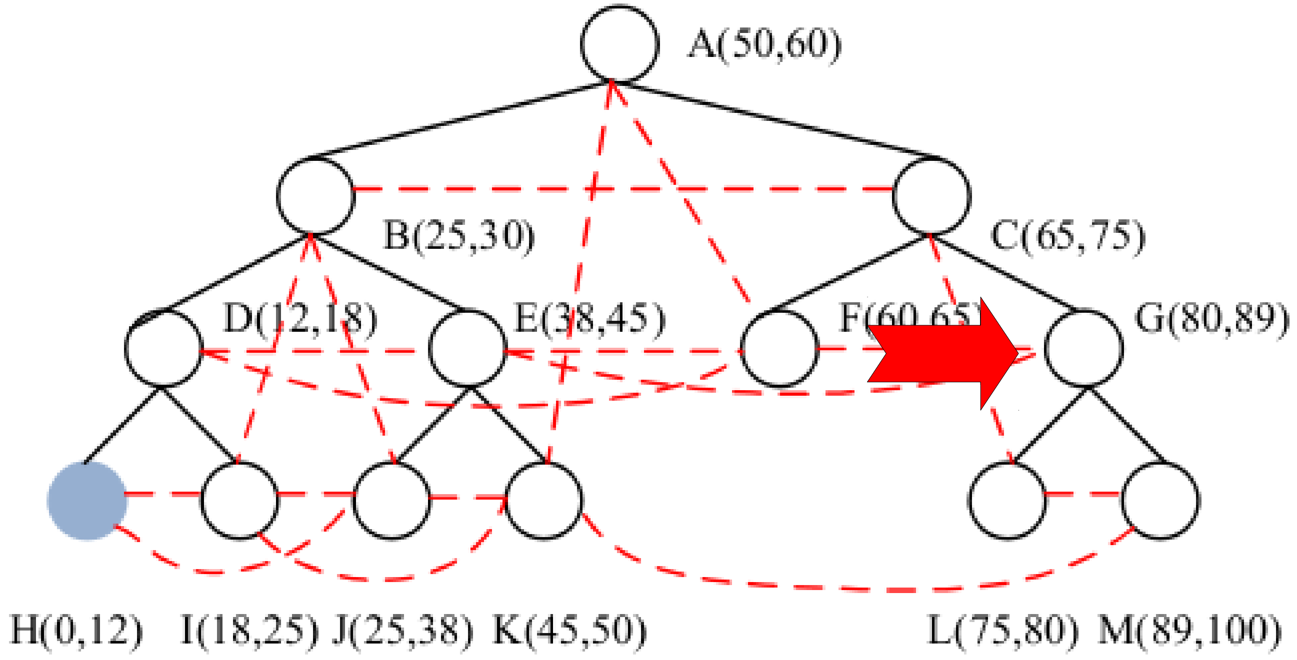


[76,96]

# Index creation - example

→ go down the tree until lower bound of range is found

→ go up the tree until the complete range is covered by this subtree



[76,96]

# Index structure

How does an index entry look like?

- Each entry has 4 attributes:
  - **blk** : disk block number
  - **range**: range of values in this node
  - **keys**: search keys used
  - **ip** : ip of remote node

# Data retrieval

## 1. How is the data retrieved?

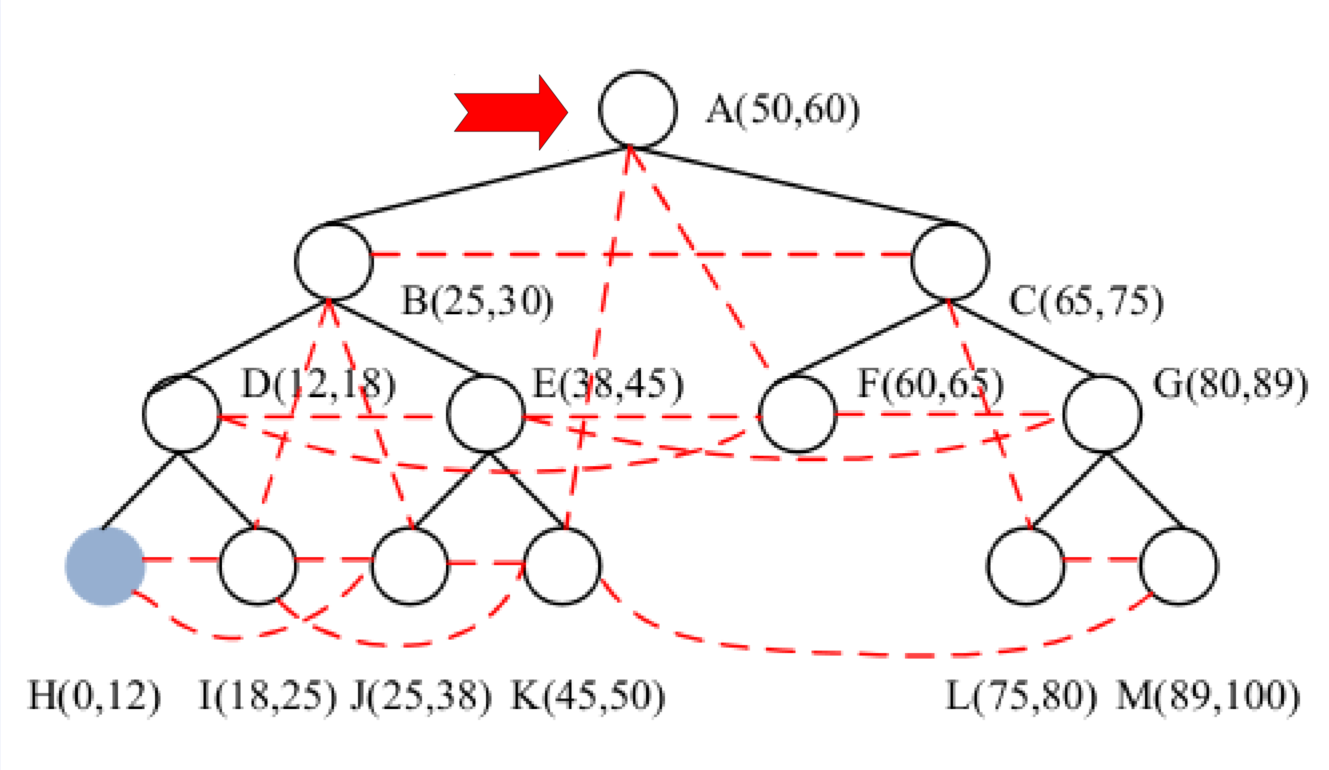
1. Find all (local) B+-tree nodes in CG-Index which overlap with query range R
    - Go to the CG-node with the lower bound of R
    - Traverse all sibling nodes until the upper bound of R, fetch B+-trees in indices

2. Search the fetched B+-trees in parallel

# Data retrieval

Go to the CG-node with the lower bound of R

Traverse all sibling nodes until the upper bound of R, fetch B+-trees in indices

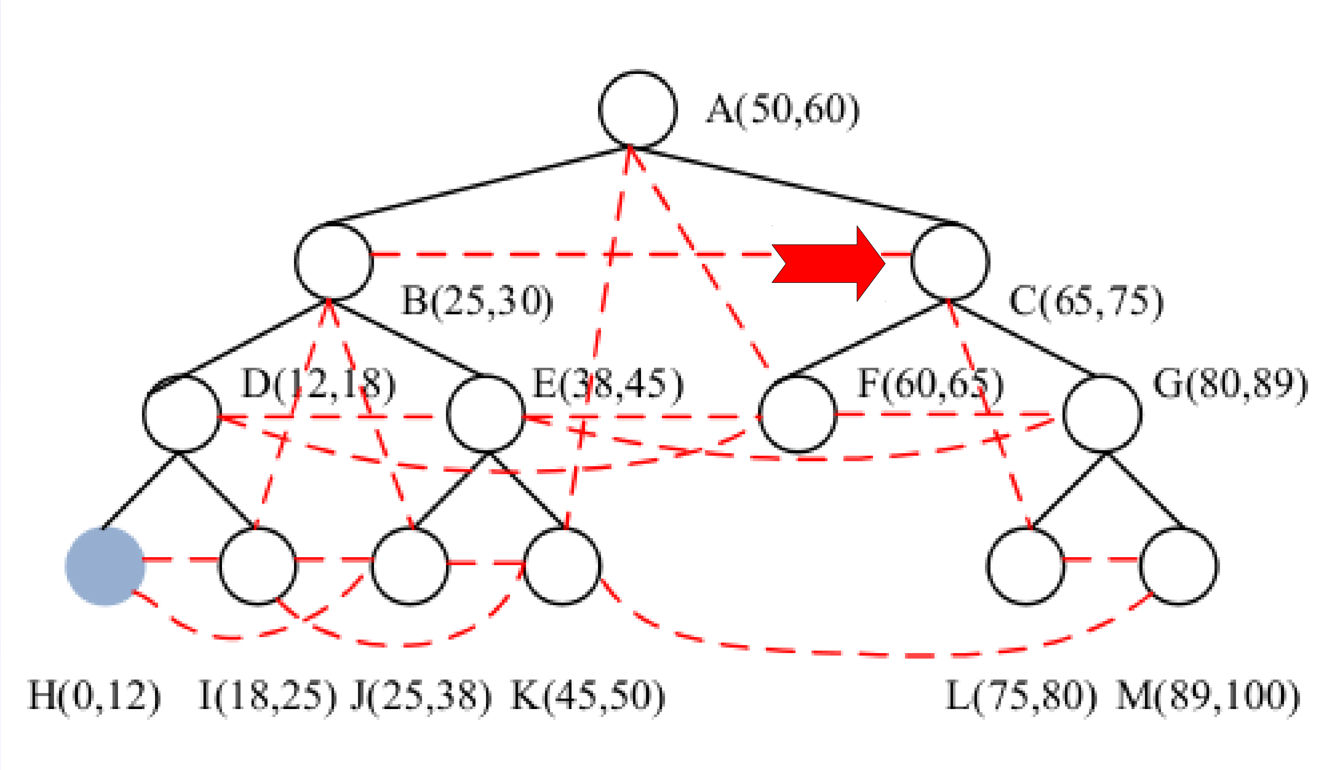


[63,96]

