INF FAKULTÄT FÜR INFORMATIK

SIMD Acceleration for Index Structures

Marten Wallewein-Eising

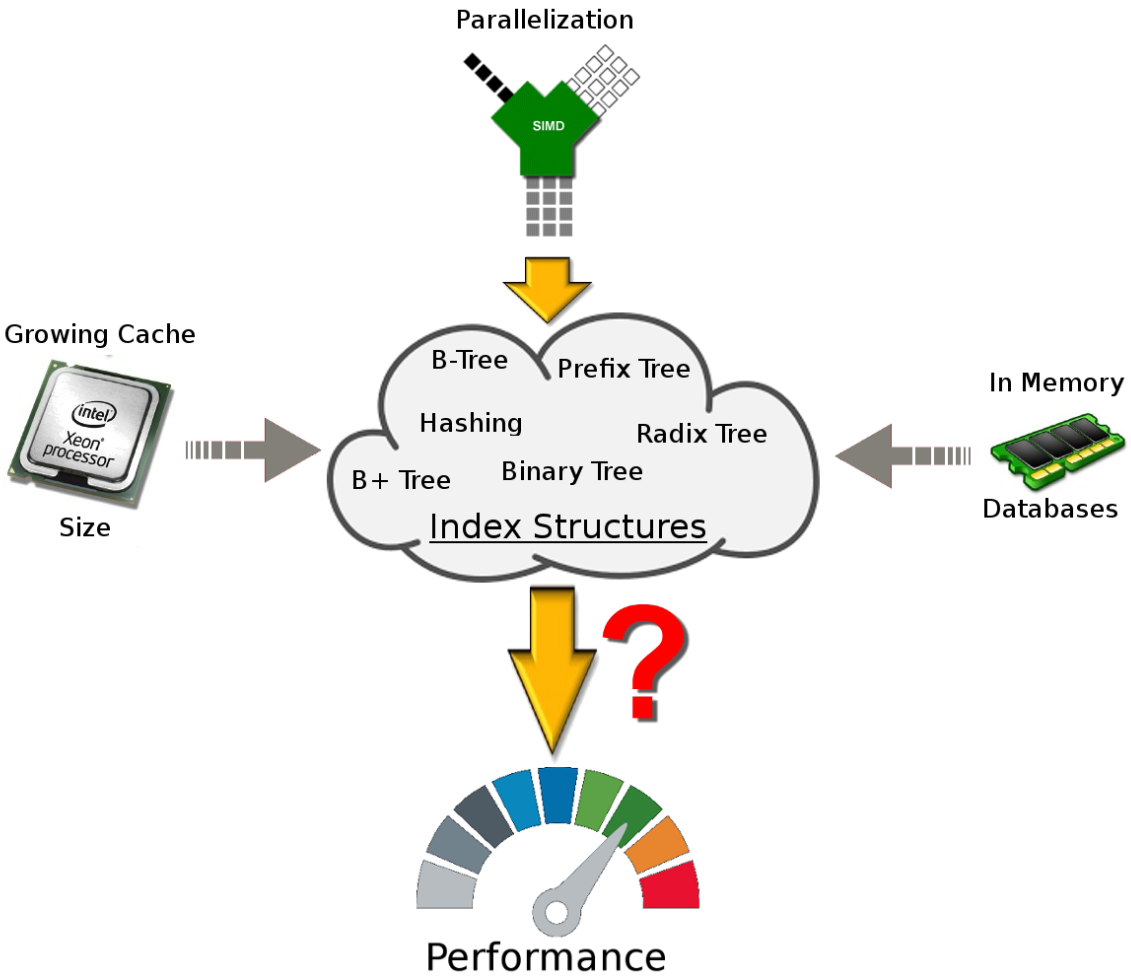
Otto von Guericke Univerity, Magdeburg

January 22, 2018

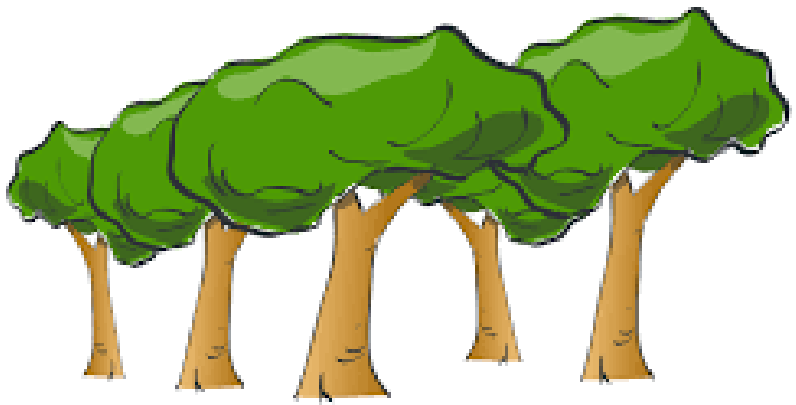
Agenda

- **Motivation**
- **B$^+$- and Radix-Trees**
- **SIMD Style Processing**
