The HV-tree: a Memory Hierarchy Aware Version Index

Rui Zhang
University of Melbourne

Martin Stradling
University of Melbourne

What is Versioned data

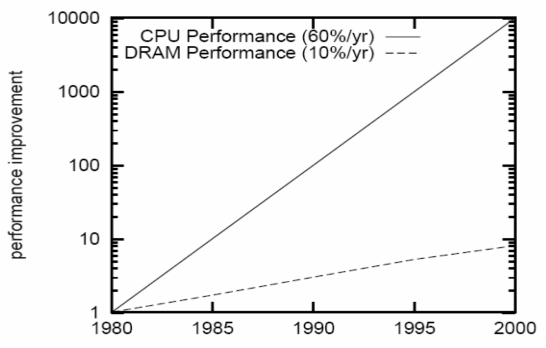
- Data describing objects that
  - has some attributes that change over time
  - we want to keep track of ALL the changes
  - (implicitly) has an attribute that never changes: identifier
- E.g.
  - Bank account: account number, balance
  - Stock: stock number, price
  - Wikipedia entry: name, contents
  - Sales record: item id, sales of a certain day
  - Star positions: star name, location
- Every changed value is called a “version” and has a “version number”
- “Versioned data” also called “temporal data”, and “version number” becomes “timestamp”

What to do on Versioned Data

- Point query (query with single key and single time)
  - What was the position of star #1234 on 1 Jan 2010?
- (Key-slice) Time-range query
  - Show the trajectory of star #1234 between 1 Jan 2010 and today.
- (Time-slice) Key-range query
  - Show the positions of all the stars on 1 Jan 2010
- Time-range key-range query (relatively rare)
  - Show the trajectories of the stars #1000~2000 between 1 Jan 2010 and today

Why do we Care about it Now

- Very large versioned databases
  - Sales record: Wal-Mart's data warehouse was 70 TB in 2001
  - Star positions: The Sloan Digital Sky Survey project receives 70 gigabytes of images every night
  - Existing version indexes do not scale well



- Improvement on Hardware
  - CPU/cache speed doubles every two years
  - Memory size increases at a similar rate
  - Existing version indexes do not take advantage of main memory techniques

Outline

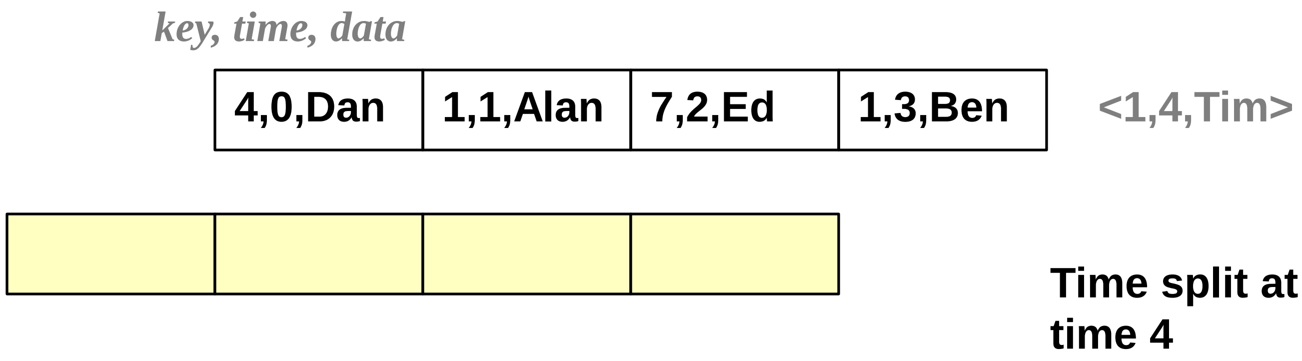
- Existing Work
  - Version indexes, especially TSB-tree
  - Main memory techniques
- Design for multiple levels of memory hierarchy
  - Principles
  - Straightforward approaches
- HV-tree
- Experimental results

