Datenbanksysteme II: MDIS on Mordern Hardware; BB Tree

Ulf Leser

Content of this Lecture

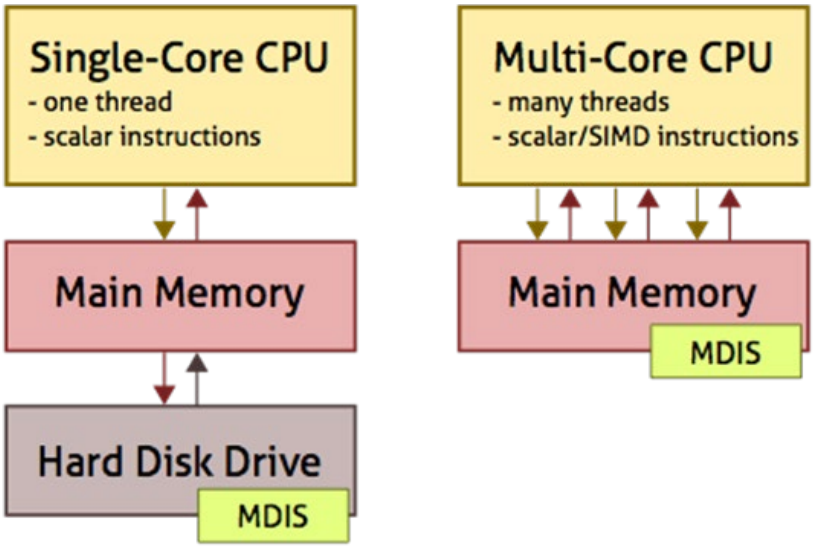
- MDIS On Modern Hardware
  - Competitor
  - Evaluation
- BB-Tree

Scan or Index?

- Selectivity of a query: % of points matching
- Selectivity of an index: % of blocks that must be touched
- Multi-dimensional range queries (MDRQ)
  - Select regions of spatially near blocks
  - To exploit access locality, MDIS try to map multi-dimensional spatial closeness to one-dimensional physical closeness
  - More dimensions – increasingly difficult
- Result: Scans outperformed only for selective queries
  - Classical paper 1998, IO based: 20%
  - IO is expensive – pruning pays off quickly
- Question: Behavior on today's hardware?

MDIS on Modern Hardware

- Main memory, multi-core, SIMD
  - Or even GPU, NVRAM, RDMA, FPGA, …
- Optimize disk block access -> Optimize mem. page access
  - CPU cache-lines, L1/2/3 caches
- Much research on one-dimensional main-memory IS
  - ART, FAST, CSSL, …
- But no previous work for MDIS



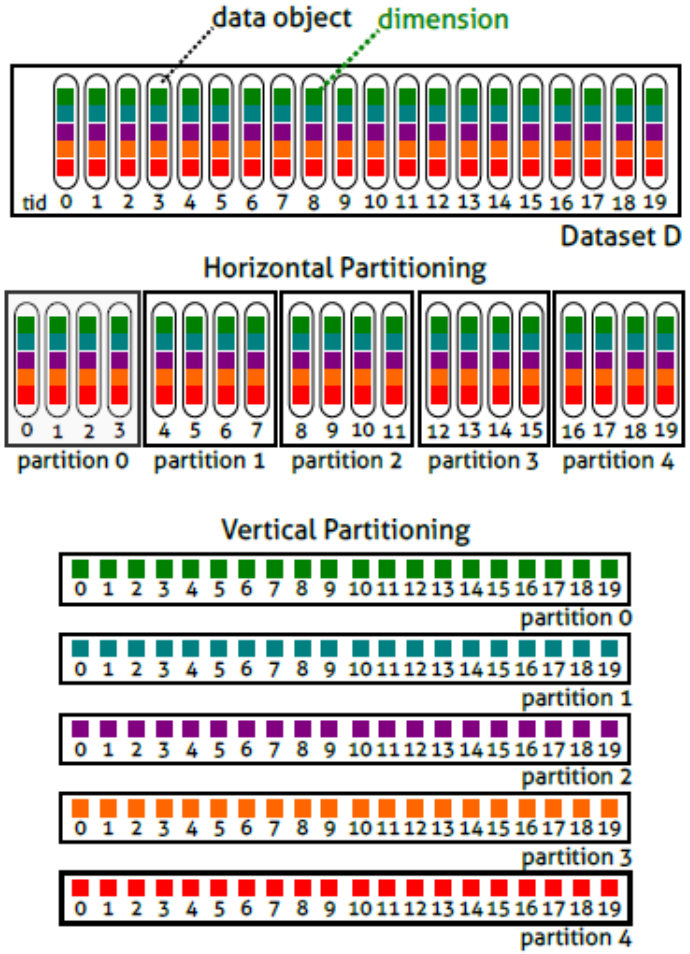
Adaptation to Main-Memory

- Conservative adaptations
  - Keep original architecture of MDIS
  - Reuse existing implementations when possible

- Scans: None, data kept in in-memory arrays
  - But different layouts for parallelization – next slide
- kd-trees: None (in-memory IS by design)
  - But we store leaves in blocks
- VA-files: Approximations and data blocks in memory
  - Very similar to partitioned hashing
- R*-trees: All kept in memory, block size = cache size
  - R*: Frequent deletion and re-insertion for optimized partitioning

Parallelization / Partitioning

- Horizontal (all MDIS)
  - Partition into subsets of tuples
  - One thread per subset
  - Pro: Load balancing
  - Con: Scans inefficient in partial queries
  - Con: Less efficient SIMD (heterogeneous values)
- Vertical (only scan)
  - Each dimension one partition
  - One thread per dimension
  - SIMD: Compare k values of one dimension per instruction
  - Pro: Pruning in partial queries
  - Con: Load balancing



Content of this Lecture

- MDIS On Modern Hardware
  - MDIS Adaptions
  - Evaluation
- BB-Tree

Experimental setup

- Throughput measured using 1000 queries, warm cache
- Two different Intel CPUs
  - 24/12 threads, different SIMD width
- MDIS construction: Insert tuple-by-tuple in random order
- kd-Tree, VA file, scans: Own implementation
- R* from libspatialindex (block size adapted)

Dataset	Data Objects	Dimensions	Domain per Dimension (real numbers)	Distinct Values per Dimension	Raw Dataset Size (MB)
SYNT-UNI (uniform distribution)	10k	5	[0,1]	9,950 (avg)	0.19 MB
	100k	5	[0,1]	95,175 (avg)	1.91 MB
	1M	5-100	[0,1]	632,257 (avg)	19.07 MB - 381.47 MB
	10M	5	[0,1]	999,956 (avg)	190.74 MB
SYNT-CLUST (with clusters)	1M	5	[0,1]	632,047 (avg)	19.07 MB
POWER	10k	3	[2556001,2566000]; [12857,17281]; [14142,19278]	10,000; 627; 698	0.11 MB
	100k	3	[2556001,2656002]; [12466,18247]; [13698,20395]	100,000; 2,089; 2,290	1.14 MB
	1M	3	[2556001,3556003]; [12466,18770]; [13698,20704]	1,000,000; 4,325; 4670	11.44 MB
	10M	3	[2,9875683]; [12282,24623]; [13281,26879]	9,875,681; 6,840; 7,634	114.44 MB
GMRQB	10M	19	Our website provides a detailed description of all properties of the data of GMRQB.		724.79 MB

Genomic Multidimensional Range Query Benchmark

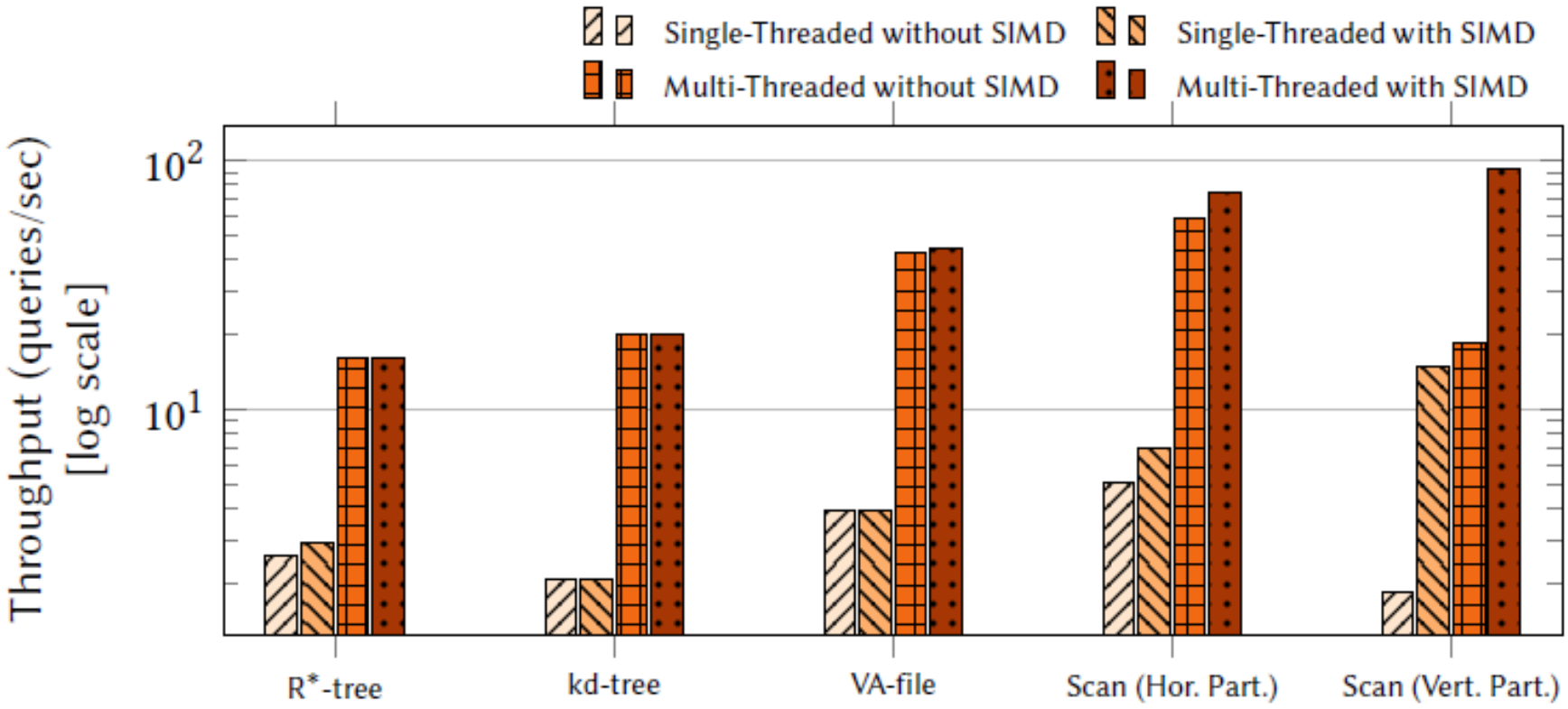
- Data from 1000 genomes project (2504 genomes)
- App. 10 Million variants, 19 dimensions
- 8 typical parameterized query templates
- Parameters set to randomly selected gene locations
- 7 our of 8 templates are partial queries