- **Adapted Tree structures**
  - Seg-Tree/Trie
  - FAST
  - VAST
  - ART
- **Evaluation**
- **Conclusion**

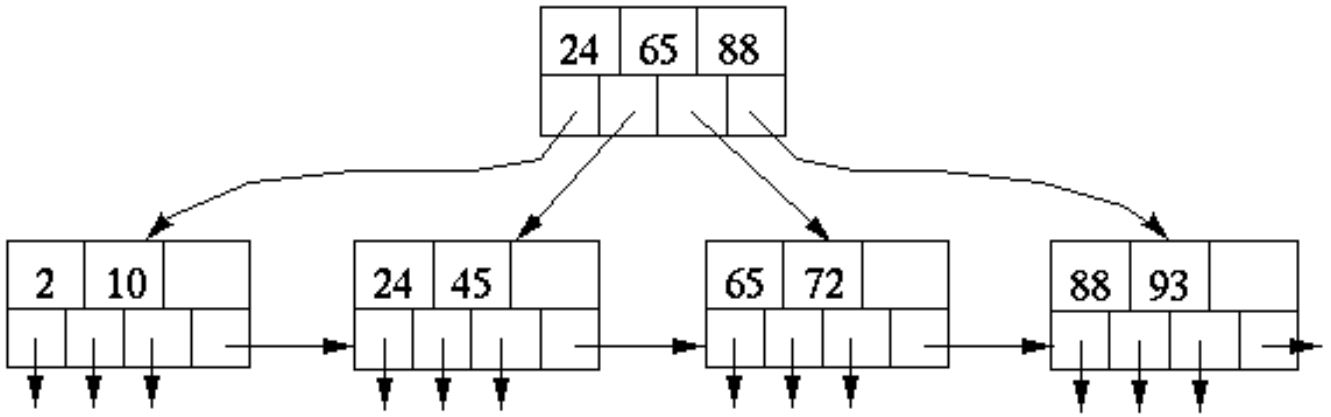
Motivation


Parallelization
SIMD
Growing Cache
intel
Xeon processor
Size
B-Tree
Prefix Tree
Hashing
Radix Tree
B+ Tree
Binary Tree
Index Structures
In Memory
Databases
?
Performance