Existing Version Indexes

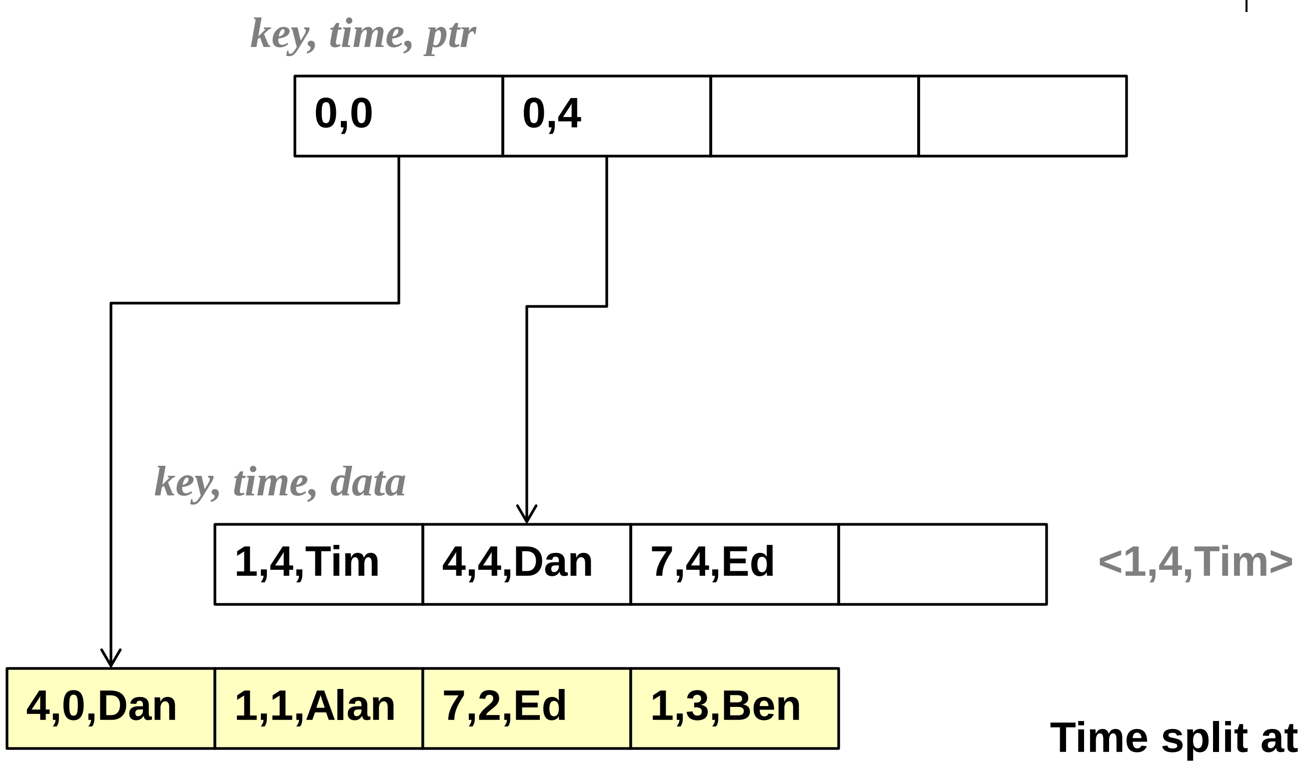
- Ones that move new data to new nodes
  - Write-once B-tree
  - Multi-version B-tree

- One that moves old data to new nodes
  - Time Split B-tree (TSB-tree)
  - Unique feature of progressively migrate old data to a new medium – a larger medium like hard disk or tape
    - Leaving current data on high-speed medium like the main memory

Time Split B-tree (TSB-tree)