GMRQB Query Template	Average Selectivity	Average # of Queried Dimensions
Query Template 1	10.76% (σ = 7.24%)	2 (σ = 0.0)
Query Template 2	2.19% (σ = 2.27%)	5 (σ = 0.0)
Query Template 3	5.36% (σ = 3.61%)	3 (σ = 0.0)
Query Template 4	0.22% (σ = 0.15%)	4 (σ = 0.0)
Query Template 5	0.20% (σ = 0.15%)	5 (σ = 0.0)
Query Template 6	0.11% (σ = 0.11%)	6 (σ = 0.0)
Query Template 7	0.05% (σ = 0.06%)	7 (σ = 0.0)
Query Template 8	0.00001% (σ = 0.00002%)	19 (σ = 0.0)
Mixed Workload	1.58% (σ = 3.58%)	5.81 (σ = 4.11)

Table 1: GMRQB query templates.

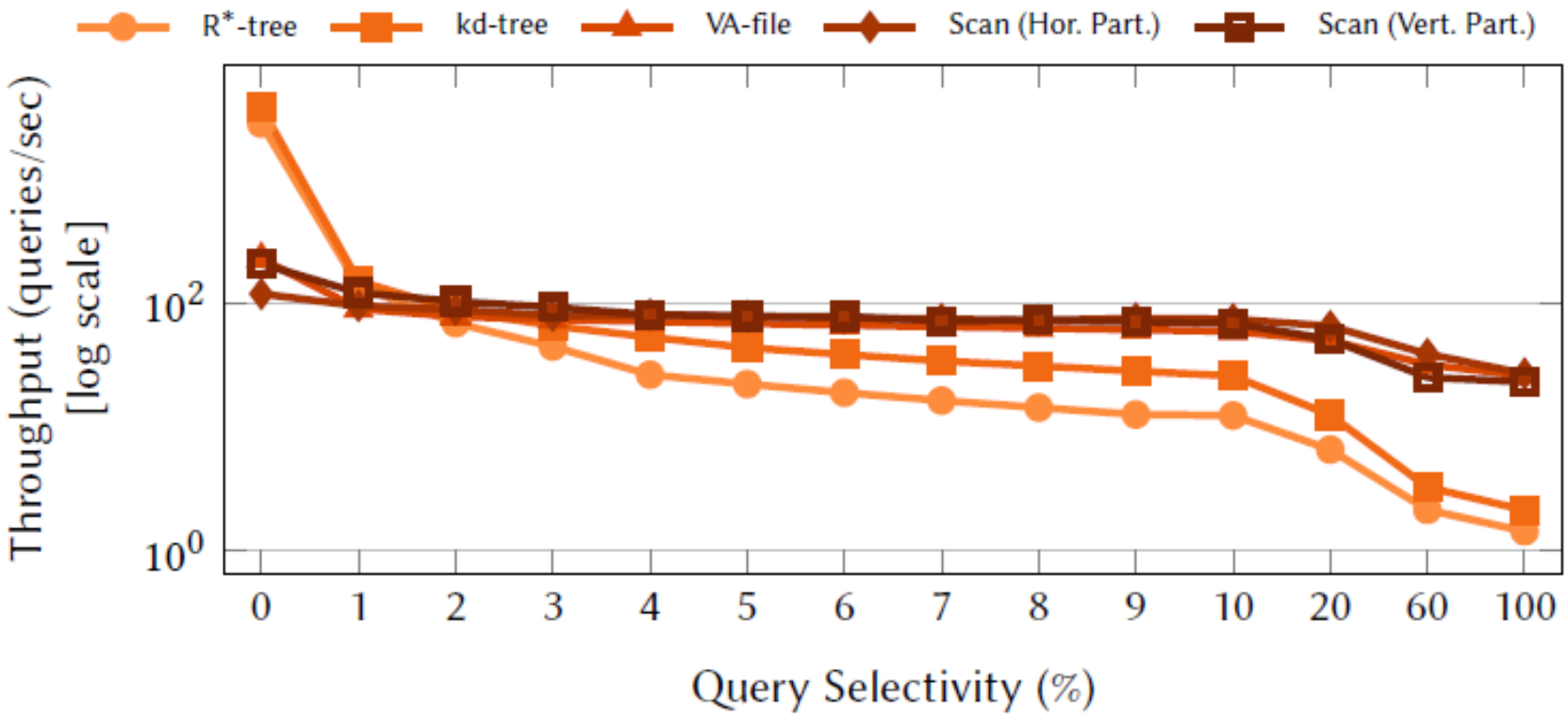
```
SELECT * FROM variations
  WHERE chromosome = 5
    AND location BETWEEN 100000 AND 1000000
    AND quality BETWEEN 10 AND 100
    AND depth BETWEEN 10 AND 1000
    AND allele_freq BETWEEN 0.5 AND 1;
```

Result: SIMD only Worth for (Vertical) Scans



Synthetic data, d=20, n=1E6, uniform, sel=0,1%

Result: Scans Hard to Beat – even at 1% Selectivity



Synthetic data, d=5, n=1E6, uniform

Vertical scans affected by dimensionality (large intermediate results)

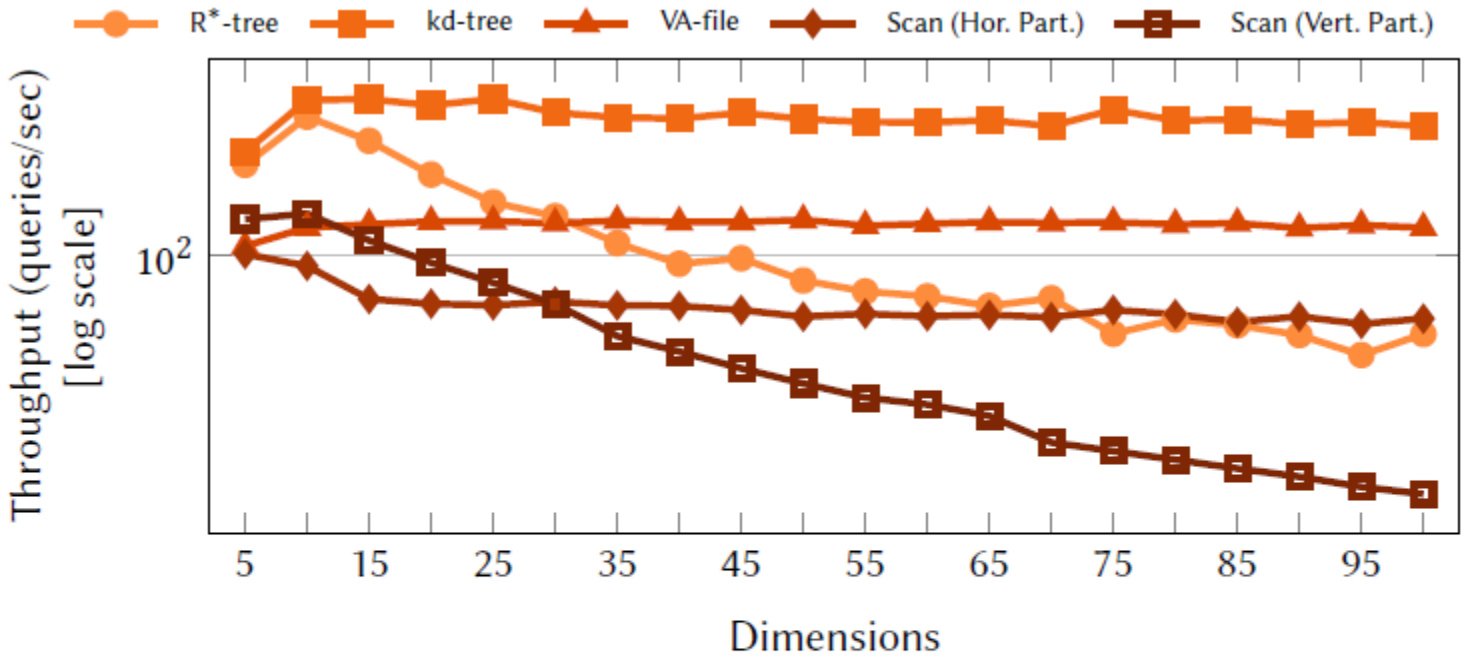


Figure 5: Throughput when executing range queries with an average selectivity of 0.4% (five dimensions) to 0.0002% (> ten dimensions) on 1 Million uniformly distributed data objects using 24 software threads depending on dimensionality.