B^+- and Radix-Trees

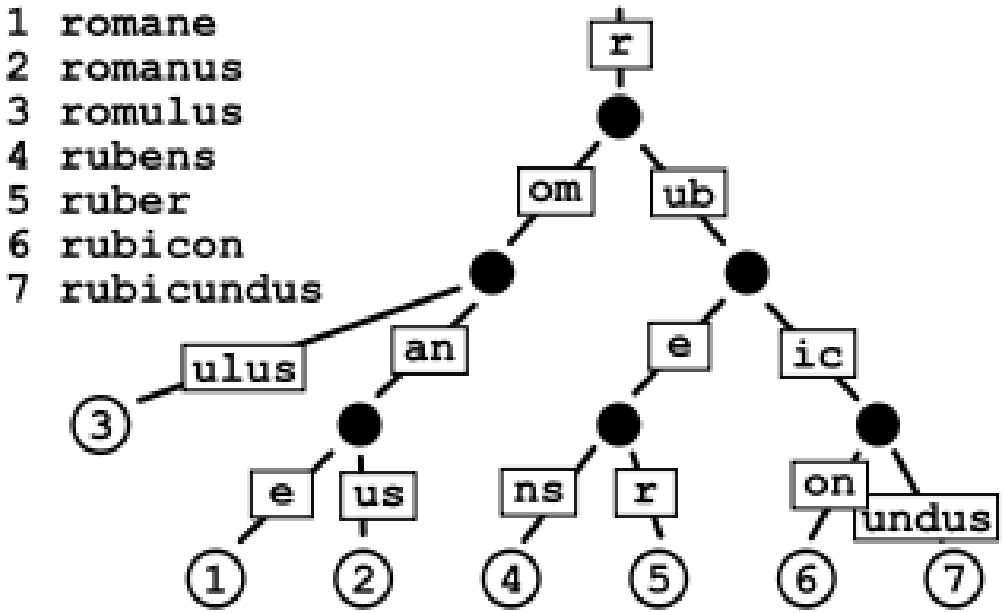


B$^+$-Tree



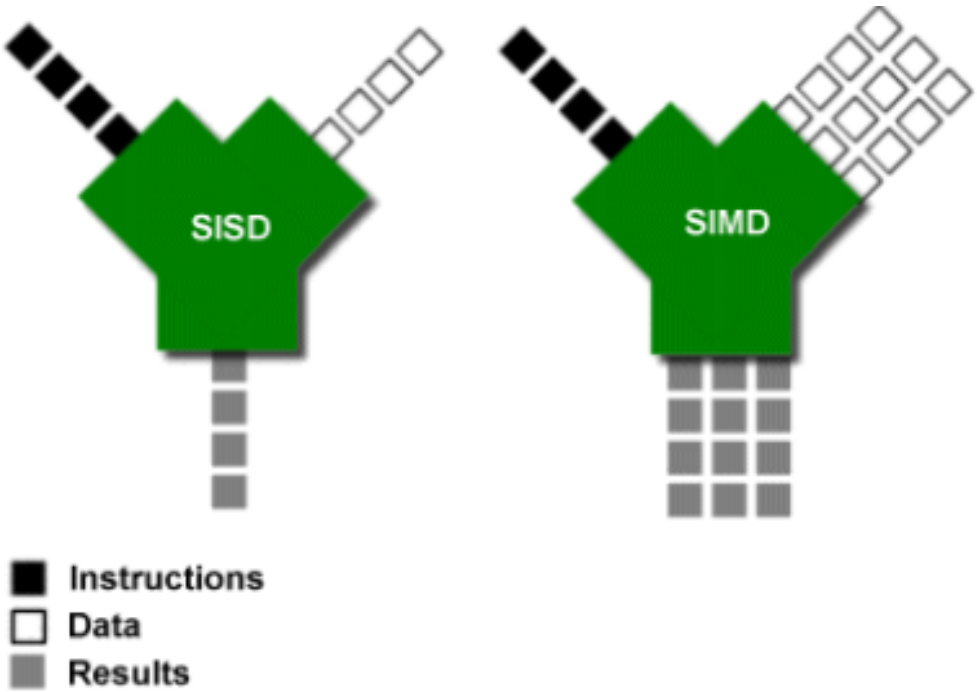
- N-ary tree with large number of children per node
- Only leaf nodes contain values, inner nodes only children
- Leaf nodes often linked for range based scans