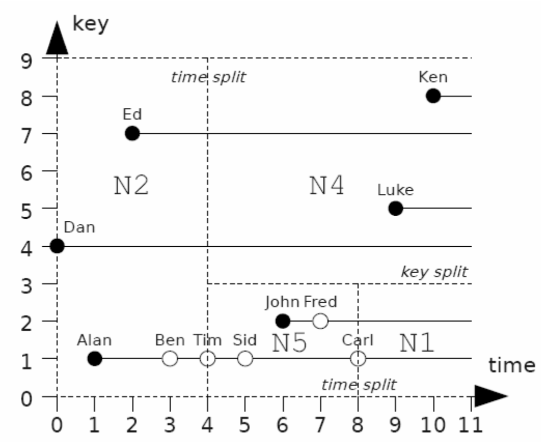


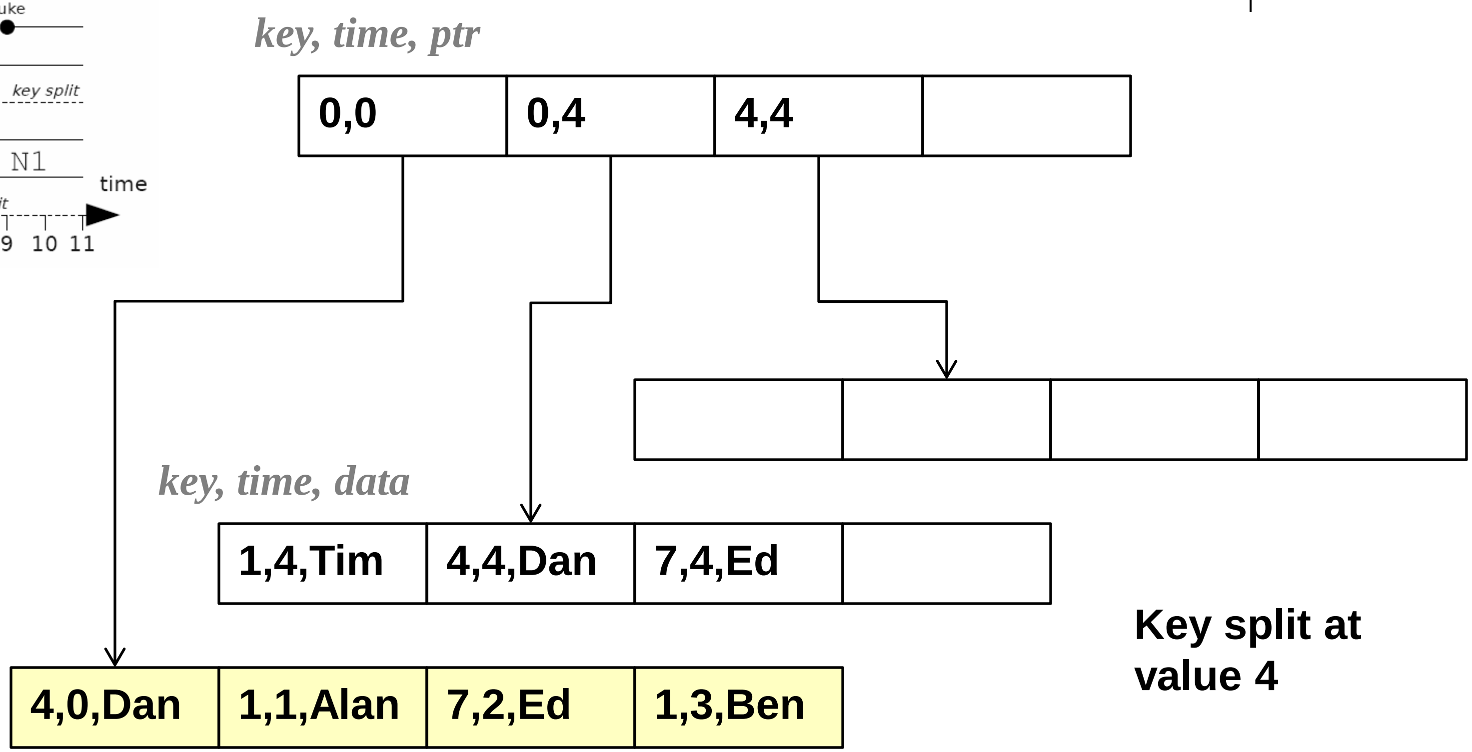
Time Split B-tree (TSB-tree)



Time split at time 4

Time Split B-tree (TSB-tree)





Key split at value 4

- Time split or key split depends on the portion of current entries in the node β: key split if β is greater than a threshold T
- Search: follow key-time range

Main Memory Indexing Techniques

- Key techniques
  - Aligning node size with block size
    - CSS-tree, CSB+-tree: use the cache block size (typically 32B or 64B) as tree node size
    - Later study shows that the actual optimal node size for the CSB+-tree is much larger than the cache block size
    - We assume the optimal node size S_{cache} is known
  - Pointer elimination
    - Hard to apply to a complicated structure like TSB-tree

Design for Multiple Levels of Memory Hierarchy

- Facts: big latency difference between adjacent levels of memories in the hierarchy, usually 1000 times.
- Principles
  - Tailor the index's structure to suits the characteristics of each level of the memory hierarchy.
  - Keep as much as possible frequently accessed data in higher levels of the memory hierarchy.