# Data retrieval

Go to the CG-node with the lower bound of R

Traverse all sibling nodes until the upper bound of R, fetch B+-trees in indices

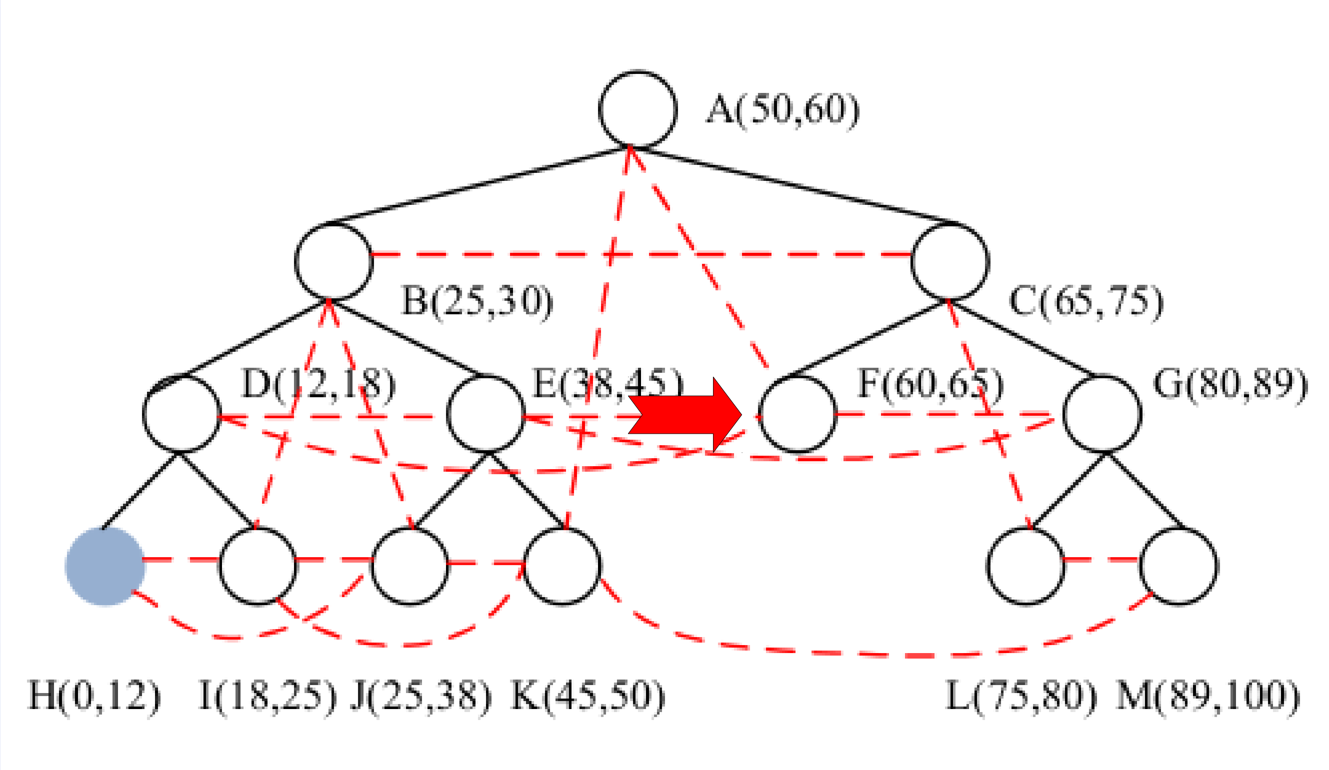


[63,96]

# Data retrieval

Go to the CG-node with the lower bound of R

Traverse all sibling nodes until the upper bound of R, fetch B+-trees in indices

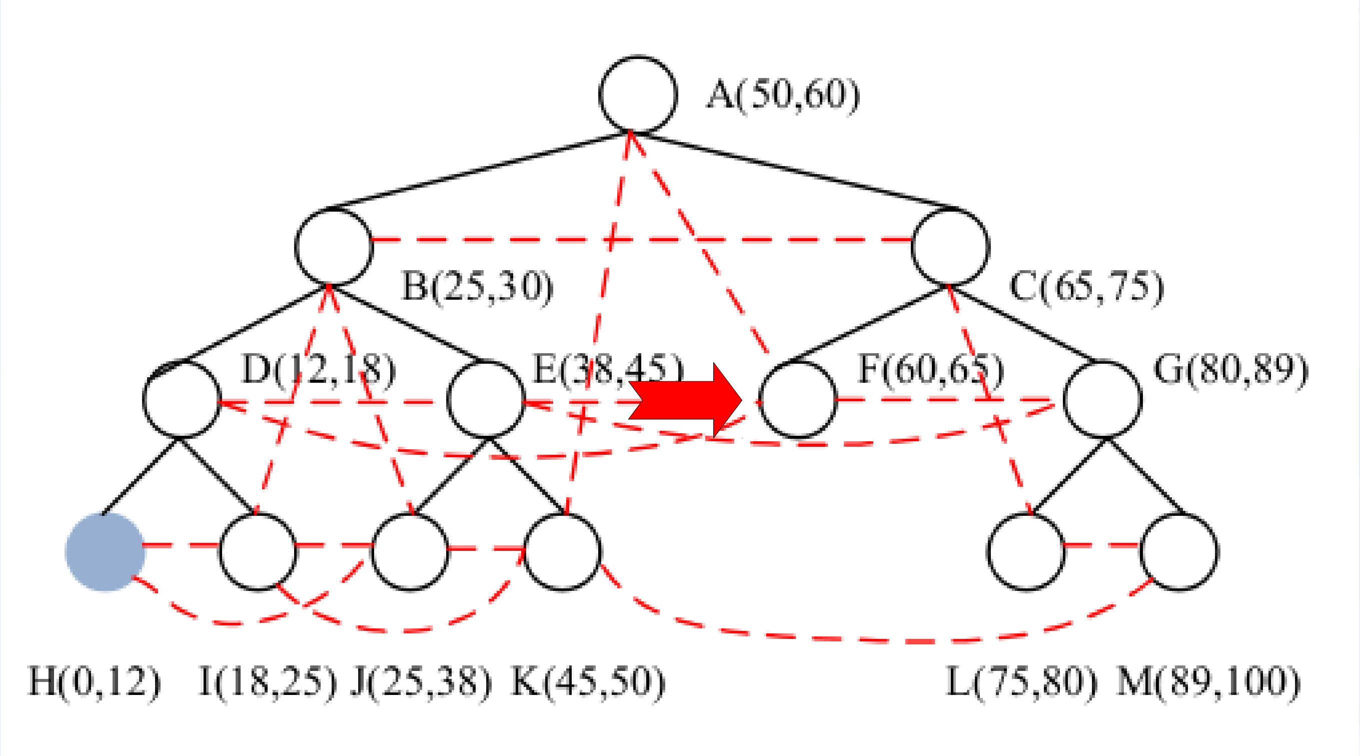


[63,96]

# Data retrieval

Go to the CG-node with the lower bound of R

Traverse all sibling nodes until the upper bound of R, fetch B+-trees in indices

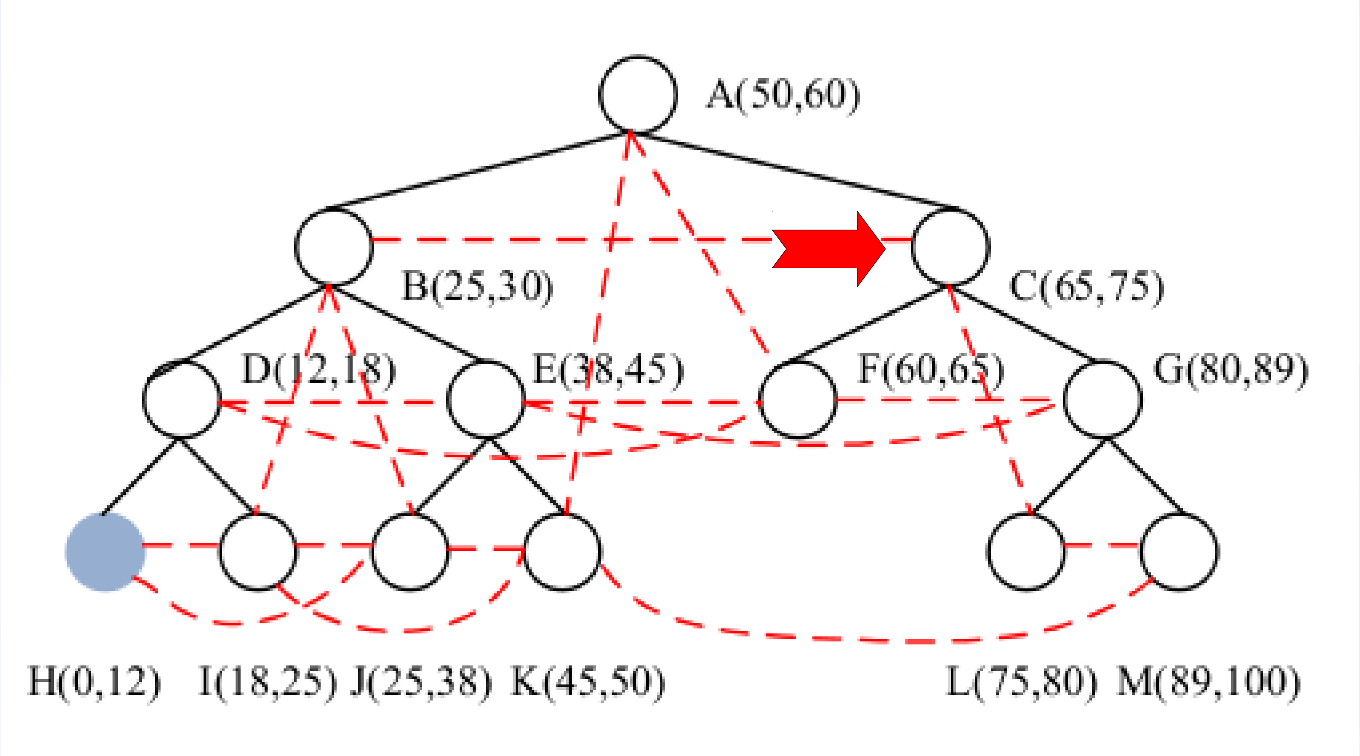


[63,96]

# Data retrieval

Go to the CG-node with the lower bound of R

Traverse all sibling nodes until the upper bound of R, fetch B+-trees in indices

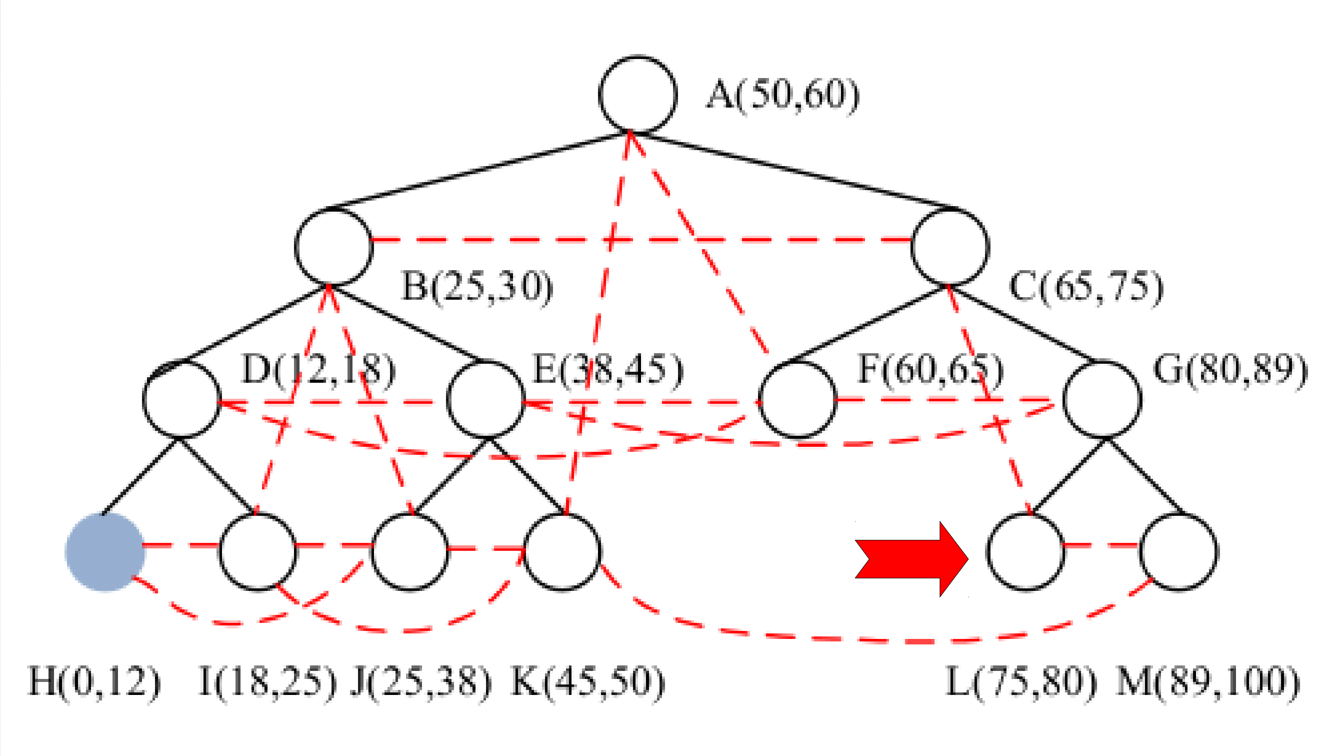


[63,96]