Radix-Tree



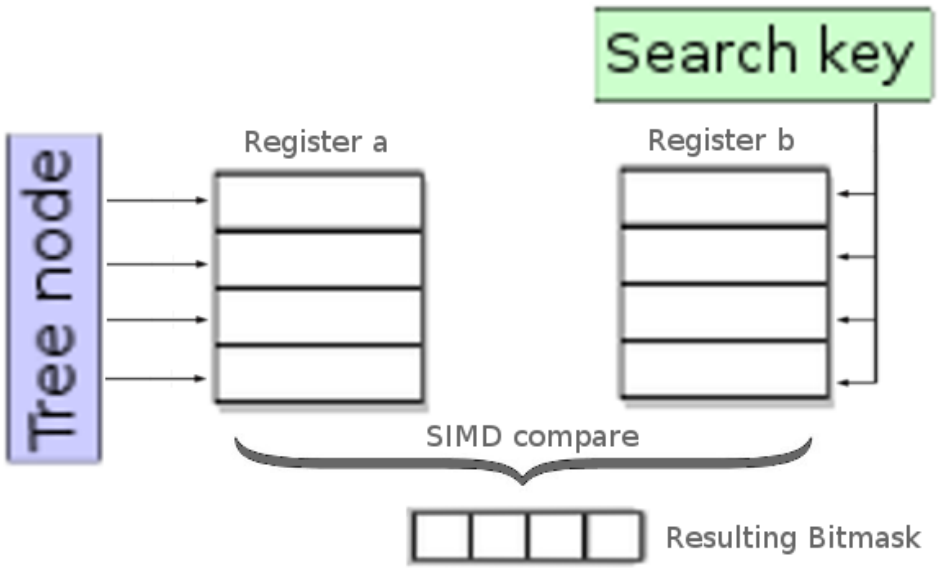
- Space optimized prefix tree
- Number of children of every inner node is at least the radix r
- Each node that is the only child is merged with its parent

Single Instruction Multiple Data



- **__m128i _mm_cmpgt_epi32 (__m128i a, __m128i b)**
  Compares 4 signed 32-bit integers in a and 4 signed 32-bit integers in b for greater-than.

Horizontal Vectorization

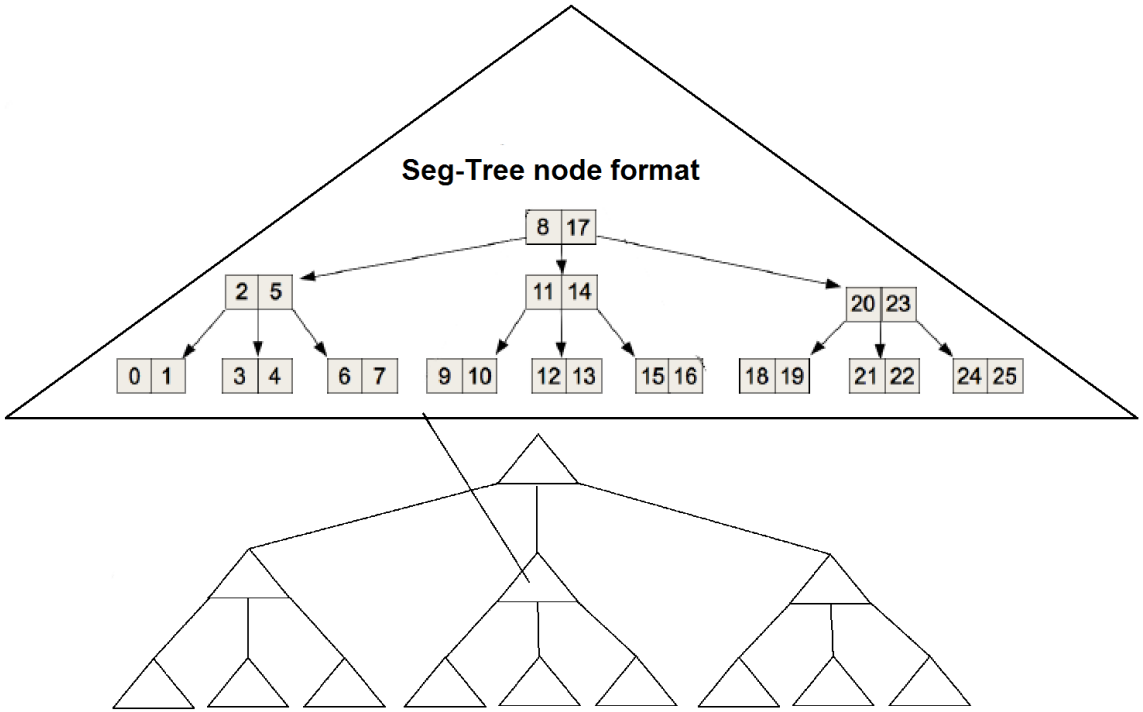


- Compare one search key to multiple keys of the index structure
- Opposite: Vertical vectorization
  - Not possible, since sequential data storage in main memory is needed