Straightforward Adaptions of TSB-tree

- TSB-small
  - use S_{cache} as the node size
  - let the operation system deal with caching and paging
  - Worse than TSB-tree because of bad paging behavior

- TSB-cond
  - use S_{cache} as the node size initially
  - expand/condense to node of size S_{disk} as historical pages are created and moved to disk
  - Worse than TSB-tree because of overhead caused by condensation

The HV-tree: memory Hierarchy aware Version tree

- Node size adjustable to the level of the memory the node resides in (Principle 1)
  - Key: Gradual change of size

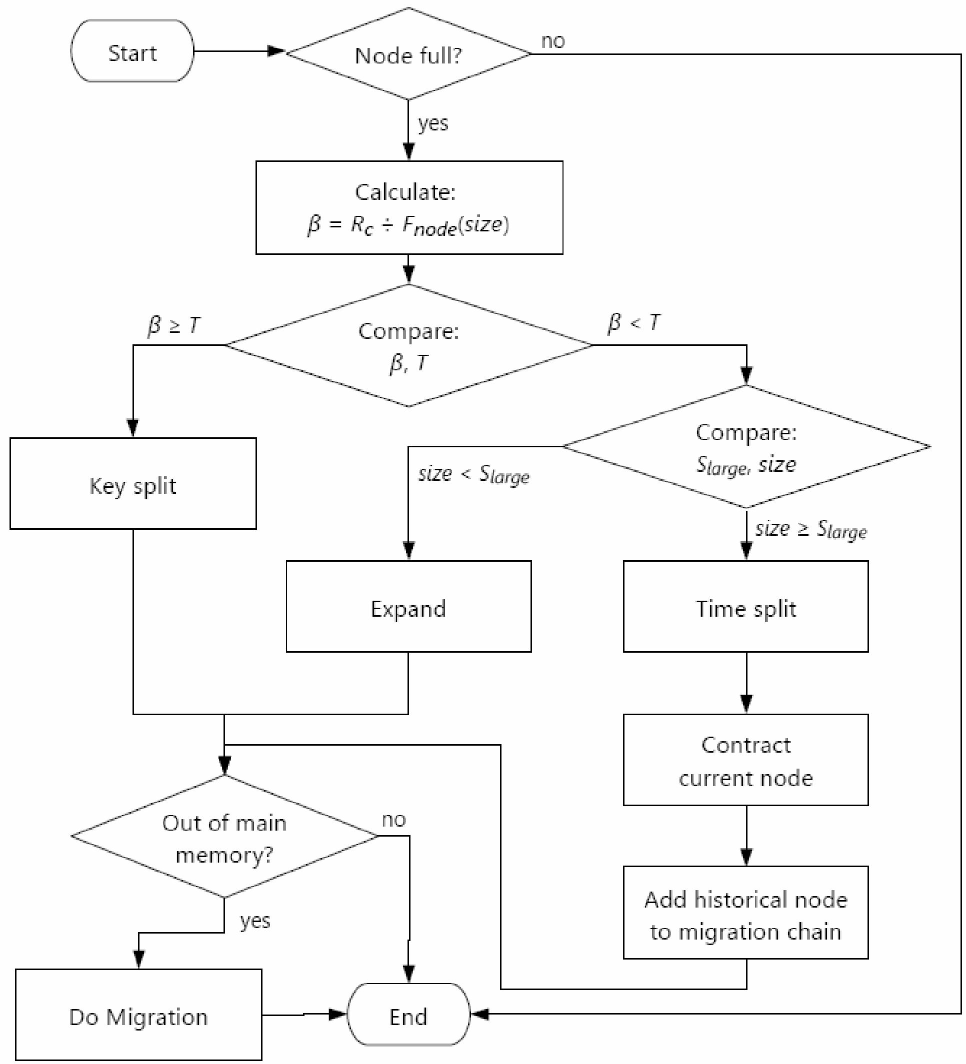
- Delayed data migration (Principle 2)

The HV-tree Structure

- Allowable node sizes
  - S_{cache} , $2S_{cache}$, … , S_{disk} (S_{disk} is a power of 2 times S_{cache})
  - E.g. S_{cache} *=1K,* S_{cache} *=4K,* so allowable node sizes are: *1K, 2K, 4K.*
- Some additional pointers maintained for data migration