Scans excel in real life data even at 1% sel. and even with PM queries

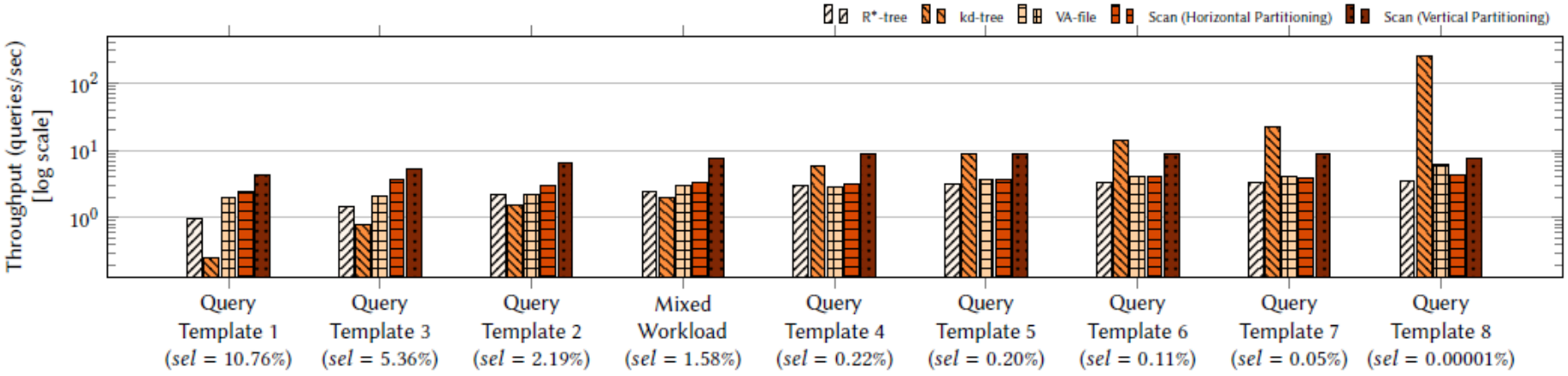


Figure 10: Throughput of contestants when executing the GMRQB with varying selectivities on 10 Million 19-dimensional data objects from the 1000 Genomes Project dataset using 24 software threads (query templates are ordered by selectivity).

Summary

- kd-Tree > R* > scans > VA-File for highly selective queries
- Scans > VA-File > kd-Tree > R* for less selective queries
  - VA almost never better than scan – yet more complex
  - kd-Tree outperforms R* trees
  - For box queries, horizontal partitioning is beneficial
  - For partial queries, vertical partitioning is superior
- Traditional MDIS faster in main memory for highly selective queries – but gains are small, admin costs are high, more difficult to parallelize, …
  - Same observations for single dimension IS (e.g. [9])

Content of this Lecture

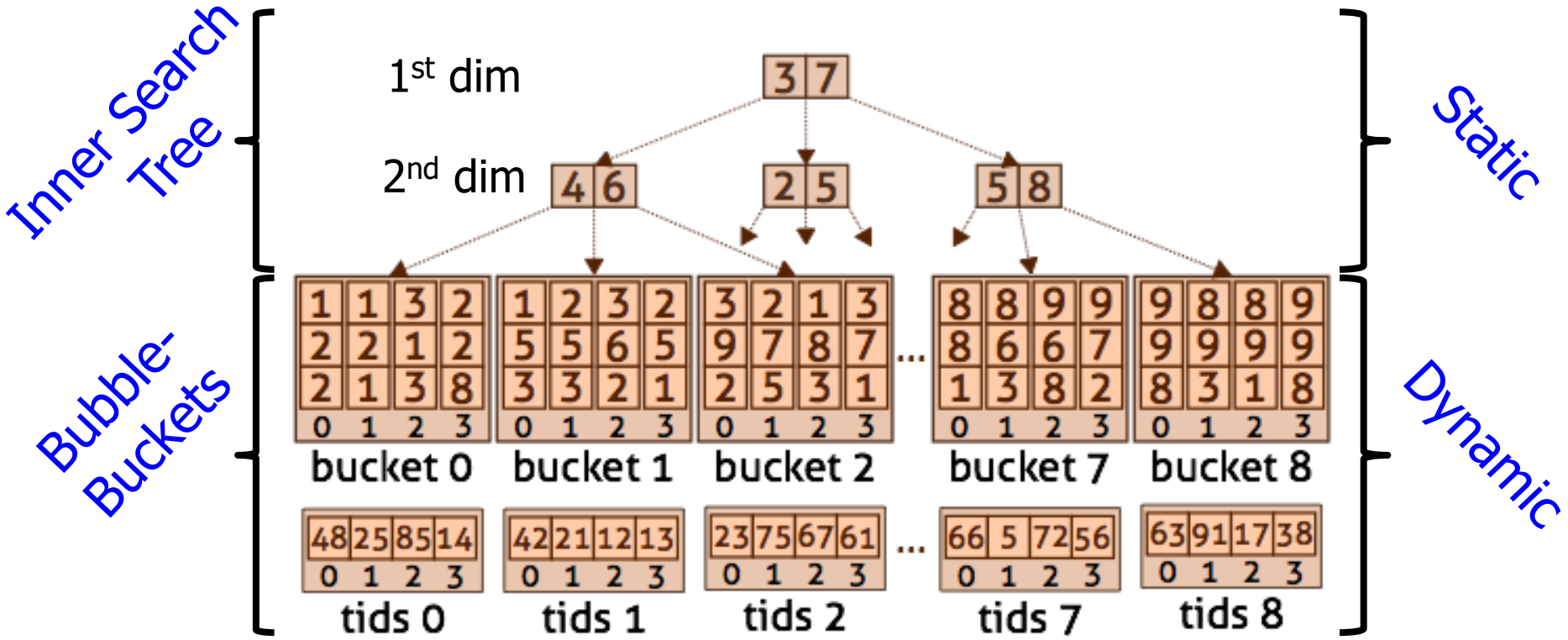
- MDIS On Modern Hardware
  - Competitors
  - Evaluation
- BB-Tree
  - Motivation
  - BB-Tree Structure
  - Evaluation

BB-Trees from 10.000 Feet

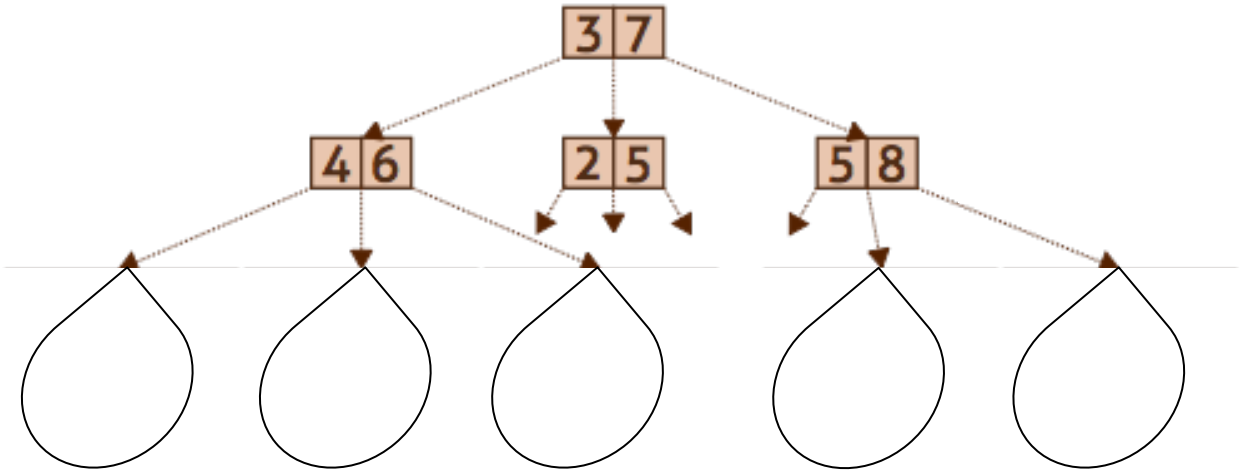
- Almost-balanced k-ary search tree
- Optimized for cache hierarchies of modern CPUs
- Elastic leaf nodes (bubble buckets)
- Updatable
- Efficient handling of low-cardinality dimensions
- Multi-threaded variant