# Data retrieval

Go to the CG-node with the lower bound of R

Traverse all sibling nodes until the upper bound of R, fetch B+-trees in indices

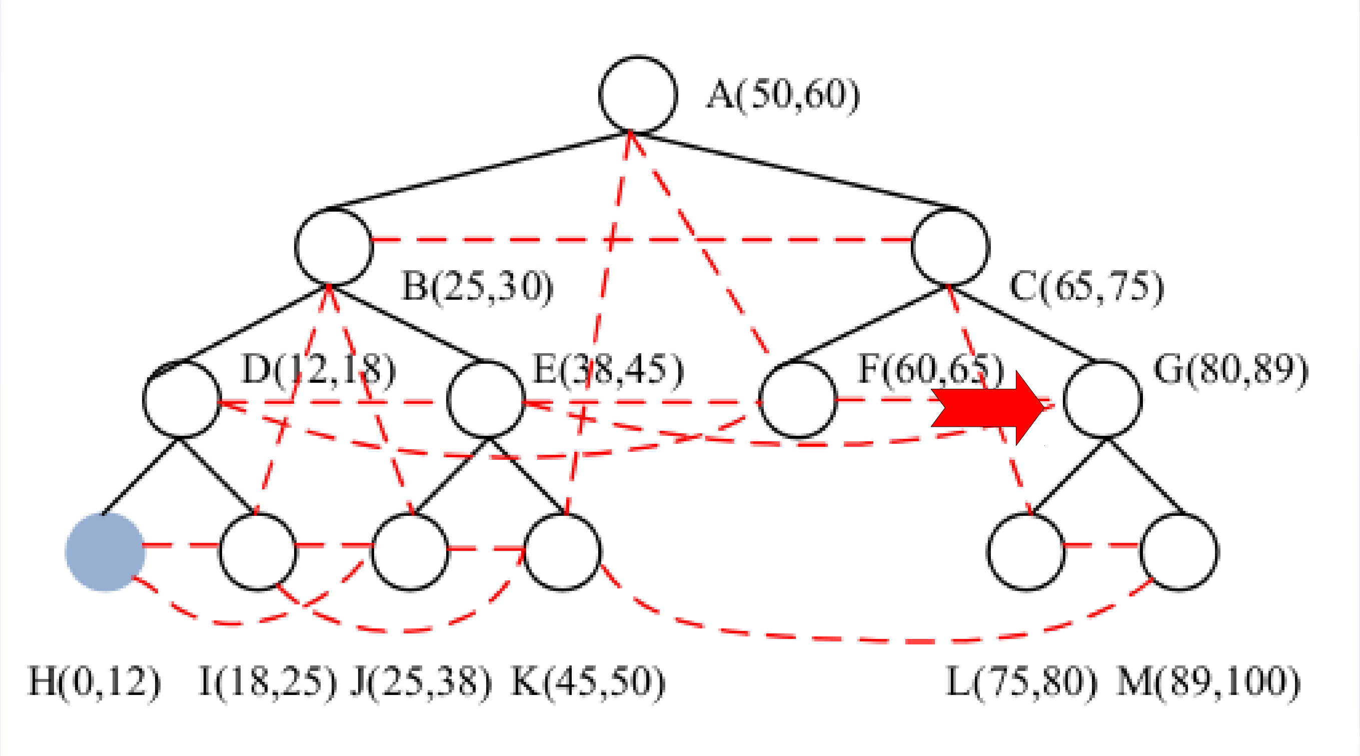


[63,96]

# Data retrieval

Go to the CG-node with the lower bound of R

Traverse all sibling nodes until the upper bound of R, fetch B+-trees in indices

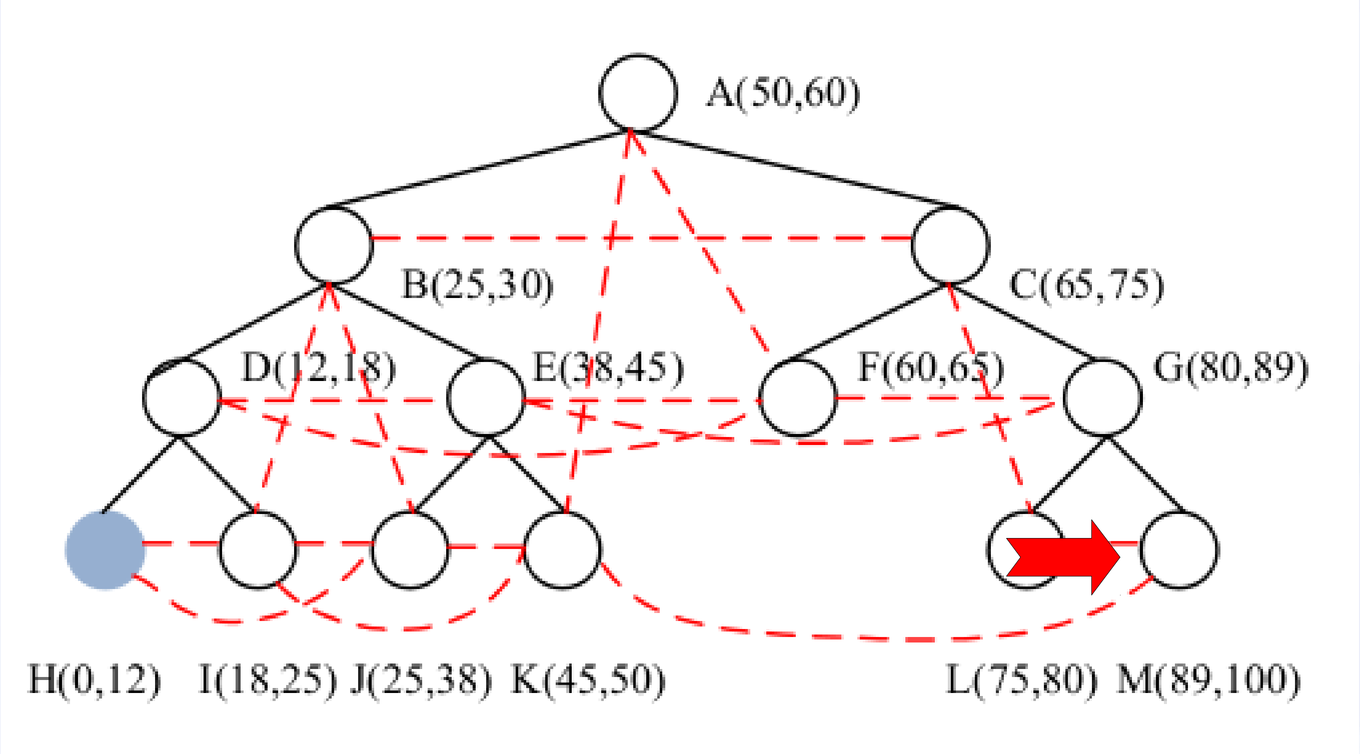


[63,96]

# Data retrieval

Go to the CG-node with the lower bound of R

Traverse all sibling nodes until the upper bound of R, fetch B+-trees in indices



[63,96]

# Optimization

Currently: Going down and up in tree after finding lowest key

→ Don't search for node with lowest key, but for arbitrary one in the search range R