Adapted Tree structures

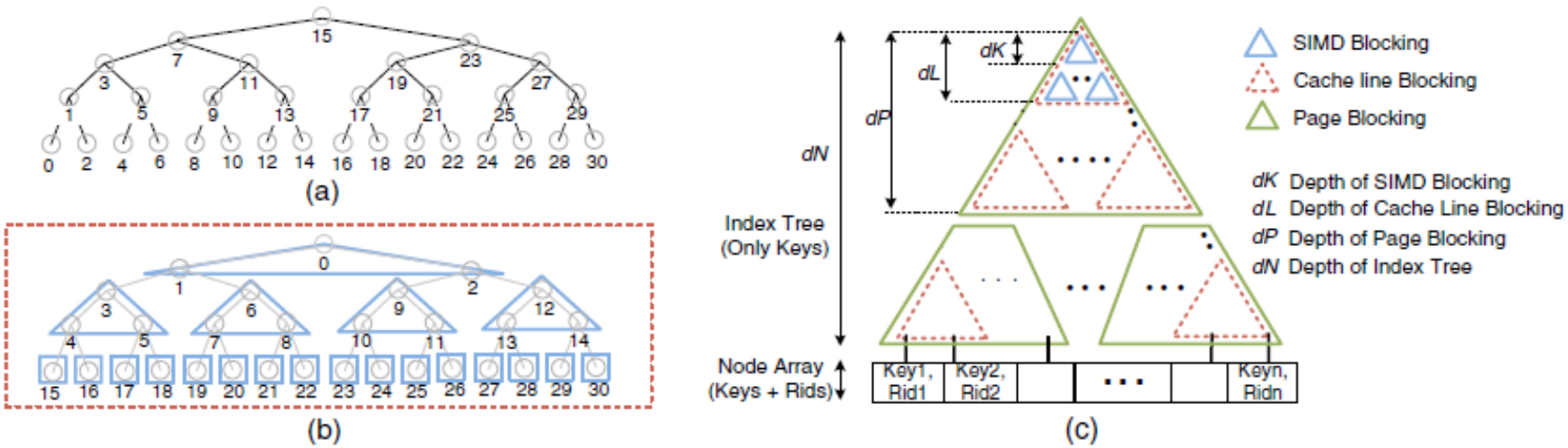
- Seg-Tree/Trie
- FAST: Fast Architecture Sensitive Tree
- VAST: Vector-Advanced and Compressed Structure Tree
- ART: Adaptive Radix Tree

Seg-Tree/Trie



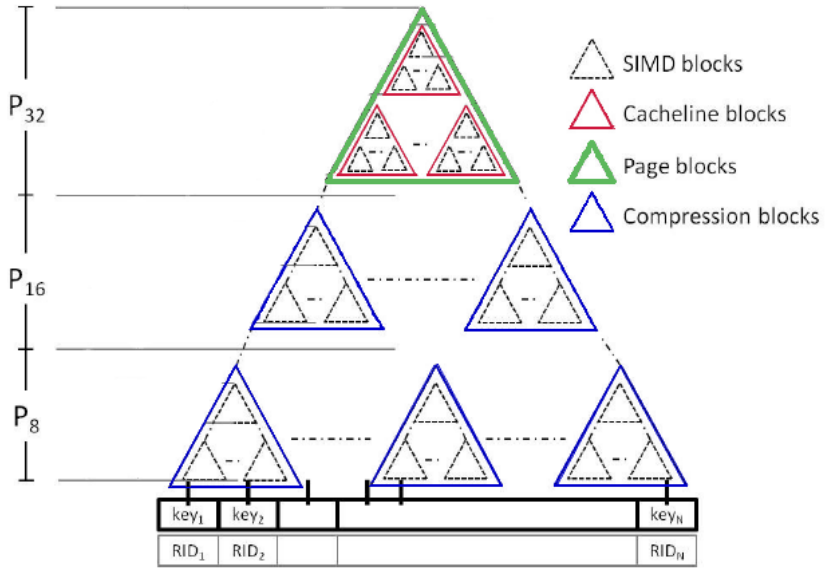
- Each node is a k-ary search tree
- $k = \frac{|SIMD|}{|Key|}$, k keys are compared in parallel