- No free lunch:
  Optimized memory layout costs (infrequent) rebuilds

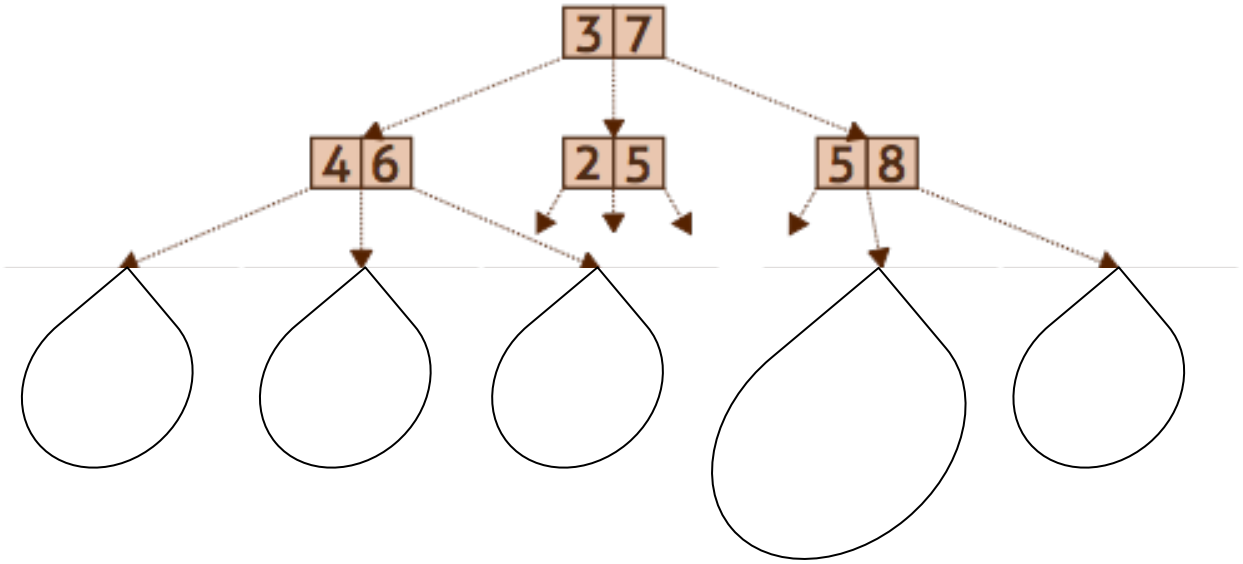
Data Layout: k-ary Search Tree



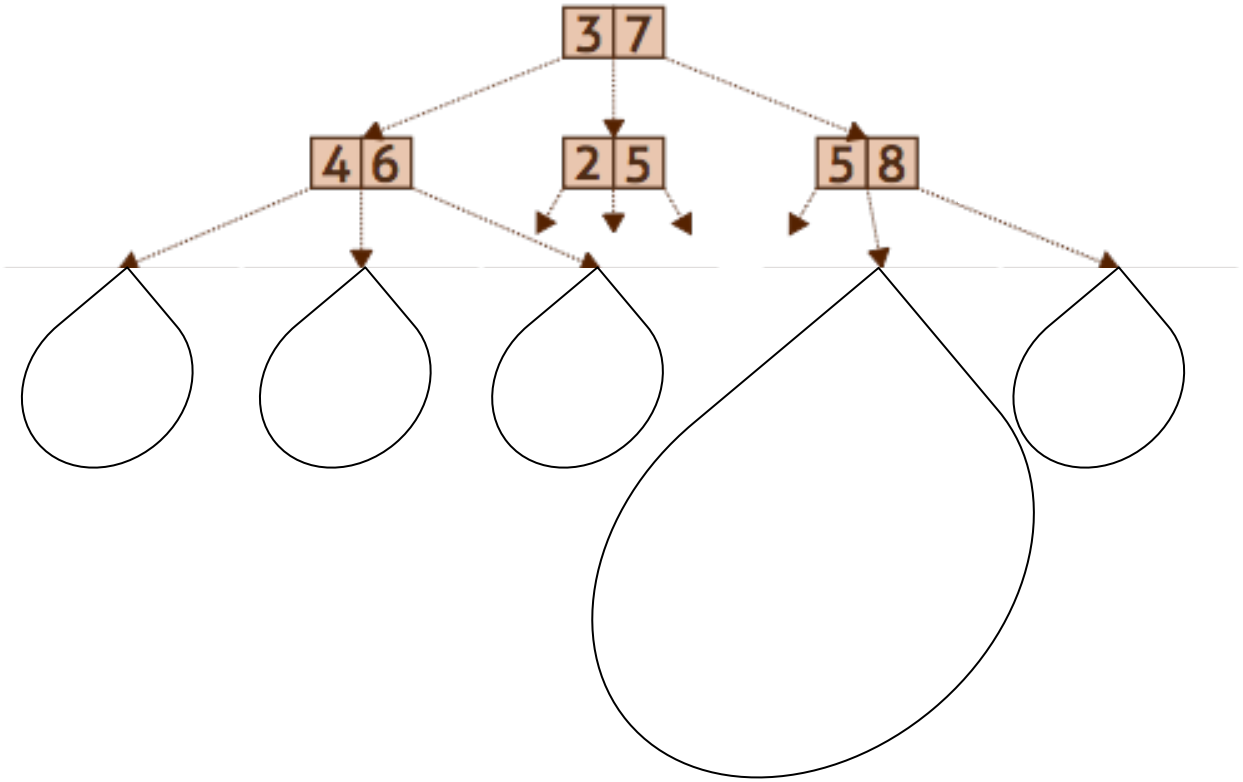
Growing and Shrinking



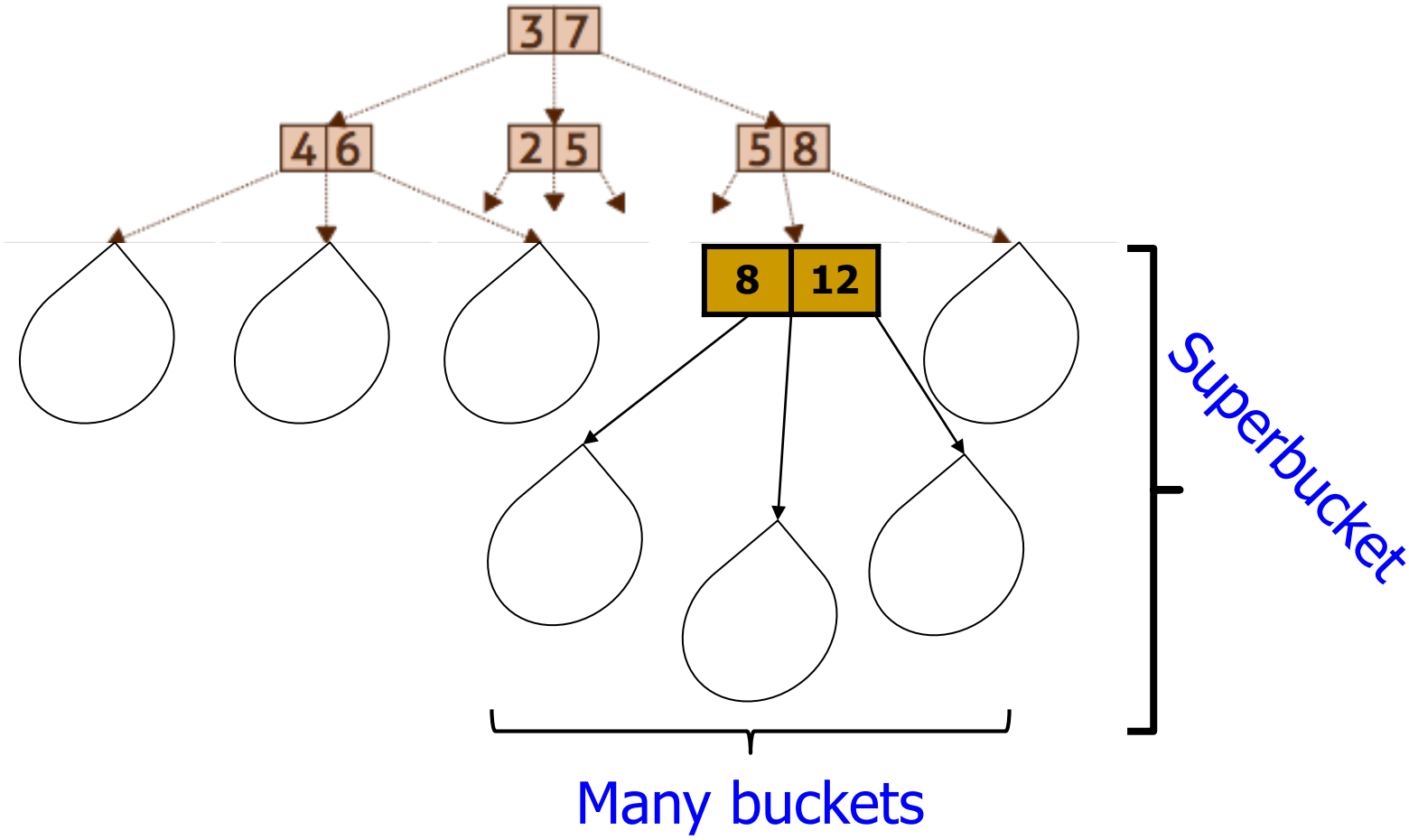
Growing and Shrinking



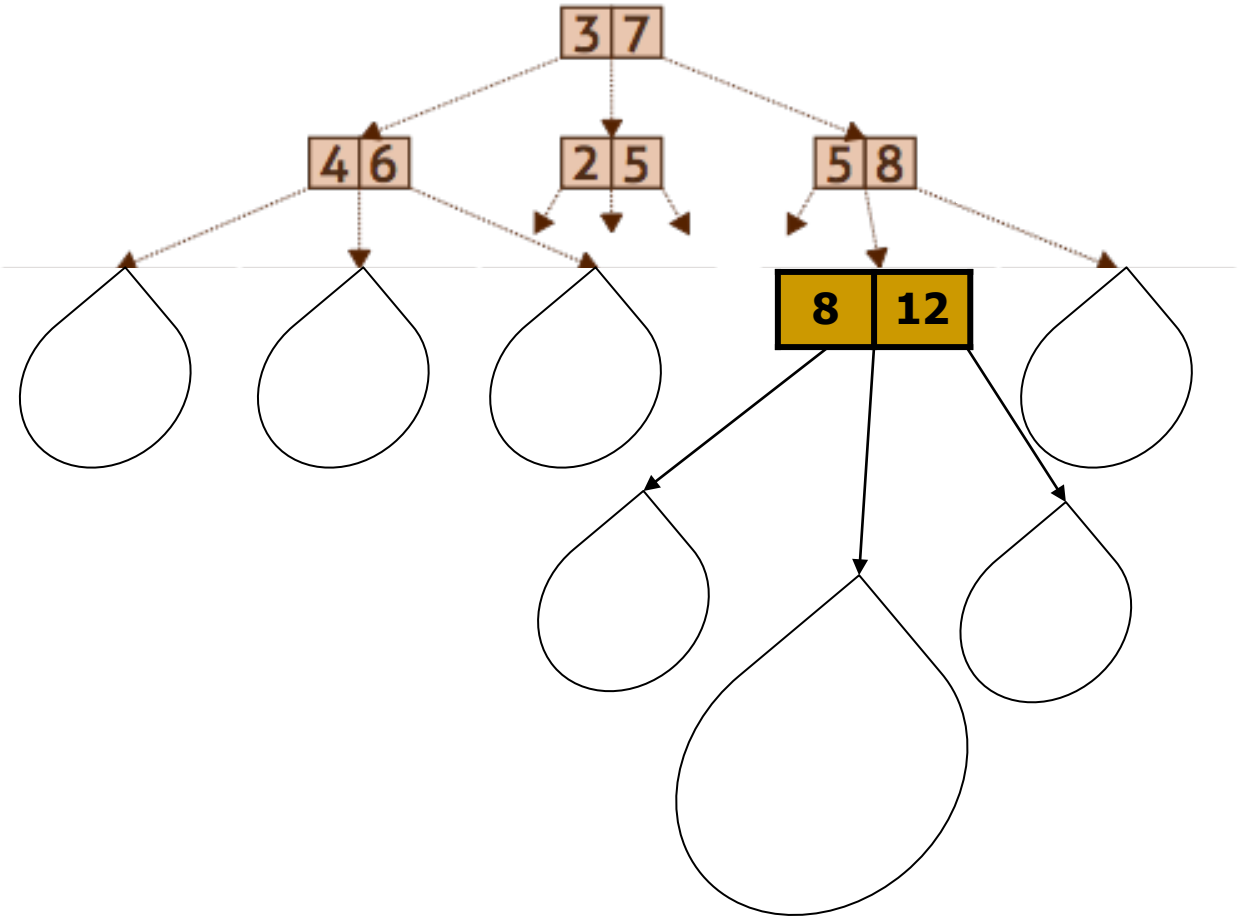
Growing and Shrinking