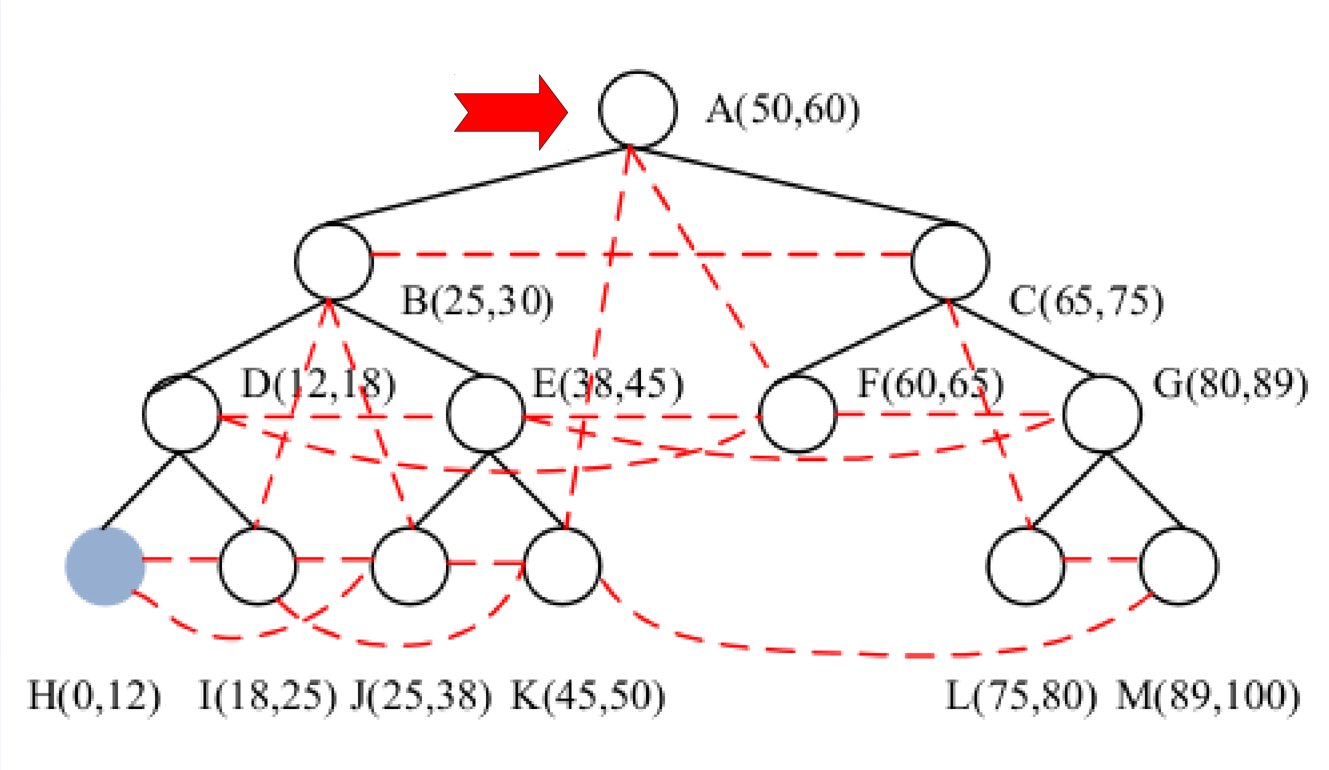
→ reduces the cost by k / |R|, where

k ist the total number of nodes
|R| is the number of nodes in the range R

# Data retrieval

Go to the CG-node with the lower bound of R

Traverse all sibling nodes until the upper bound of R, fetch B+-trees in indices

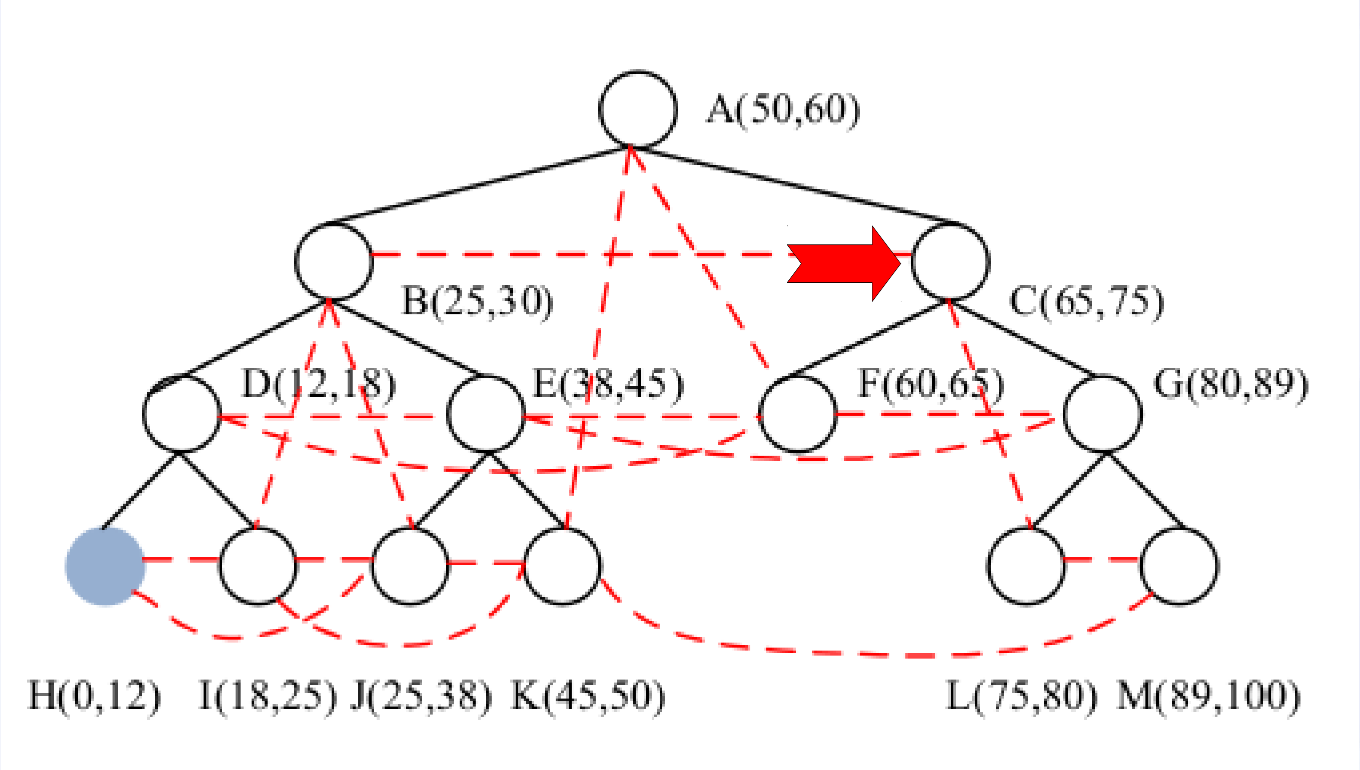


[63,96]

# Data retrieval

Go to the CG-node with the lower bound of R

Traverse all sibling nodes until the upper bound of R, fetch B+-trees in indices

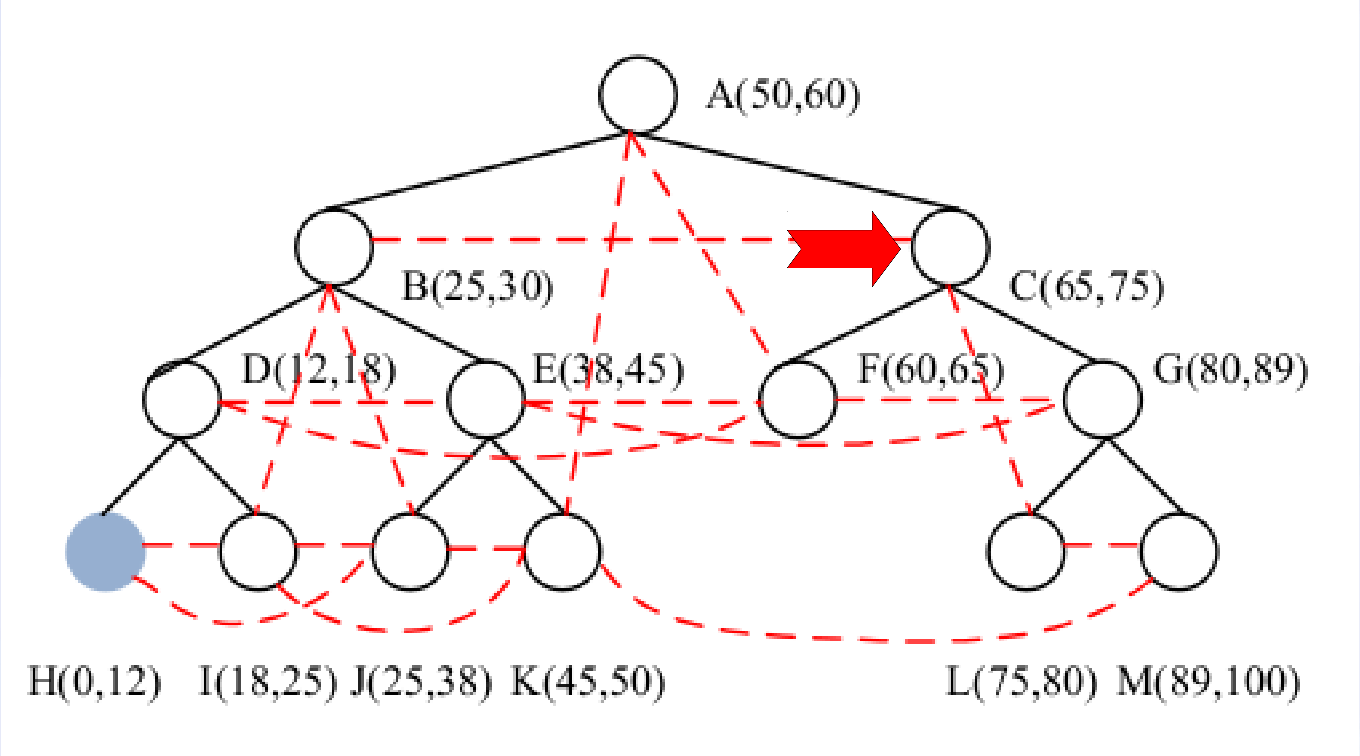


[63,96]

# Data retrieval

Go to the CG-node with the lower bound of R

Traverse all sibling nodes until the upper bound of R, fetch B+-trees in indices

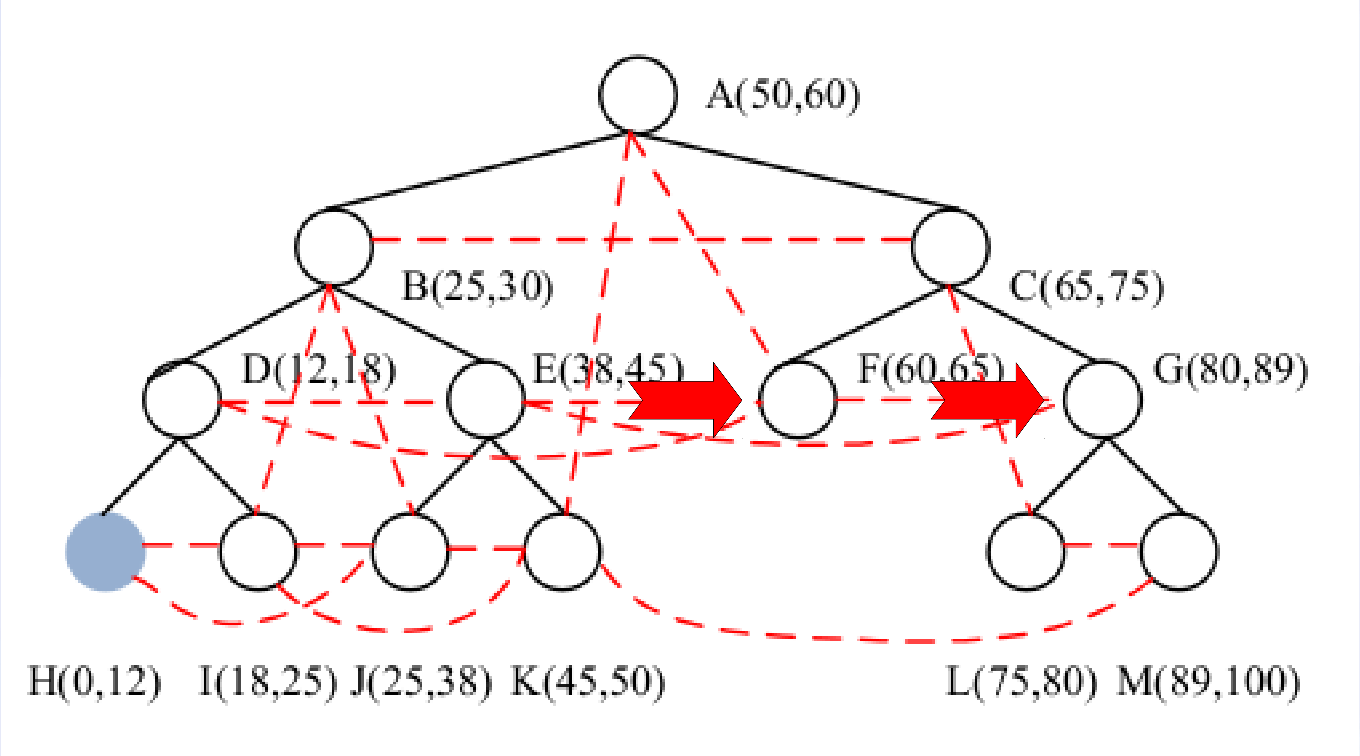


[63,96]

# Data retrieval

Go to the CG-node with the lower bound of R

Traverse all sibling nodes until the upper bound of R, fetch B+-trees in indices

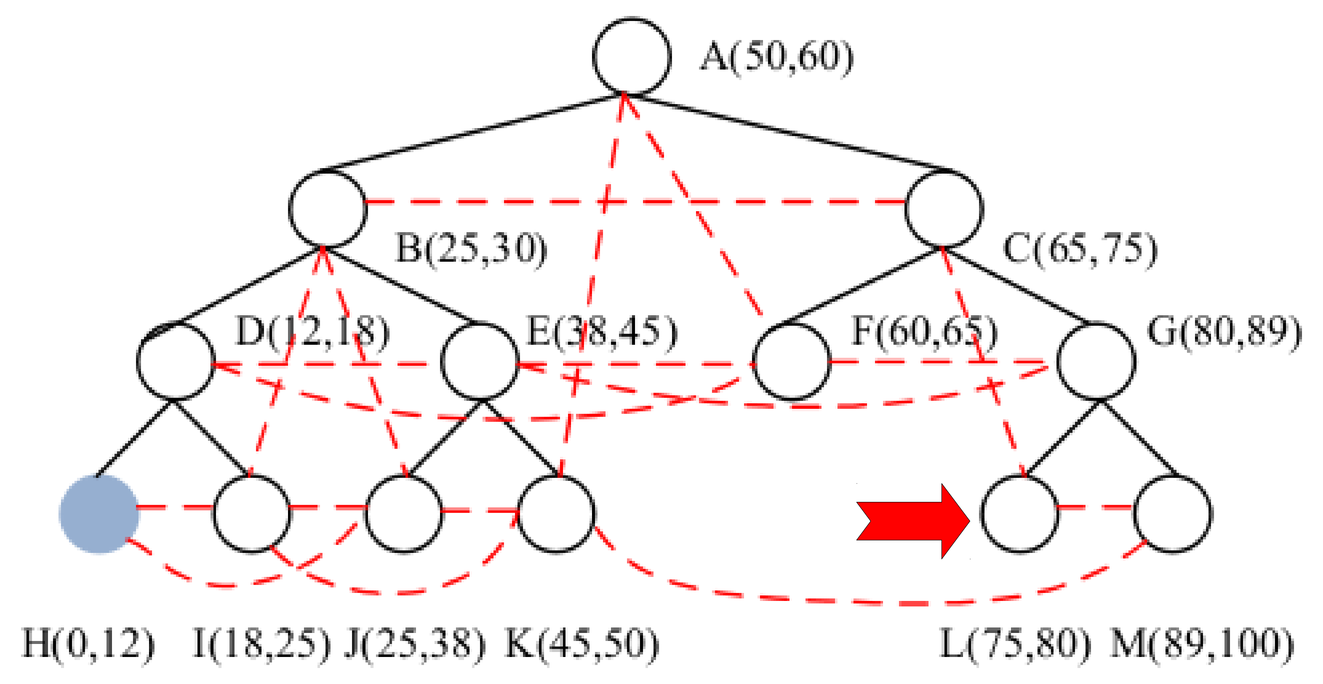


[63,96]

