Datenbanksysteme II: MDIS on Modern Hardware; BB Tree

Ulf Leser
Content of this Lecture

- MDIS On Modern Hardware
  - Adaptions
  - 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
  - No 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 - later

- **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**
  - Partition into **subsets of tuples**
  - One MDIS and thread per subset
  - SIMD: Compare k dimensions per inst.
  - Pro: Load balancing
  - Con: Scans inefficient in PM queries

- **Vertical (only scan)**
  - Each dimension one partition
  - One thread per dimension
  - SIMD: Compare k values of 1 dimension per inst.
  - Pro: Pruning in PM 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)
GMRQB

- 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 PM queries

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

Table 1: GMRQB query templates.

SELECT * FROM variations
WHERE chromosome = 5
AND location BETWEEN 1000000 AND 1000000
AND quality BETWEEN 10 AND 100
AND depth BETWEEN 10 AND 1000
AND allele_freq BETWEEN 0.5 AND 1;
SIMD only Worth for (vertical) Scans

Synthetic data, d=20, n=1E6, uniform, sel=0,1%
Scans hard to beat – even at 1% selectivity

![Graph showing query throughput against selectivity with different data structures.]  

Synthetic data, d=5, n=1E6, uniform
Vertical scans good for parallel PM queries
Hyperthreading mostly does not help at low sel.

Genomic data, $d_{\text{queries}}=2\ldots19$, $n=88,4\ E6$, avg query sel. $\sim1.5\%$
Vertical scans affected by dimensionality (too many 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.
MDIS sensitive to clustered data

Figure 8: Throughput when executing range queries with an average selectivity of 0.38% (one cluster) to 27.40% (twenty clusters) on 1 Million five-dimensional data objects using 24 software threads depending on the number of clusters.
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 complete-match queries, horizontal partitioning is beneficial
  - For partial-match queries, vertical partitioning is superior
- MDIS faster 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])

- Few arguments for conservatively adapted MDIS, but ...
Content of this Lecture

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

- Many real-life data sets are multidimensional \((d=2..100)\)
- Data analytics requires multidimensional queries
  - Point queries
  - Range-queries
  - Partial match queries
  - Neighborhood / similarity queries
Why not .... e.g. ART? [Leis et al. ICDE 2013]

- Optimized for point queries
- Range queries require expensive pointer traversals

![Diagram of radix tree](image)

Fig. 1. Adaptively sized nodes in our radix tree.
Why not .... e.g. ART? [Leis et al. ICDE 2013]

- Optimized for point queries
- Range queries require expensive pointer traversals

Fig. 1. Adaptively sized nodes in our radix tree.
Why not .... e.g. ART? [Leis et al. ICDE 2013]

- Optimized for point queries
- Range queries require expensive pointer traversals

- **Linearize** the tree?
  - Yes, but makes INSERT / DELETE difficult

- Use **buckets** as leaves?
  - Yes, but we have multidimensional data – no sort order

- For point queries, **hashing** is a strong competitor
• Actually, we can sort multidimensional data
  – Fix permutation of dimensions
  – Build concatenated keys
  – Use conventional, 1-dim index

• MD range queries become a series of 1-dim range queries
But …

- Actually, we can sort multidimensional data
  - Fix permutation of dimensions
  - Build concatenated keys
  - Use conventional, 1-dim index
- MD range queries become a series of 1-dim range queries

- But: Partial match queries
  - E.g. condition only on Z
  - Pruning power lost
Content of this Lecture

• MDIS On Modern Hardware
  – MDIS Adaptions
  – 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

Many buckets

Superbucket
Growing and Shrinking
Growing and Shrinking

High fan-out
Many buckets
**Buffers many INS**
But ...
Growing and Shrinking
Rebuild

Extended IST
Additional layer
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: 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 (little pruning power) are usually ignored
  - 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 mixed

• Five competitors
  – kd-tree, PH-tree, VA-file, R*-tree, scans
  – See [Sprenger et al., SSDBM 18] for adaptation to modern hardware
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

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

Essentially a point query
Modern Hardware

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

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

- Scan scales better (10x vs 5x)
  - Scan: Very few serial components
  - BB-Tree: Single-threaded IST search
- Hyper-threading offers little to BB-Tree (compute-bound)
More Experiments in the Paper

- Impact of bucket capacity
  - Allows tuning for expectations on average query selectivity

- Point queries
  - kd-Tree is faster, but not too much

- Inserts and deletes
  - BB-Tree is fastest (after scan for inserts) despite (rare) rebuilds

- Varying dimensionality (up to 100)
  - All methods degrade

- Mixed workloads
  - BB-Tree fastest, but depends on the mix point/range/update

- Space consumption
  - BB-Tree requires least space (after scan)
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 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
References


• Sprenger, S., Schaefer, P. and Leser, U. (2019). “BB-Tree: A main-memory index structure for multidimensional range queries”. Int. Conf. on Data Engineering, Macau, China
Acknowledgements