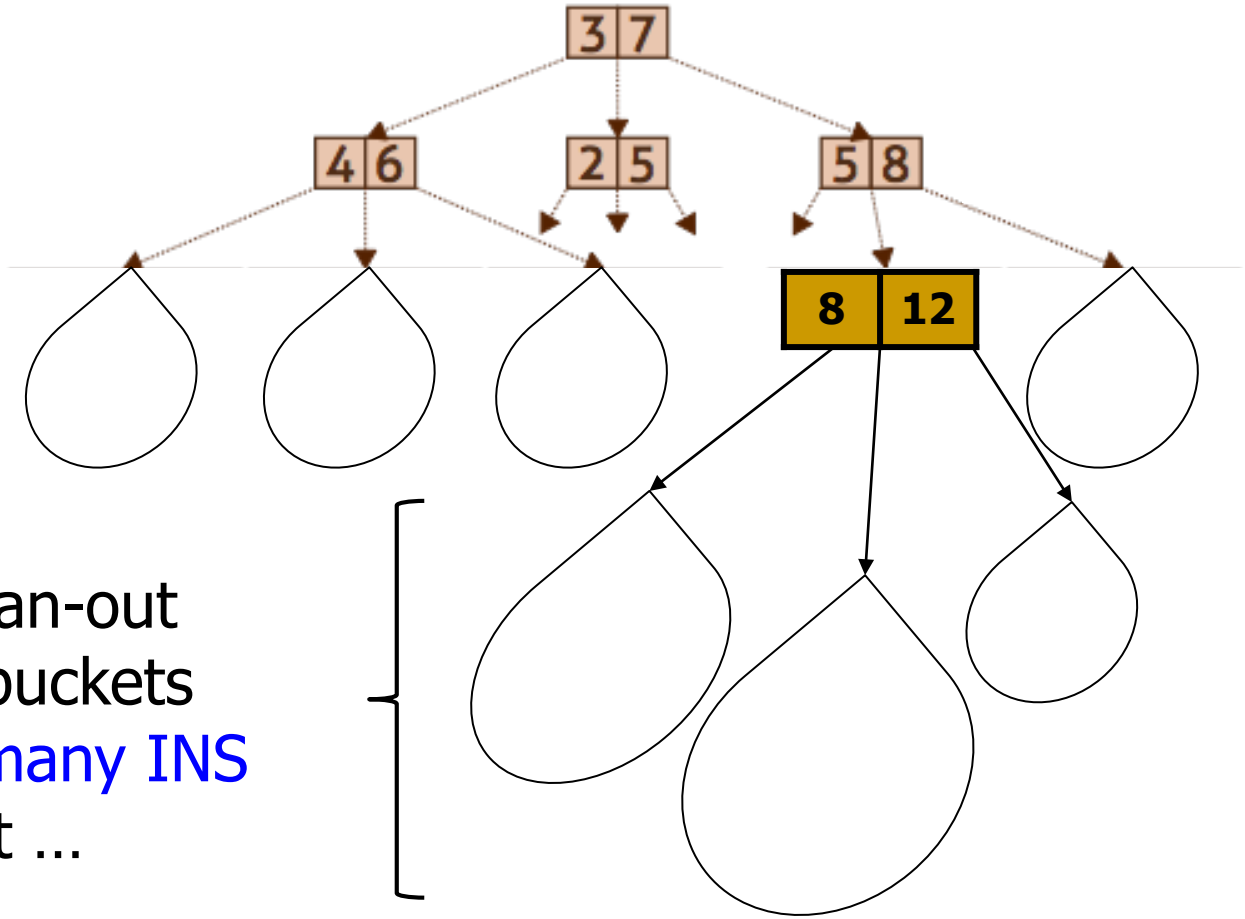
Growing and Shrinking



Growing and Shrinking

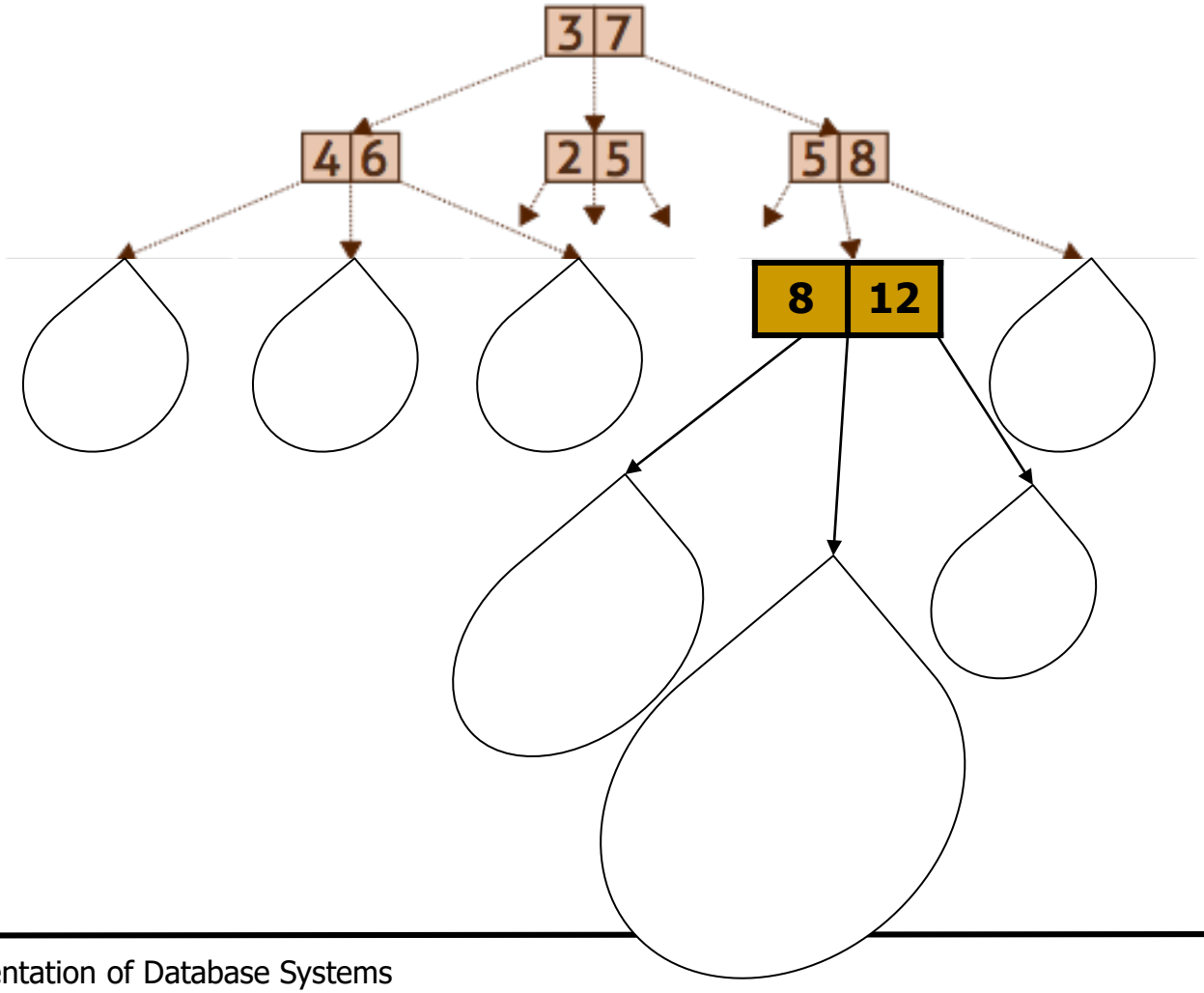


Growing and Shrinking

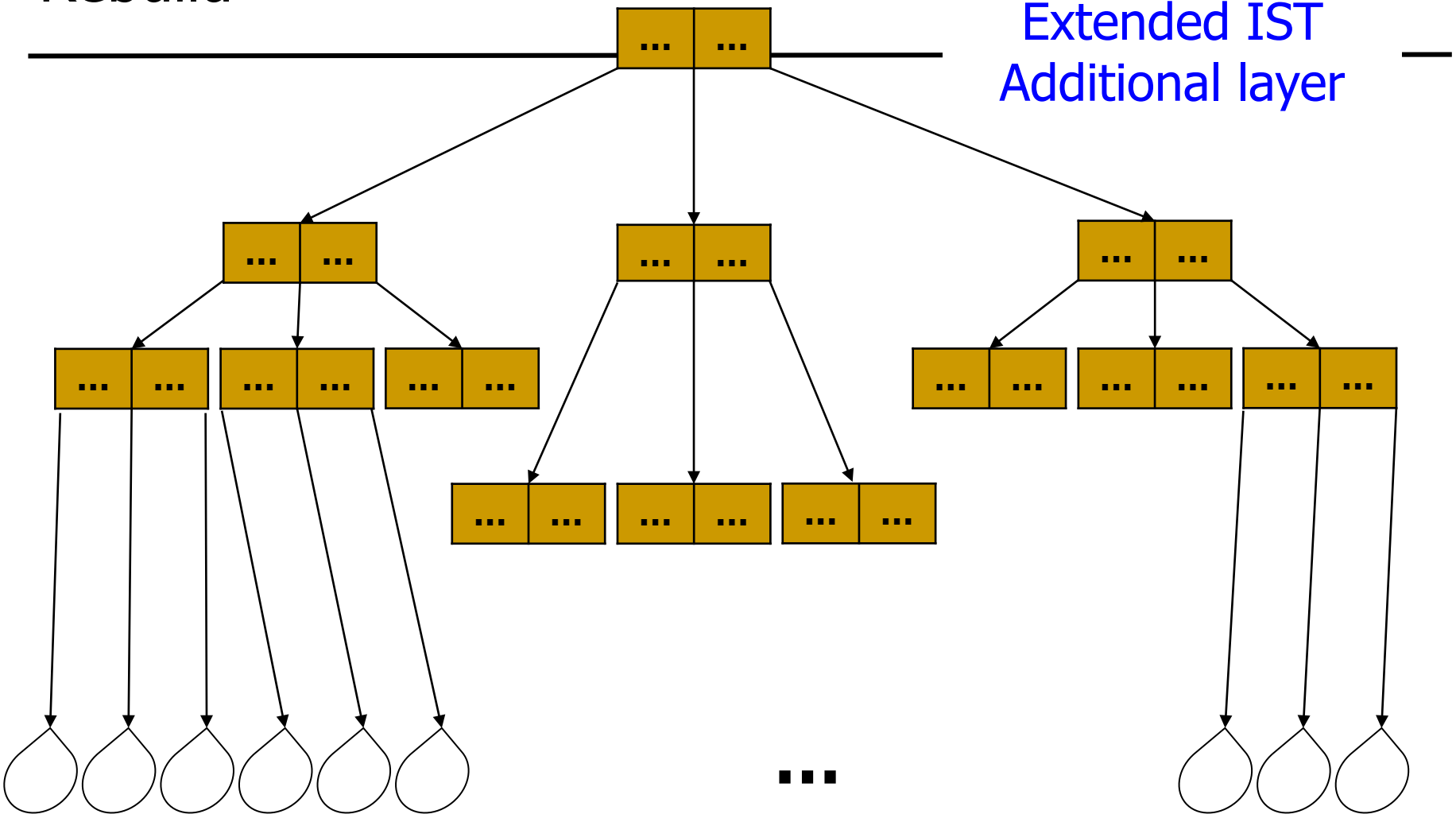


High fan-out
Many buckets
Buffers many INS
But …

Growing and Shrinking

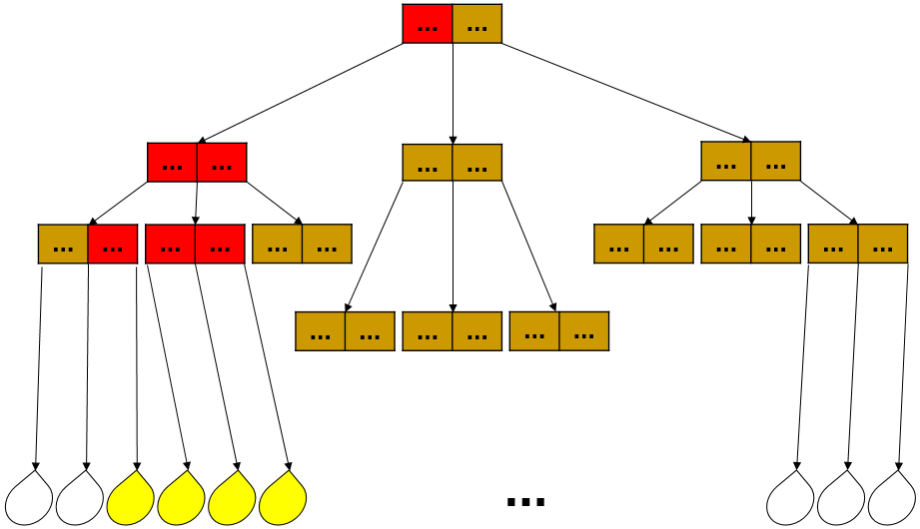


Rebuild

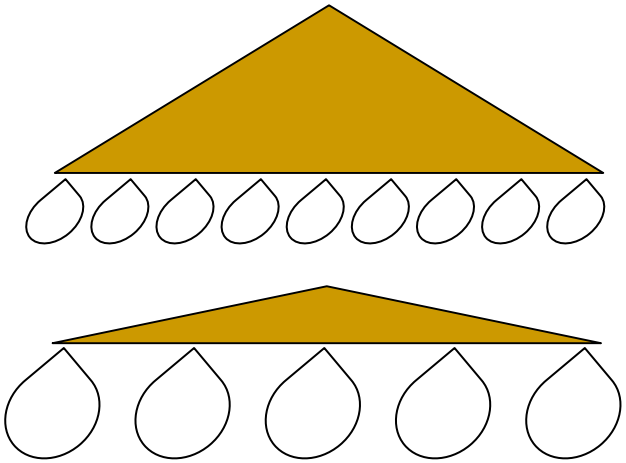