# Data retrieval

Go to the CG-node with the lower bound of R

Traverse all sibling nodes until the upper bound of R, fetch B+-trees in indices

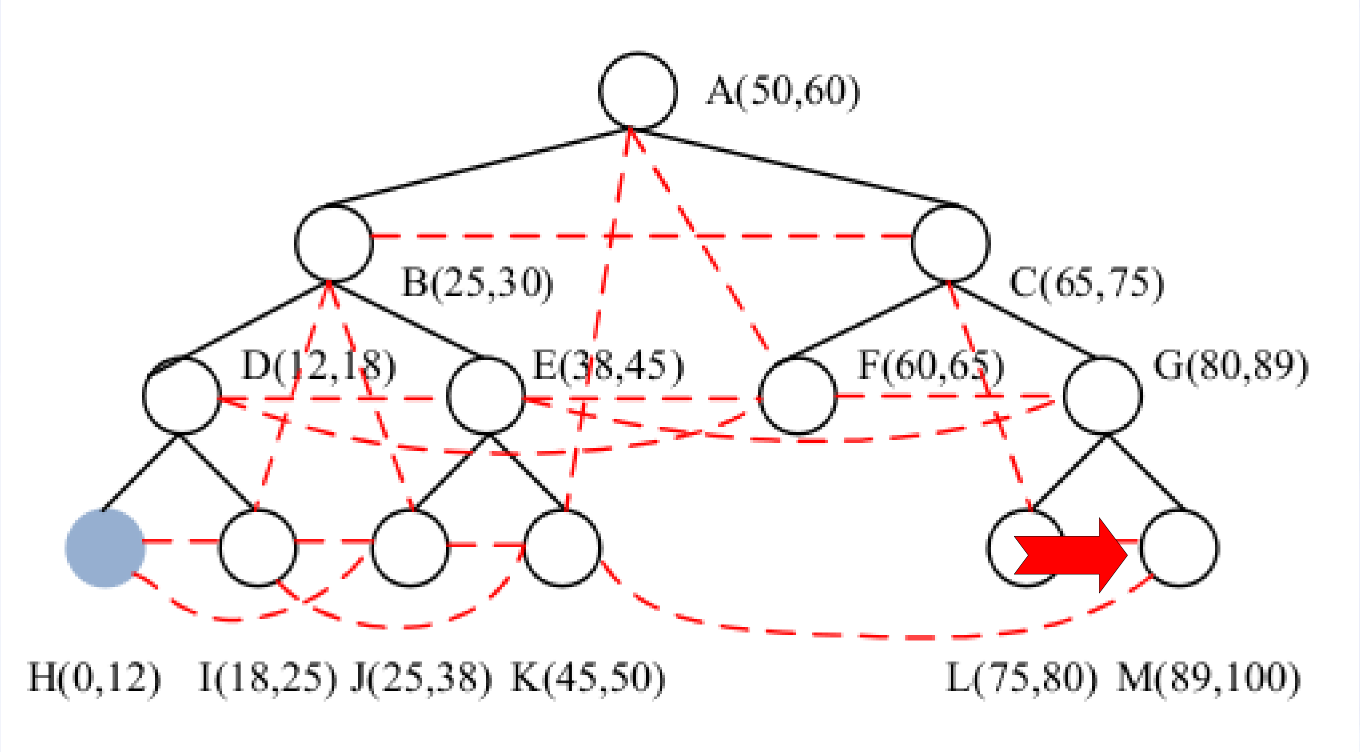


[63,96]

# Data retrieval

Go to the CG-node with the lower bound of R

Traverse all sibling nodes until the upper bound of R, fetch B+-trees in indices



[63,96]

# Optimization 2

- Our approach is working sequential, fetching nodes one after another

  → search indices in parallel

1. Find parent node which covers the whole tree
2. After this, broadcast message is sent to

# Final search algorithm

1. Find node in the range R

2. Find parent node which covers the whole tree

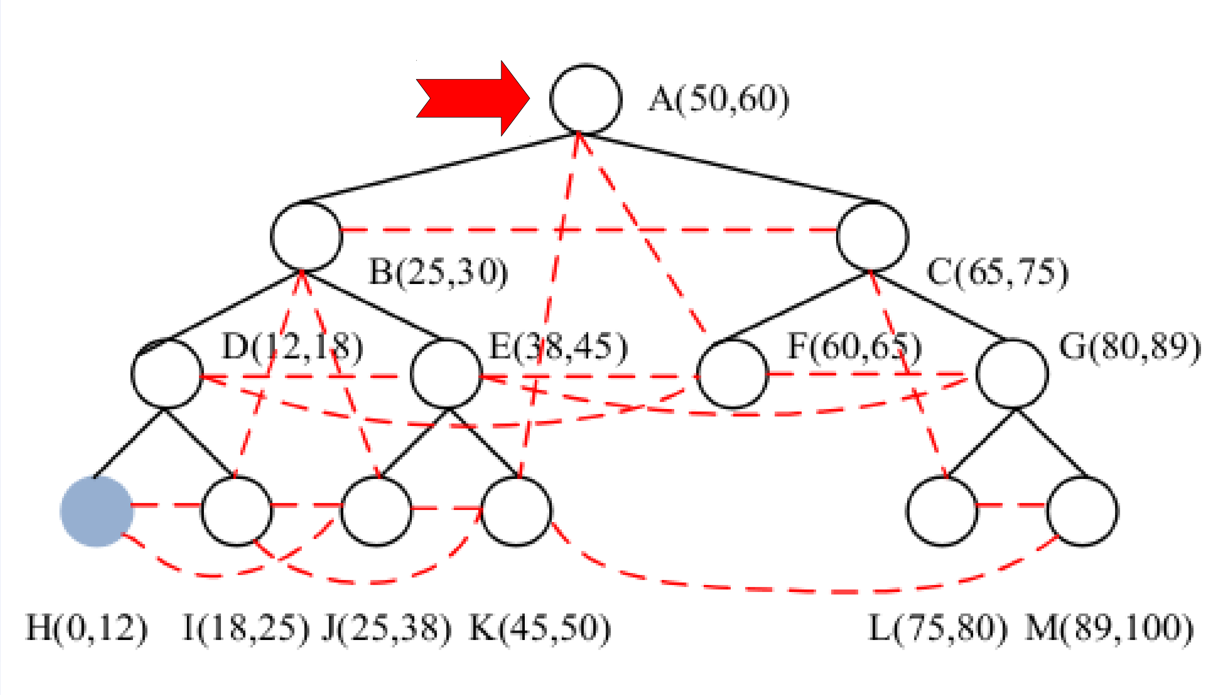
3. Send broadcast message to child nodes, which then search in parallel

# Final search algorithm

1. Find node in the range R

2. Find parent node which covers the whole tree

3. Send broadcast message to child nodes, which then search in parallel



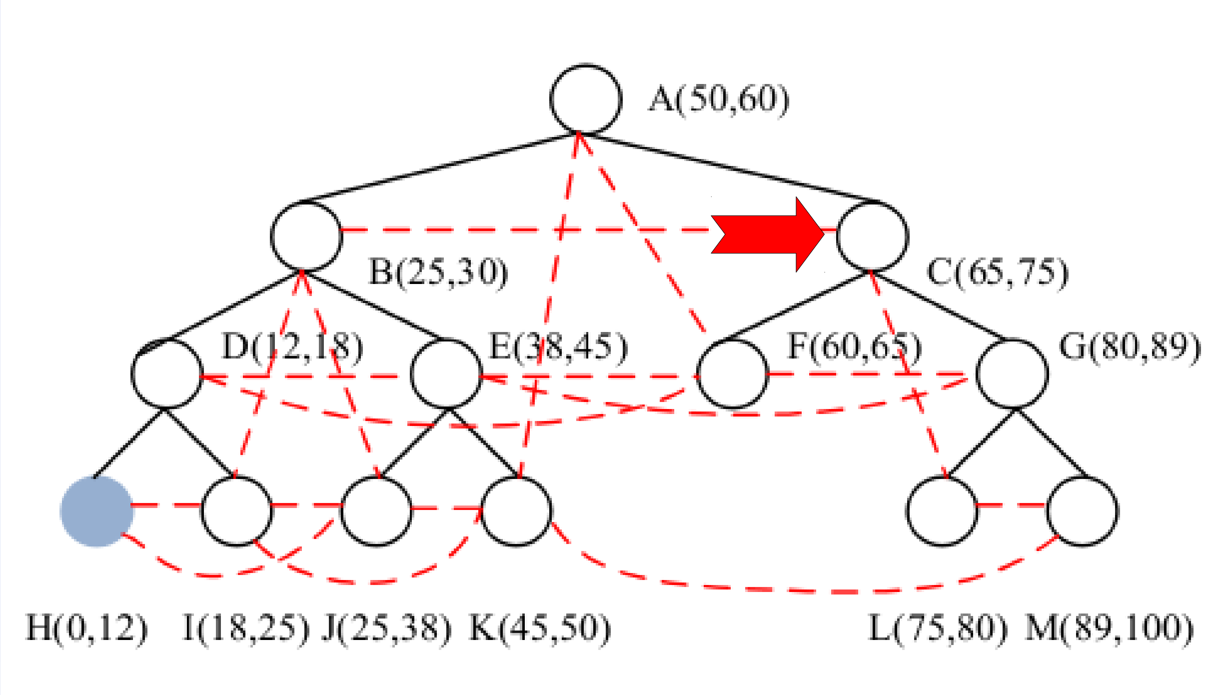
[63,96]