Fast Architecture Sensitive Tree



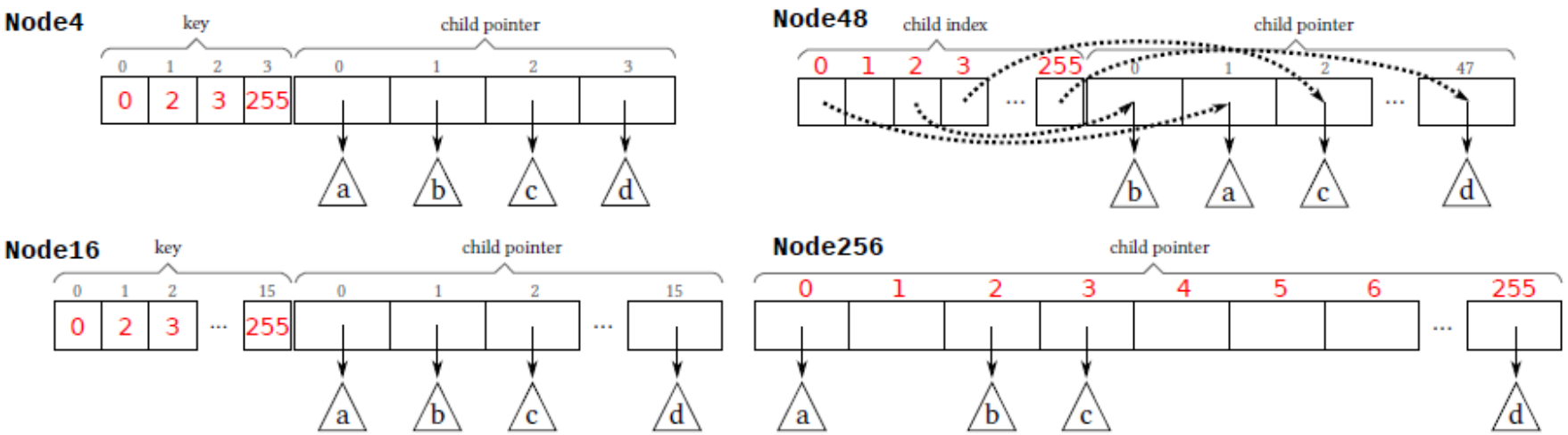
- Based on binary tree
- Hierarchical blocking: SIMD, cache line and page blocks
- Efficient cache line and page usage

Vector-Advanced and Compressed Structure Tree



- Layer P_{32} same blocking structure as FAST, 32 Bit keys
- Layer P_{16} and P_8 compressed keys to 16 and 8 Bit (lossy)
- Leaf nodes compressed lossless

Adaptive Radix Tree



- Different node formats and sizes
  - More flexible
- Horizontal Vectorization on keys of *Node16*
- Lazy expansion and path compression

Evaluation

Important criteria for performance increase:

- Horizontal vectorization
- Minimized key size
- Adapted node sizes and types
- Decreased branch misses
- Full use of cache line using blocking and alignment
- Usage of compression
- Adapt search algorithm for linearised nodes

Evaluation

Implementation of the considered performance criteria and their impact:

Criterium	Seg-Tree/Trie	FAST	ART	VAST	Impact
Horizontal vectorization	x	x	x	x	high
Minimized key size	o	-	x	x	high
Adapted node sizes and types	-	x	-	x	low
Decreased branch misses	-	x	-	x	medium
Full use of cache line using blocking and alignment	-	x	-	x	medium
Usage of Compression	o	-	x	x	medium
Adapt search algorithm for linearised nodes	x	-	-	-	low

Legend: x: implements the issue, o: partially implements the issue, -: not implements the issue

Conclusion

How to adapt index structures to modern database systems:

- Compare as many keys as possible in parallel with SIMD
  - Direct performance increase up to a multiple
- Efficient usage of cache line
- Decrease branch misses
- Use compression or adapted search algorithms

Sources

- http://infolab.stanford.edu/~nsample/cs245/handouts/hw2sol/sol2.html
- https://en.wikipedia.org/wiki/Radix_tree
- https://www.clker.com/clipart-bosque.html

Thank you for your attention!