Searching the BB-Tree

- Phase I: Search IST
  - Range queries lead to multiple search paths
  - Partial match queries must scan entire levels
- Phase II: Scan buckets
  - Serial or parallel

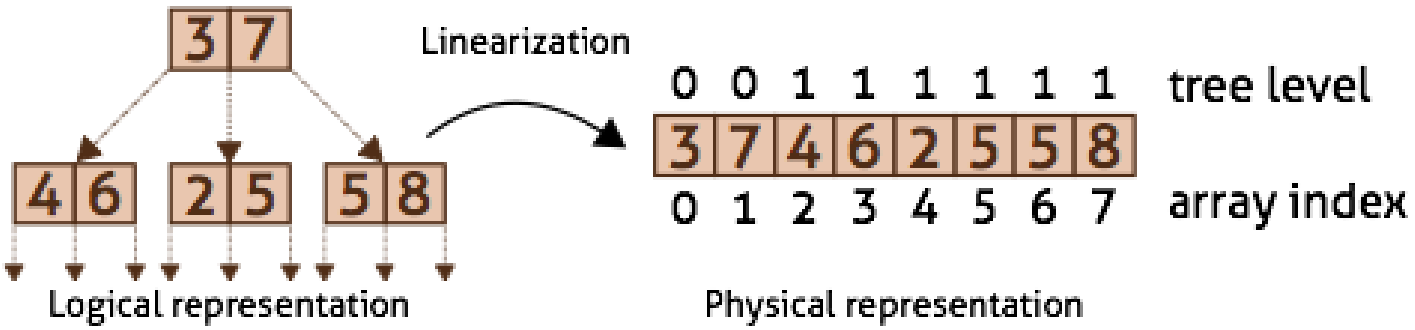


- Max-size of buckets: Trade-Off search / scan
  - Low selectivity queries: More scan
  - High selectivity queries: More search