# Final search algorithm

1. Find node in the range R

2. Find parent node which covers the whole tree

3. Send broadcast message to child nodes, which then search in parallel



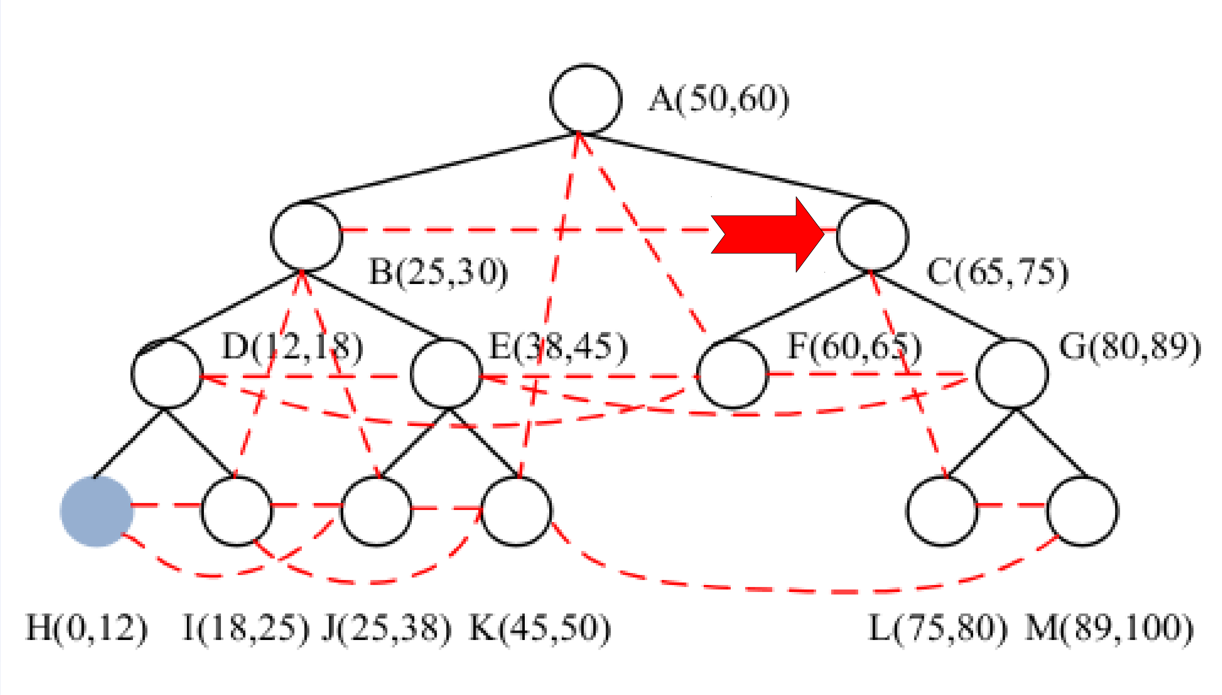
[63,96]

# Final search algorithm

1. Find node in the range R

2. Find parent node which covers the whole tree

3. Send broadcast message to child nodes, which then search in parallel



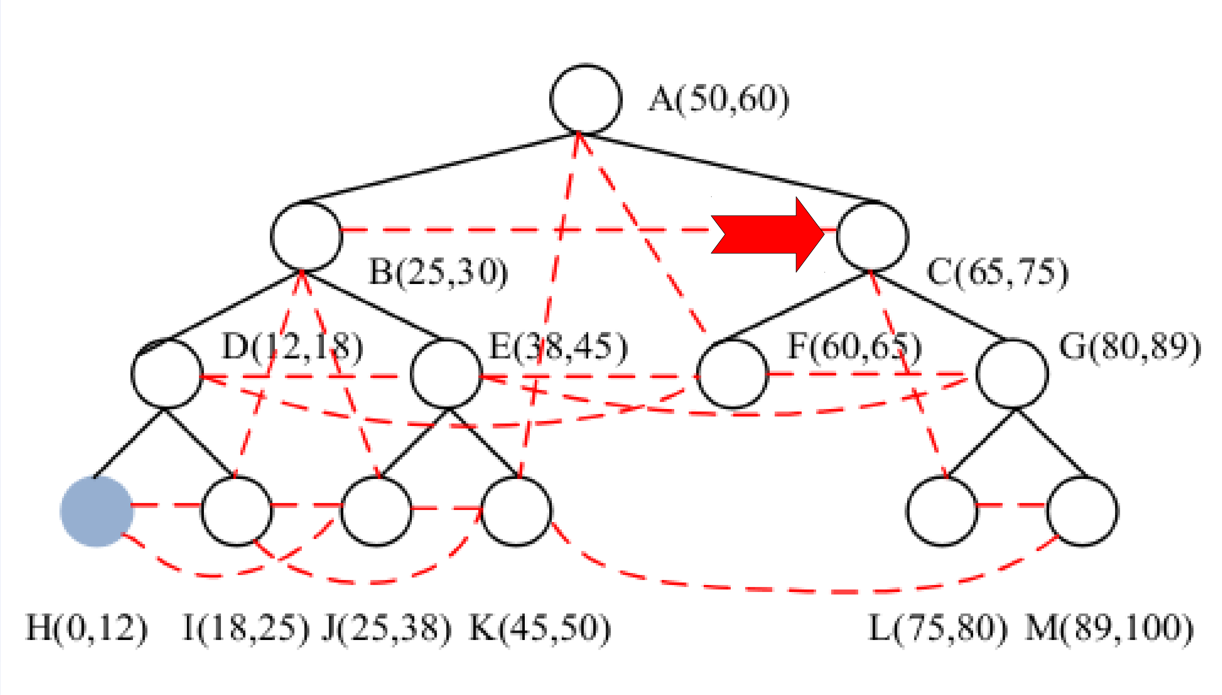
[63,96]

# Final search algorithm

1. Find node in the range R

2. Find parent node which covers the whole tree

3. Send broadcast message to child nodes, which then search in parallel



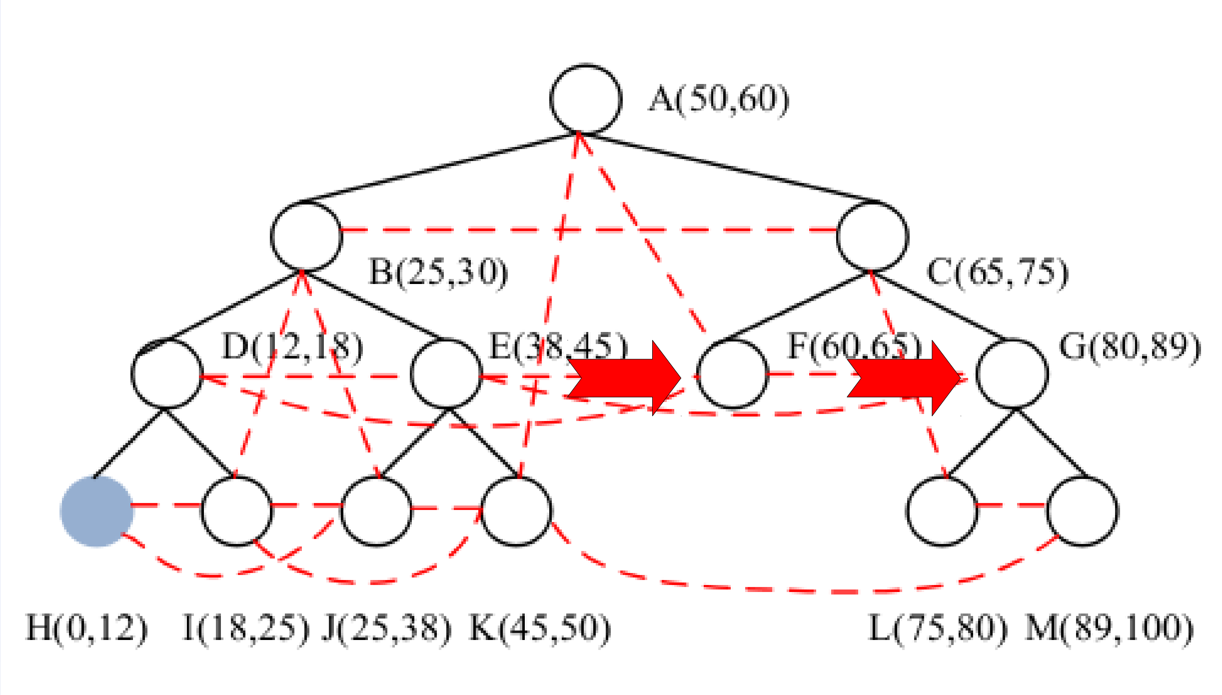
[63,96]

# Final search algorithm

1. Find node in the range R

2. Find parent node which covers the whole tree

3. Send broadcast message to child nodes, which then search in parallel



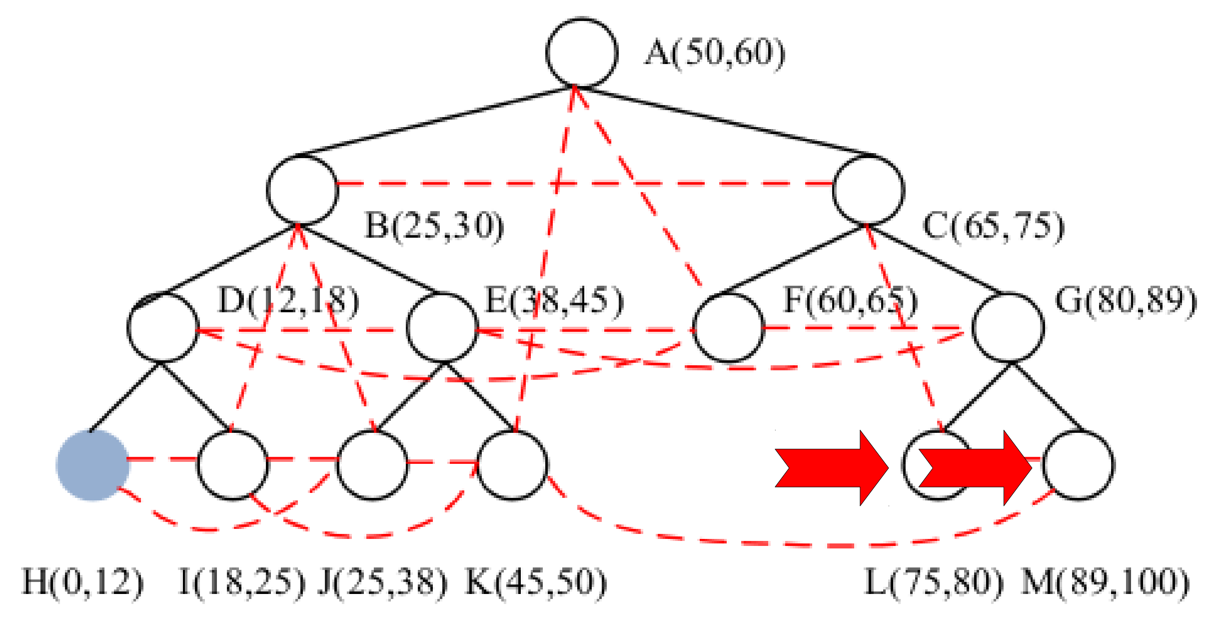
[63,96]

# Final search algorithm

1. Find node in the range R

2. Find parent node which covers the whole tree

3. Send broadcast message to child nodes, which then search in parallel



[63,96]

# Outline

- Motivation
- Solutions: creation & usage
- **Solutions: maintenance**
- Tuning
- Evaluation
- Conclusion

# Updates

## 4. How are updates performed?

- Trivial for local B+ - trees

- Harder for global CG-index

- 2 different types of updates for CG-index:

  - Lazy updates:
    - Missed updates **do not** lead to wrong results
    - All updates are committed together **after a time threshold**

  - Eager updates:
    - Missed updates **do** lead to wrong results
    - Committed **immediately**

- Only updates in left- and rightmost part of B+-tree can lead to changed query ranges
  → *only eager update for some of these nodes possible*

# Lazy updates

*What to do if two nodes n1 and n2 are merged/split?*

1. If both **n1 and n2 are in the CG-index** and are **merged**
   → **replace** their entries with the **merged one**

2. If **only one** is in the CG-index (let's say n1) and **merged**
   → **replace** entry of **n1** with **merged one**

3. If n is in the CG-index and **split**
   → replace entry of n with entries of the **2 new nodes**

# Eager updates

- Updates <u>shrinking the node range</u> generate <u>false positives</u>:
  - Node n is stored with range [10,20] in the index
  - Update deletes 10, next smallest tuple is 12
    → range is now [12,20]
  - For range query from [5,11], index will also return n, although there is no tuple

→ But this doesn't violate consistency, no data is missed

→ **Apply lazy update technique**

# Eager updates

- Updates expanding the node range generate false negatives:
  - Node n is stored with range [10,20] in the index
  - Update inserts 8
    → range is now [8,20]
  - For range query from [5,9], index will not return n, although there is a tuple