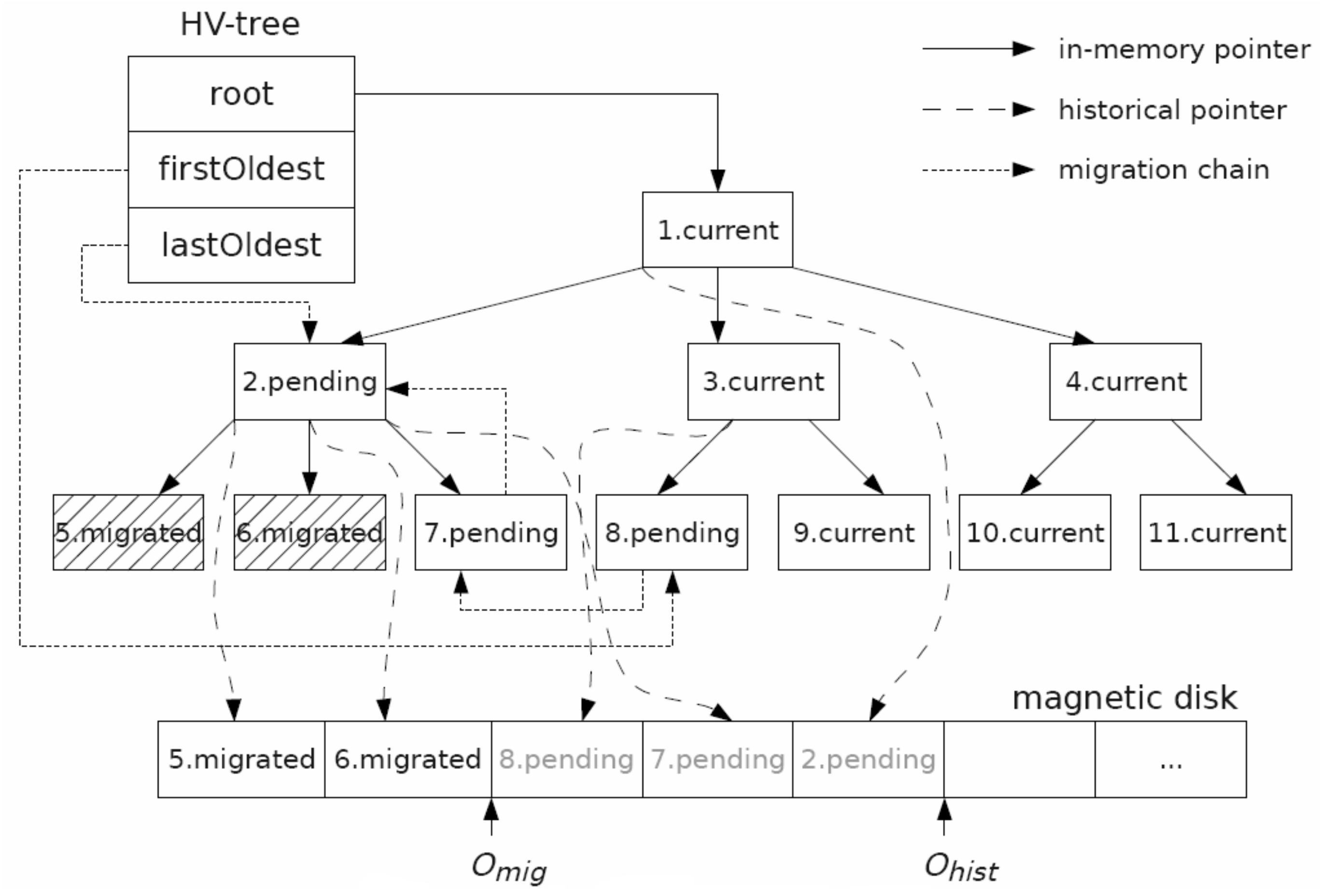
The HV-tree Insertion

- Start with the smallest allowable node size
- When node is full
  - key split
  - time split
  - or node expansion
- Choice of T
  - S_{cache} / S_{disk}