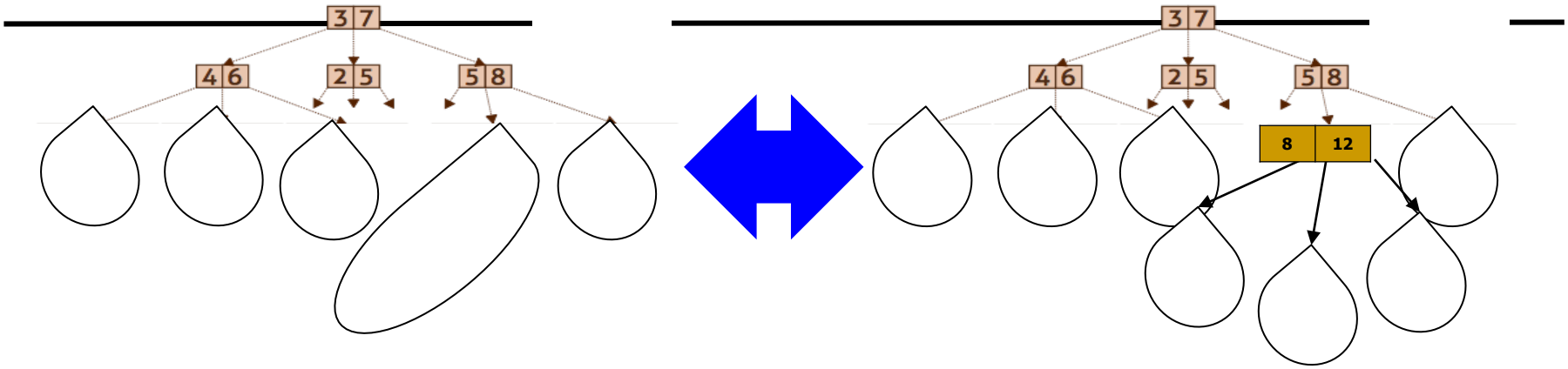


Inner Search Tree (IST)

- Linearized storage
  - IST mapped into static dense array (no growth/shrinkage)
  - No pointer chasing during traversal
- Fan-out (k) aligned to size of cache lines
  - Typically k=16 for INT values
  - High fan-out: Low tree, fast IST traversal
  - No cache-misses within IST node



Superbuckets



- Bubble buckets morph between different representations
  - Overflowing ordinary buckets turn into superbuckets
  - Underflowing superbuckets turn into ordinary buckets
  - Overflowing superbuckets trigger index rebuild
  - Underflowing ordinary buckets trigger index rebuild
- Superbuckets
  - Increase height by only 1 – virtually same search performance
  - High capacity (dep on k): Drastically reduce frequency of rebuilds
- Simple idea – quite some impact

Rebuilding the BB-Tree

- Four steps (bucket capacity fixed)
  - Determine required number of buckets – IST height
    - Leave some free space
    - Bubble buckets are dynamic arrays – still good space utilization
  - Sample at random (~10% of data) and compute dim. cardinalities
    - Most costly operation
  - Sort dimensions by cardinality (high – low)
    - Assumption: High cardinality dimensions have more selective queries
    - Low card-dimensions have little pruning power
  - Recursively determine delimiter values (in sample)
    - Such that k equal-size groups emerge (what if <k unique values?)
  - Build IST
  - Re-distribute objects into buckets

Content of this Lecture

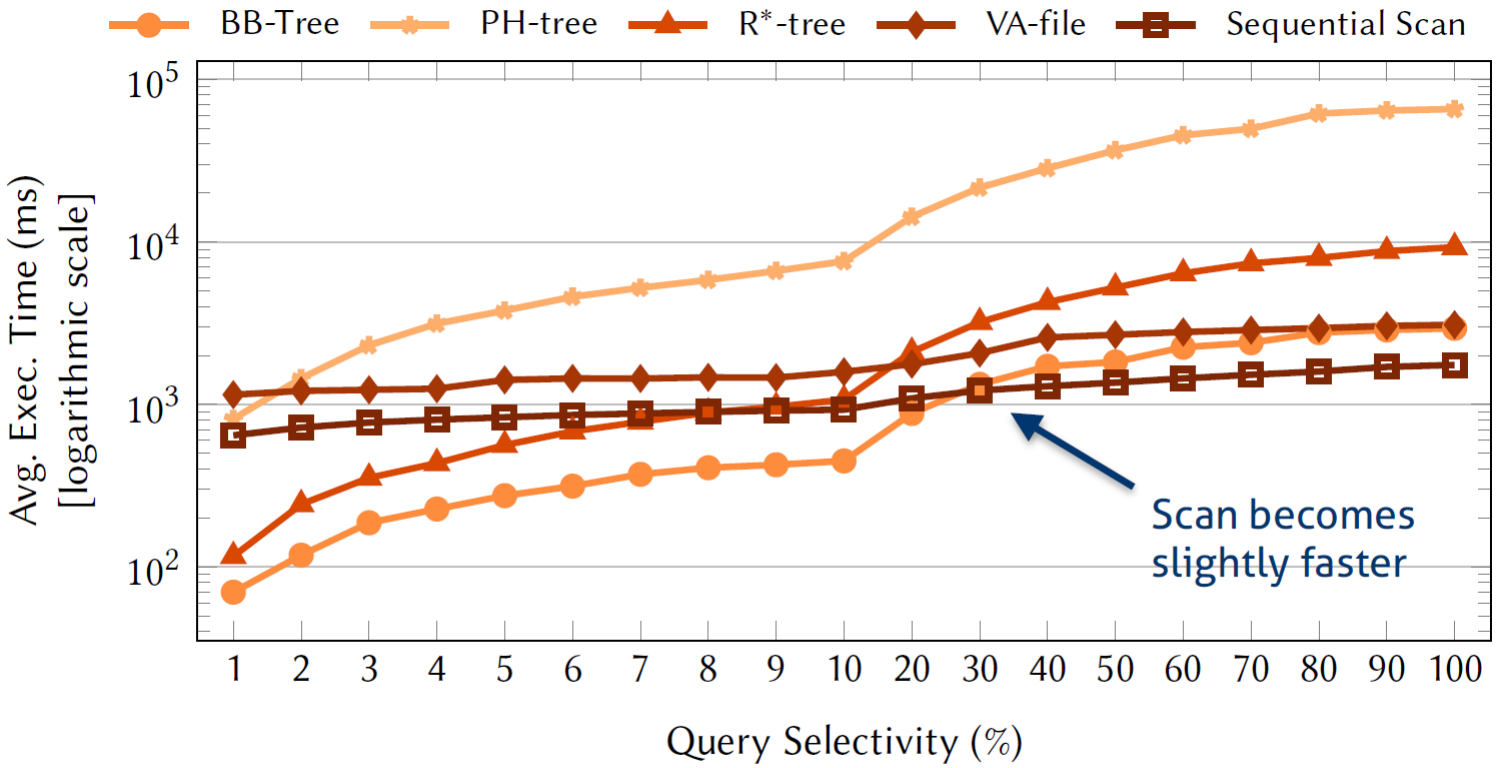
- MDIS On Modern Hardware
  - MDIS Adaptions
  - Evaluation
- BB-Tree
  - Motivation
  - BB-Tree Structure
  - Evaluation

Evaluation

- Four data sets
  - UNIFORM (synthetic data, 5 to 100 dimensions)
  - CLUSTERED (synthetic data, 5 dimensions)
  - POWER (real-world data, 3 dimensions)
  - GENOMIC (real-world data, 19 dimensions)
- Synthetic and realistic workloads, read-only and R/W
- Five competitors
  - kd-tree, PH-tree, VA-file, R*-tree, scans

Random Range Queries

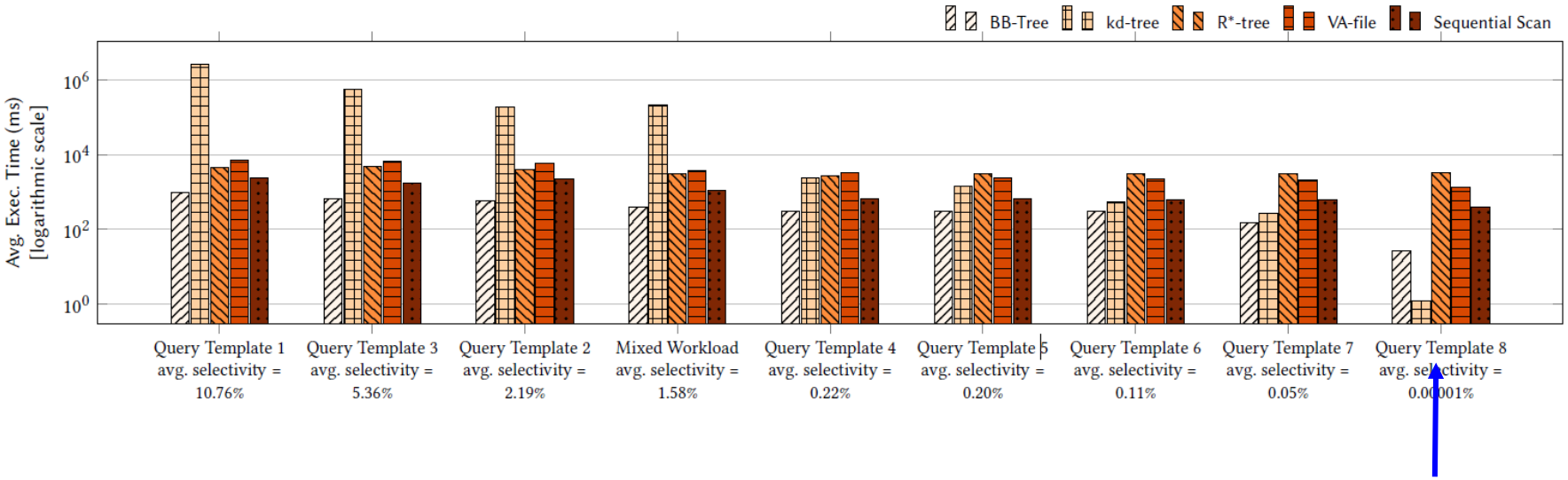
kd-Tree extremely slow for low selectivities
and never faster than BB-Tree



Synthetic range queries on 10 Million 5-dimensional data objects from UNIFORM.