→ Violates consistency, data is missed

→ **Apply eager update technique**

# Replication

*How is data consistency assured?*

- Left and right neighbour nodes have a copy
- Left node is primary copy
- On update, copies are notified first, main node is commiting as the last
- Nodes ping their routing neighbours frequently → check if alive
- If primary node restarts after failure

  → compares timestamps with current master node

  → applies missing updates

# Outline

- Motivation

- Solutions: creation & usage
- Solutions: maintenance
- **Tuning**

- Evaluation

- Conclusion

# Tuning

- Several tuning approaches proposed, but not yet implemented

1. Routing Buffer
   *Buffer often visited nodes*

2. Selective expansion
   *Only select children nodes which are used*

# Tuning : routing buffer

*Routing Buffer*

- Reduce cost for traversing the BATON routing tree
- If Baton node is found for a query: requesting compute node saves the node's IP and range in a buffer
- Node is then checking first its buffer for the next query
- Buffer has S entries, LRU strategy

→ **Frequently queried ranges are accelerated**

# Tuning: selective expansion

*Selective expansion*

- Indexing strategy indexes **all** children
- Nodes have often **more than 100 children** with real data
- Not efficient if **only one child** is frequently used

→ compute benefit for each single child, not for the whole group

→ decide which children should be indexed

→ keep parent indexed as long as not all children are indexed

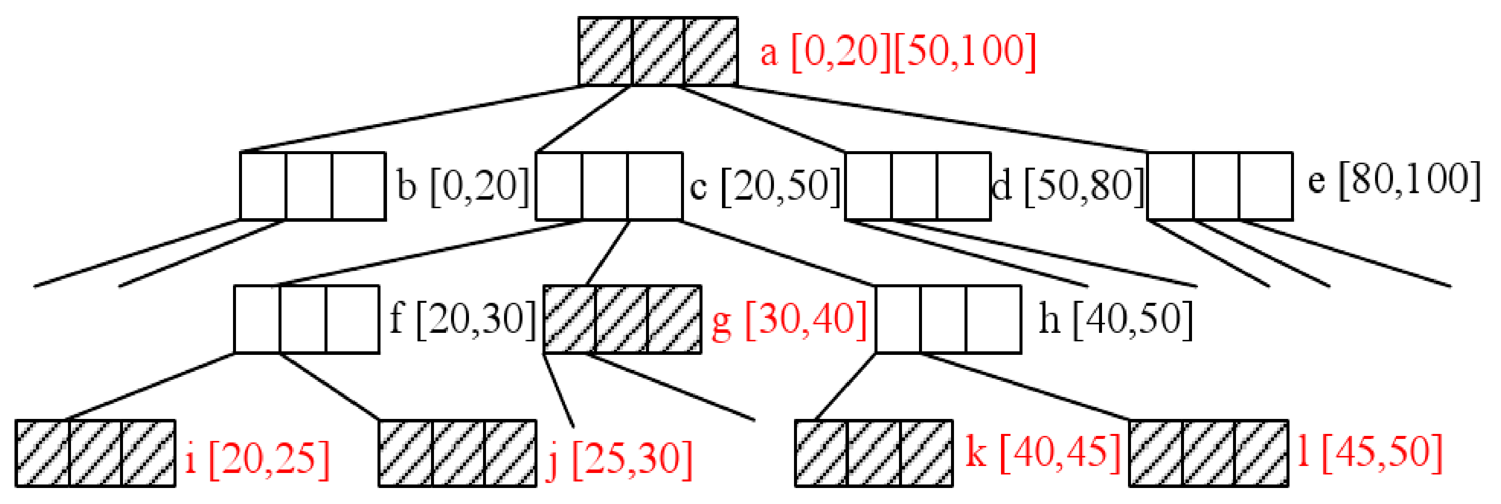
# Tuning: selective expansion

*Selective expansion*

→ compute benefit for each single child, not for the whole group

→ decide which children should be indexed

→ keep parent indexed as long as not all children are indexed



# Outline

- Motivation

- Solutions: creation & usage
- Solutions: maintenance
- Tuning

- **Evaluation**

- Conclusion