HV-tree Data Migration

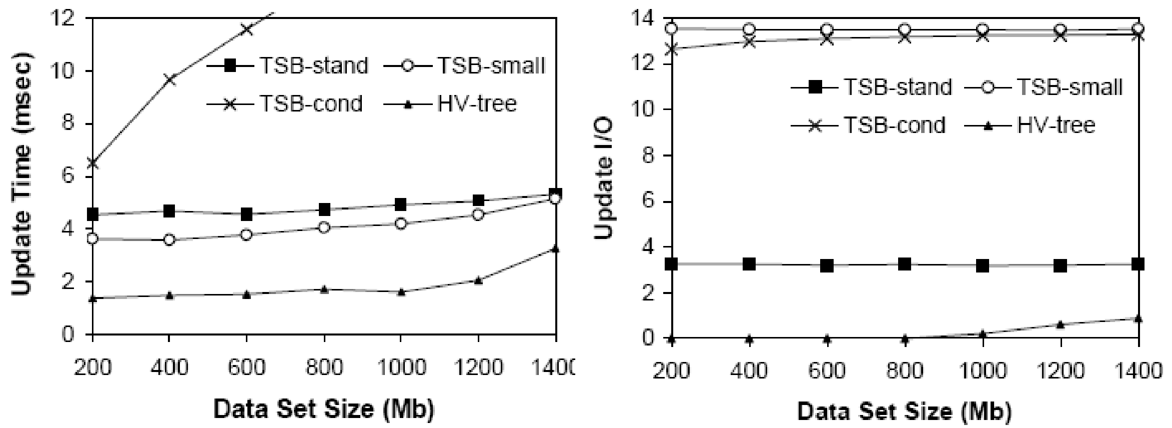
- Upon creation of a historical node
  - Do not move to disk immediately
  - Added to a *Migration Chain*
  - Migrate when out of memory



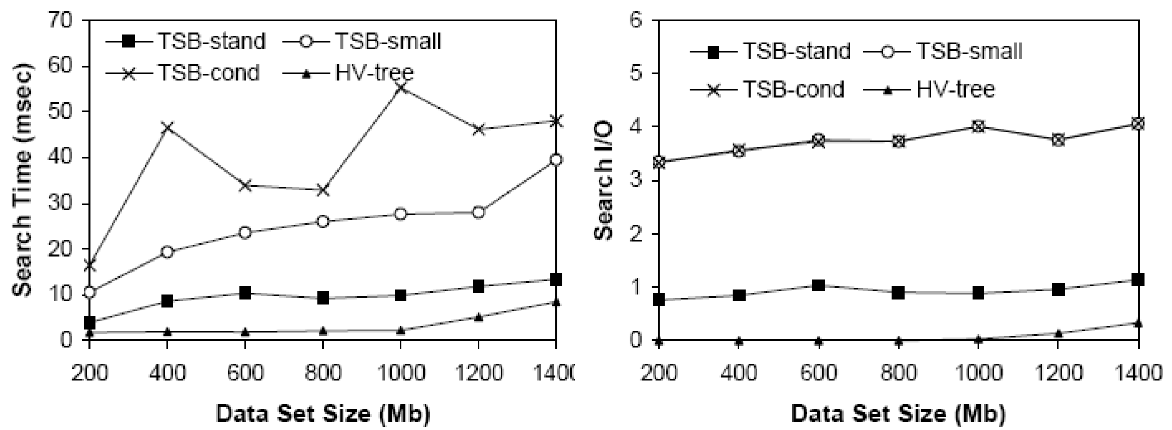
Experimental Setup

- Generated datasets with updates, search and mixed workloads
- Sizes: 500MB, 1000MB
- Queries follow Zipfian distribution, with varying skewness
- Hardware:
  - 3GHz CPU, 1GB memory, 80GB disk
  - L1 cache: <8K, 64B, 1>, L2 cache: <512KB, 64B, 8>
  - $S_{cache}=1K$, $S_{cache}=4K$