Genomic Multidimensional Range Query Benchmark

Ph-Tree crashed on GMRQB



Essentially a point query

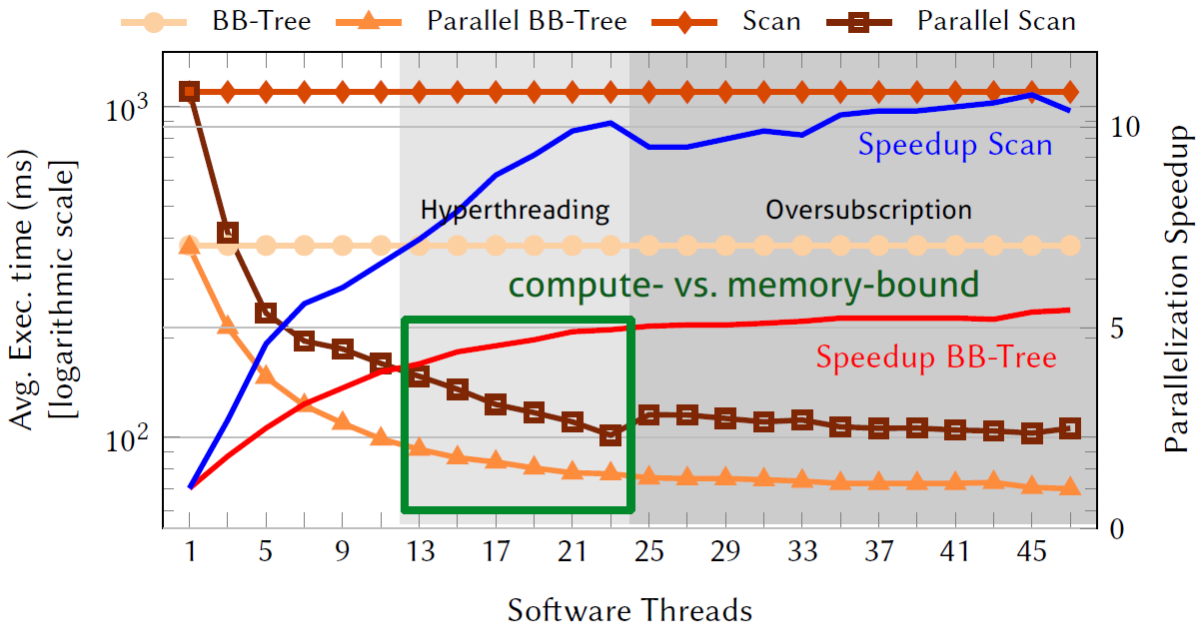
- Eight real-life query templates from genomics
  - Mostly partial-match
- Data from 1000genomes project, 10M points, 19 dim
- Sorted by average selectivity

Modern Hardware

	BB-Tree	kd-tree	PH-tree	R*-tree	VA-file	Scan
CPU Cycles	**164M**	8,306M	1,908M	252M	2,934M	1,582M
LLC Accesses	1.0M	824M	1.2M	2.5M	1.8M	**0.5M**
LLC Misses	0.7M	0.9M	0.8M	0.5M	1.6M	**0.3M**
TLB Misses	0.3M	1.0M	0.3M	0.3M	0.2M	**0.1M**
Branch Mispr.	**0.1M**	0.7M	3M	0.2M	10M	7M

Table 3: Performance counters per range query (1% selectivity;n=10M, m=5, UNIFORM).

Parallel BB-Tree



Realistic range queries (Mixed Workload from GMRQB, avg. sel.= 1.6%) on 10 Million 19-dimensional data objects from GENOMIC with varying # threads.

- Scan scales better (10x vs 5x)
  - Scan: Very few serial components
  - BB-Tree: Single-threaded IST search
- Hyper-threading offers little to BB-Tree

Insert / Delete / Mixed

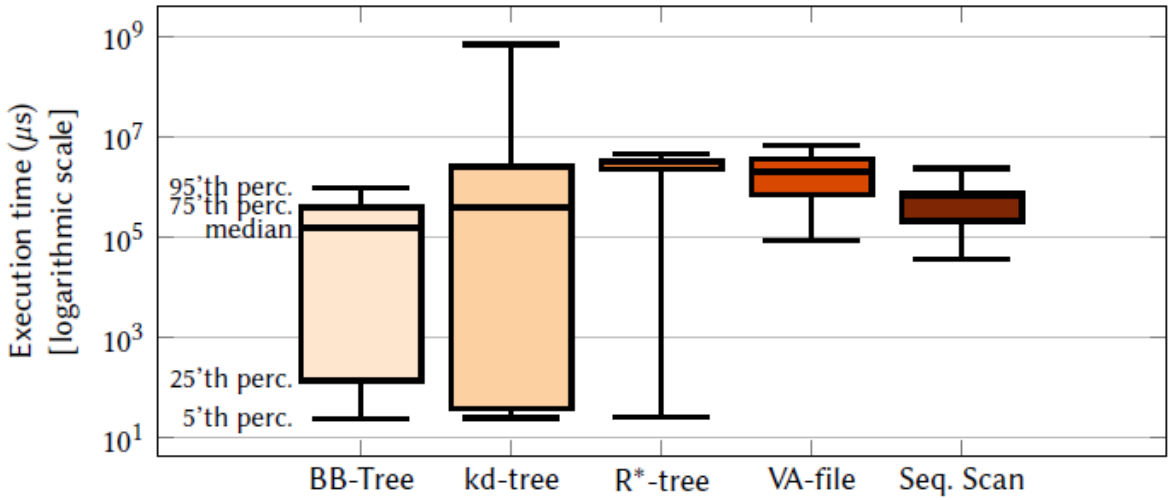


Figure 16: Execution times of single queries (inserts, deletes, exact-match and range queries) from a mixed workload in random order; bulk insert is not included; PH-tree ran out of memory (n=10M, m=19, GENOMIC).

Conclusions

- BB-Tree: Fastest main-memory MDIS to-date (2018) for analytical workloads
  - Read mostly, (partial-)range queries, high to moderate selectivity
- Careful tuning to properties of modern hardware
  - SIMD didn't pay off
- Bubble-buckets allow for static IST while buffering many (but not infinitely many) inserts

Limitations

- Superbuckets currently are not balanced
  - May create "super ordinary bucket" – large local scan
  - Solution: Keep superbuckets balanced (with depth 1)
- Order of delimiter dimensions is global
  - Limited fit to data clustering in subspaces
  - Solution: Recursive re-partitioning; expensive
- Rebuilds are costly (index stalls)
  - Solution: Rebuild in background; reservoir sampling
- Analytical workloads versus write-heavy workloads
  - Solution: Do not use BB-Trees for write-heavy skewed workloads
- Rebuild capacity calls for workload adaptation
- No concurrent writes / transaction management

Since 2018

- Learned indexes – learn function to map keys on blocks
  - E.g. regression: O(d) for computing location, O(1) for access
    - If location is predicted perfectly, otherwise some neighborhood search
  - Difficult (impossible) to update
- Adaptive indexing: Start with empty index and build tree sequentially based on delimiters of real queries
  - Automatically adapts index to workload (if stable)
  - No updates ever implemented – rebuild regularly
- ELF: Prefix tree over compressed dictionary
  - Replace all values with index of a sorted dictionary per dimension
    - Less space necessary for keys, uniform length
  - Impossible to update

Many Experiments, Summary

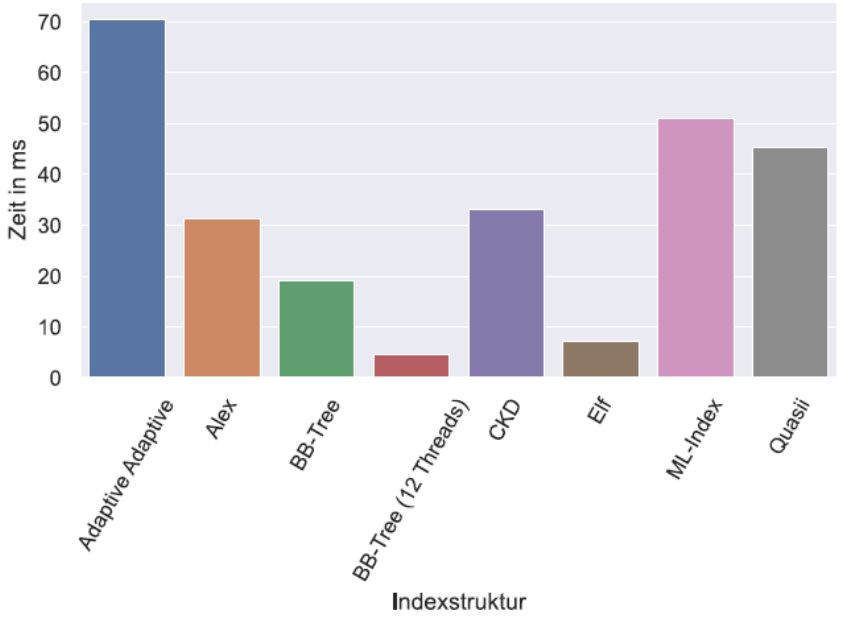


Abbildung 30: Experiment 2: GMRQB, Bereichsabfragen (gemischte Templates), durchschnittliche Laufzeit (in ms), 1.000.000 Datenpunkte, 10.000 Abfragen inklusive BB-Tree Multithreading