# Hardware

- Amazon EC2 cloud

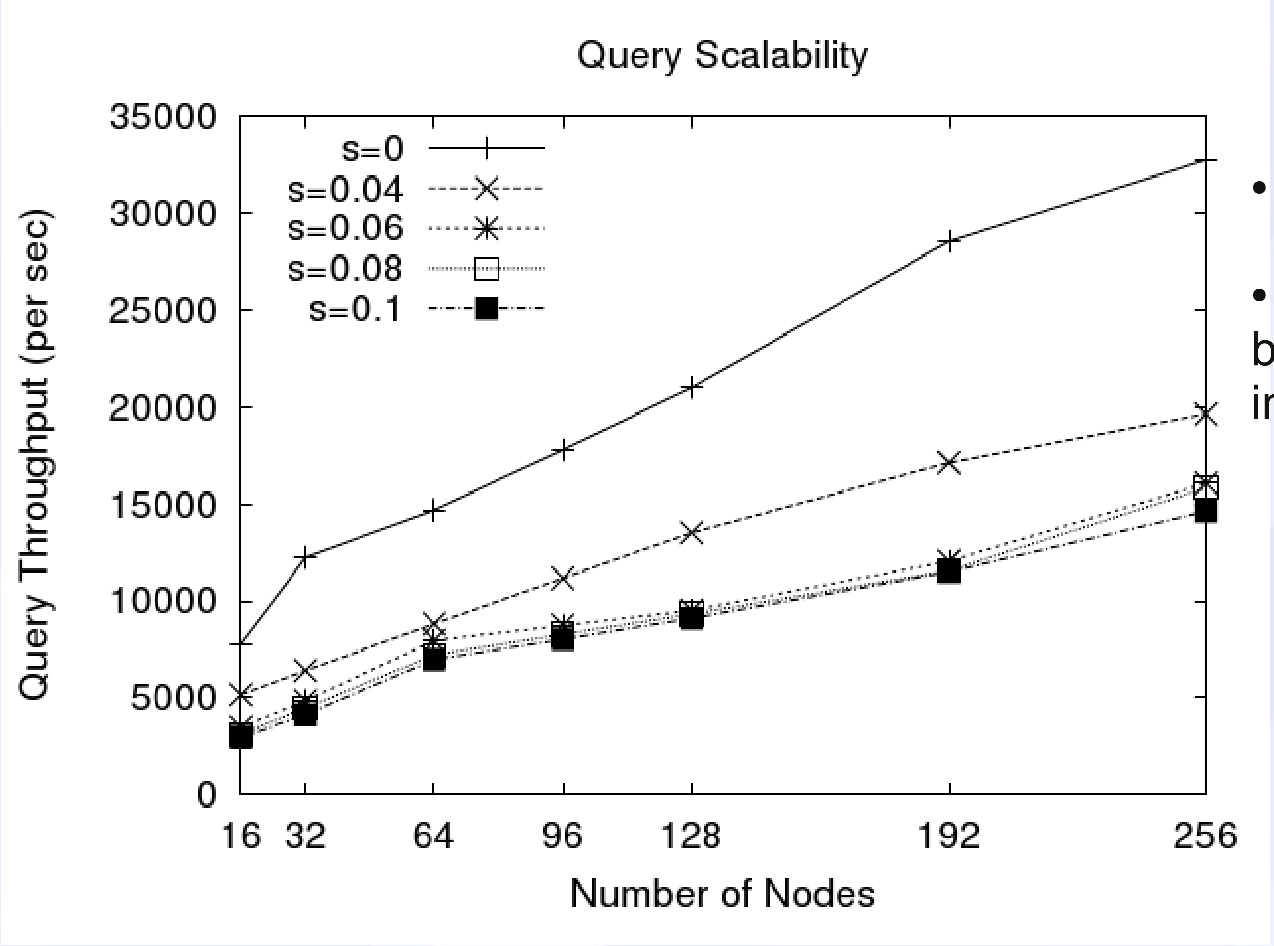
- 250 Mbps network

- Each node:
  - Intel Xeon, 1.7 Ghz
  - 1.7 GB memory

- 500.000 tuples on each node, random generated

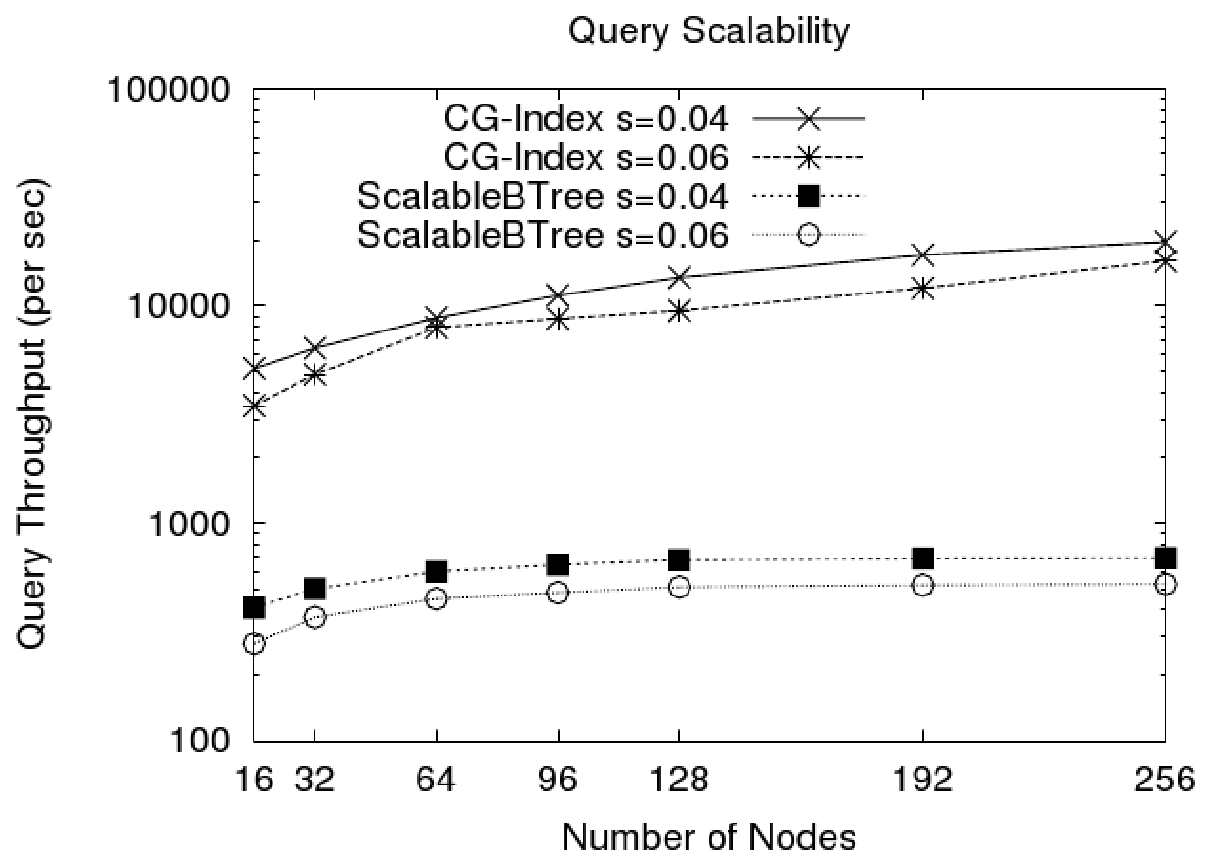
- Skew in generated data by zipfian law

# Test 1 – query throughput



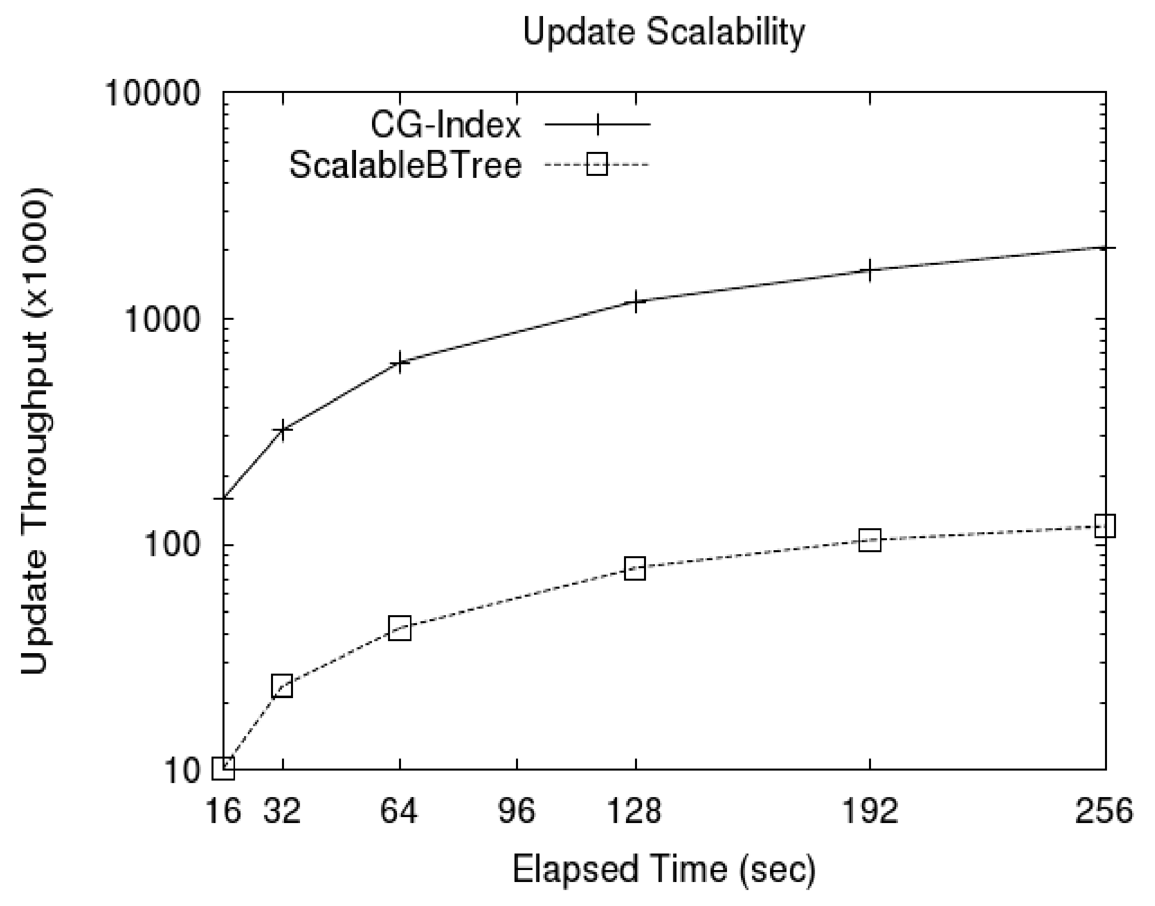
- Exact query (s=0) fastest
- Greater range (s) slower, because more nodes involved

# Test 2 – scalability



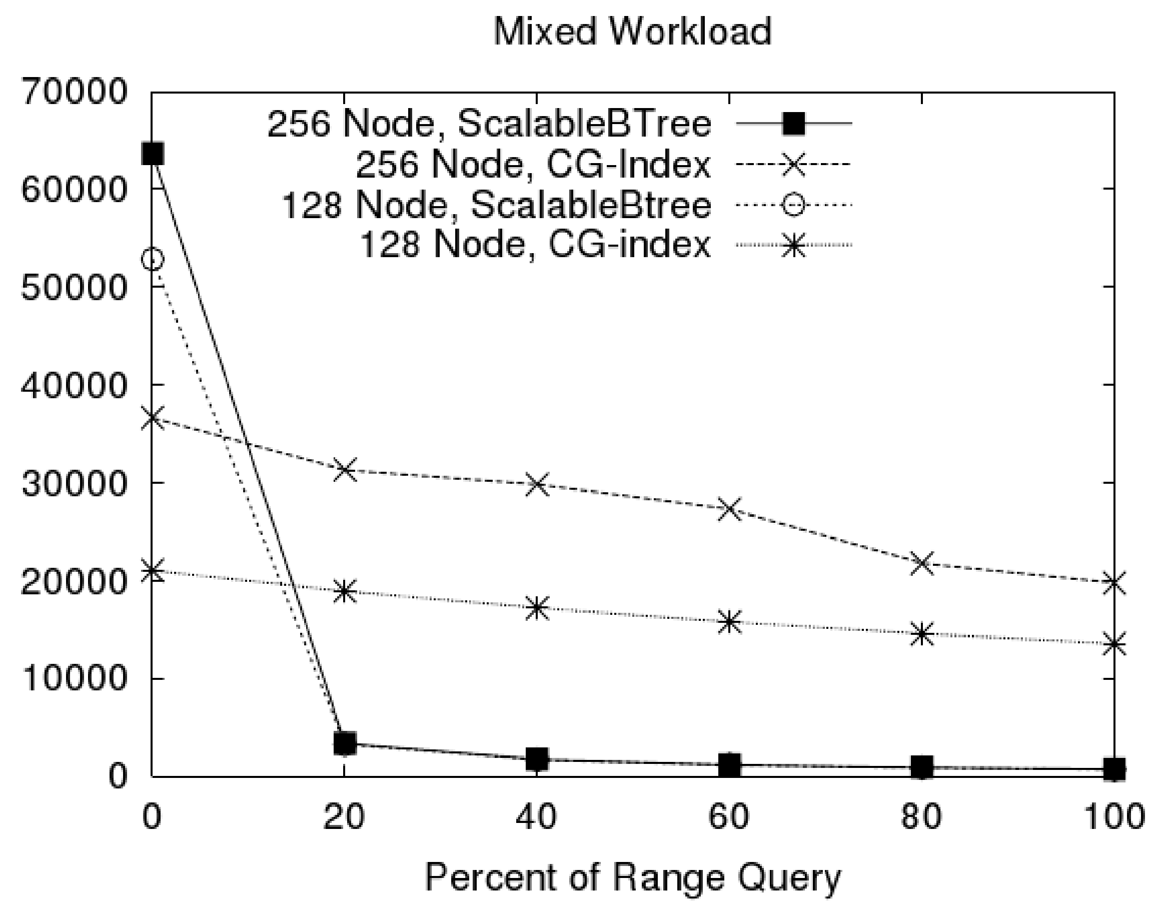
- CG scales almost linear
- SBT only until certain number of nodes
- Overall performance of CG a lot better

# Test 3 – updates



- Overall performance of CG better
- CG broadcasts only some updates to index
- SBT broadcasts every update

# Test 4 – mixed workload

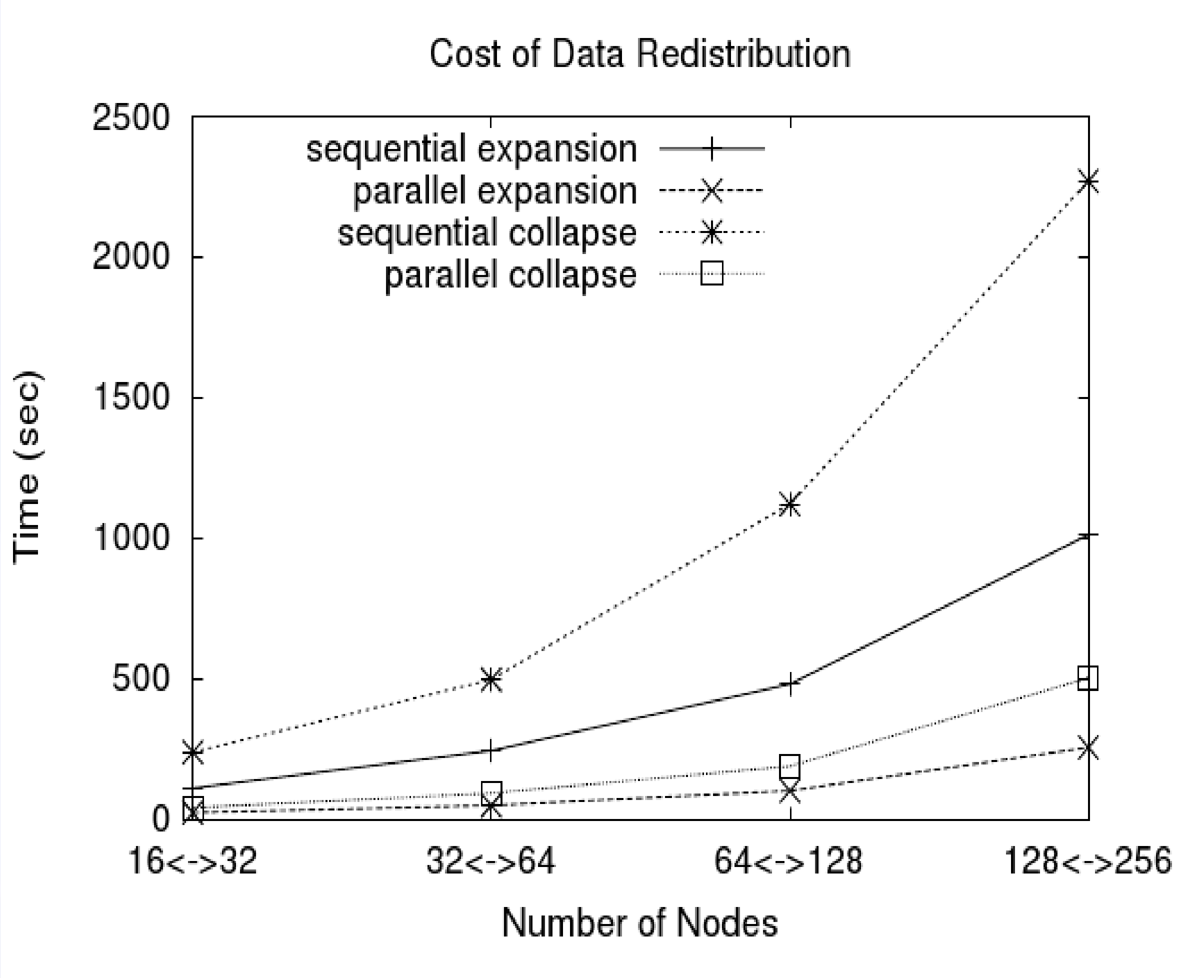


- SBT better for point queries

- CG needs several hops because of BATIN

- Pay-off for range queries

# Test 5 – flexibility



- Parallel
 (all nodes join at one time)
vs. sequential
(nodes join one after another)

- Good overall performance

- Parallel faster than sequential

- Expansion faster than collapse

# Missing parts

- Missing tests
    - Evaluation against Hashing approach
    - Flexibility of other approaches
    - Only few tests with point queries

… maybe because CG performed worse in these fields...?

- How is the BATON tree behaving on a split / merge of an indexed B+ - node ?

→ the authors even refused this question on the conference...

# Outline

- Motivation
- Solutions: creation & usage
- Solutions: maintenance
- Tuning
- Evaluation
- **Conclusion**

# Conclusion

- Secondary indices are often needed
- Current solutions delay updates or do not scale

➡ Presented a decentralized solution, using
**B+ - trees locally** and
**BATON on top** of these trees

→ Efficient updates and direct availability

→ Good scalability

→ Good performance for range queries

→ Weaknesses on point queries