Results: Updates and Point Queries



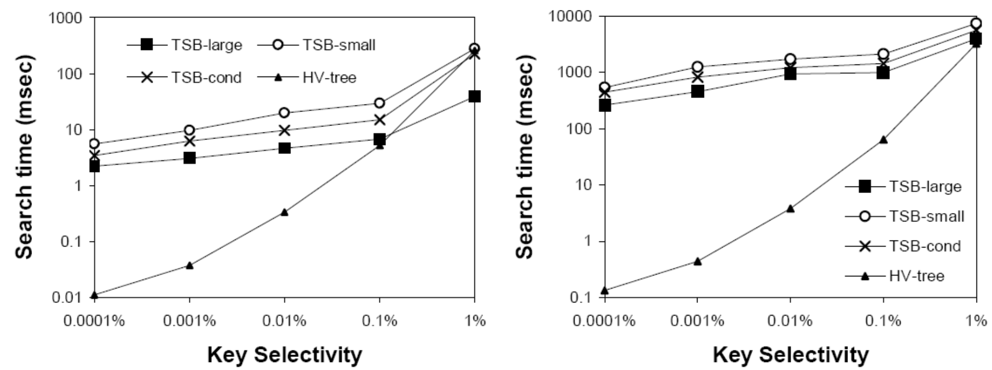
(a) Update



(b) Search

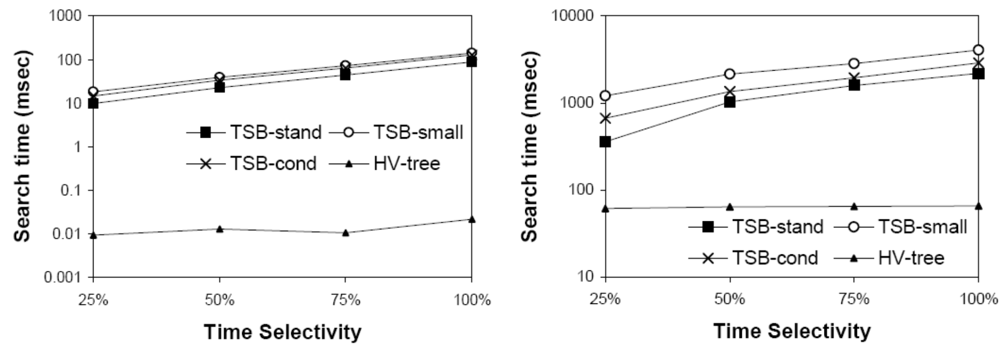
Results: Key-Range Queries, Time-Range Queries

- Key-range queries



(a) Time-slice (b) Time-range

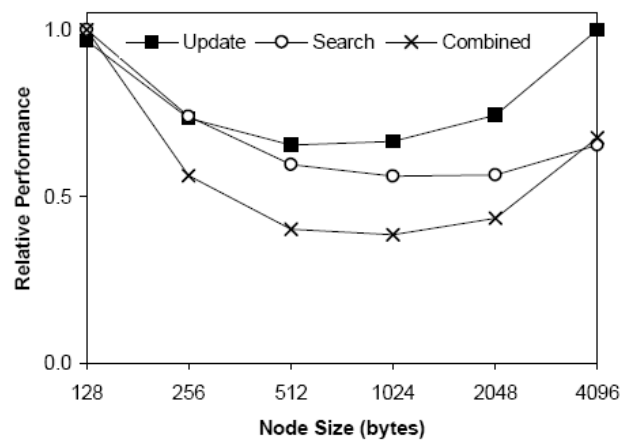
- Time-range queries



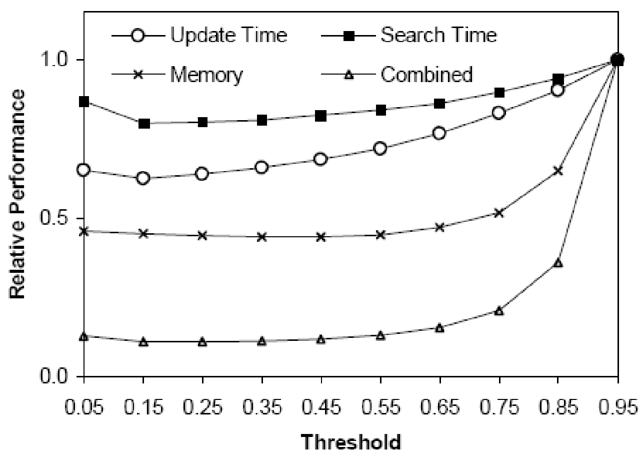
(a) Single key (b) Key-range

Experiments

- Finding S_{cache}



- Validation of T



Conclusions and Future Work

- First index design optimizing performance for multiple levels of memory hierarchy, achieving a highly scalable and efficient version index.

- Key techniques
  - Difference node sizes
  - Gradual change of node sizes
  - Data migration chain

- Performance
  - Several times faster for updates and point queries
  - 1000 times faster for key/time range queries

- Future work
  - Other data structures
  - Multi-core machine