Making B+-Trees Cache Conscious in Main Memory

Jun Rao and Kenneth A. Ross
Columbia University
SIGMOD 2000, Dallas, TX

Presented by Simonas Šaltenis
September 27, 2010
Outline

• Motivation
  ▪ Memory hierarchy

• The CSB*-tree
  ▪ Structure
  ▪ Algorithms
  ▪ Segment CSB+-tree
  ▪ Full CSB+-tree

• Performance experiments

• Conclusions

• Evaluation
Motivation

• Memory sizes grow fast.
• The need for performance also grows.
• Thus, main-memory database systems are becoming widely used.

• **Main-memory index structures** are essential to high performance main-memory data access.
  - None of the straight-forward solutions use *main-memory hierarchy* optimally
    - Balanced binary search trees
    - T-trees
    - B*-trees
Motivation

• Why memory hierarchy?
  ▪ CPU and memory performance gap

The graph adopted from:
Motivation

- Memory hierarchy

- Decreasing cost per bit
- Increasing capacity
- Increasing access time
- Decreasing frequency of access of the memory by the processor (the goal of the architecture)
Motivation

- Cache and main memory
  - The memory is divided into consecutive blocks, each $K$ words long
  - A cache stores $C$ blocks of data

- Sizes in modern processors
  - $K \approx 64 - 128$ bytes
  - L1 cache (data/instruction) $\approx 32 - 64$ Kb
  - L2 cache $\approx 256$Kb – 8Mb
  - L3 cache $\approx$ few to tens of Mb
CSB\(^+\)-tree structure

- Node size = cache line size
- **The goal:** squeeze in as many keys per node as possible
  - Increased fan-out $\rightarrow$ reduced tree height $\rightarrow$ reduced # of node accesses $\rightarrow$ reduced # of cache misses

- Key assumption:
  - The size of the pointer is similar to the size of the key $\rightarrow$ pointers occupy a large portion of a B\(^+\)-tree node.

- Idea:
  - *Remove all but one pointers from a node!*
  - Store all children of a node in a continuous block of memory – *node group.*
CSB\(^+\)-tree structure

- CSB\(^+\)-tree node (stores from \(d\) to \(2d\) keys):
  - \(n\text{Keys}\): # of keys in the node
  - \(first\text{Child}\): pointer to the first child node
  - \(key\text{List}[2d]\): a list of keys

- Search the same as in B\(^+\)-trees:
  - To get to the \(k\)-th child: \(first\text{Child} + k \times \text{nodeSize}\)
CSB$^+$-tree insertion

- Insertion is analogous to B$^+$-tree, except for node splitting:
  - Case 1: parent does not get overfull
    - Allocate a new, large node group, remove old node group
    - Let’s insert 23
**CSB\(^+\)-tree insertion**

- Insertion is analogous to B\(^+\)-tree, except for node splitting:
  - Case 2: parent gets overfull
    - Split parent, create a new node group and assign nodes to the two groups according to the parent’s split
    - Let’s insert 41
CSB\(^+\)-tree insertion

- Insertion is analogous to B\(^+\)-tree, except for node splitting:
  - Case 2: parent gets overfull
    - Split parent, create a new node group and assign nodes to the two groups according to the parent’s split
    - Let’s insert 41

![Diagram of CSB\(^+\)-tree insertion]
Full CSB\textsuperscript{+}-tree

- Problem with CSB\textsuperscript{+}-tree:
  - Split is expensive: allocation/de-allocation of node groups, copying of multiple nodes

- **Full CSB\textsuperscript{+}-tree**
  - Idea: pre-allocate all node groups to be of maximum size
Full CSB\(^+\)-tree

- Full CSB\(^+\)-tree
  - Let’s insert 23.
Full CSB\(^+\)-tree

- Full CSB\(^+\)-tree
  - Let’s insert 23.
Segmented CSB\(^+$\)-tree

- Problem with full CSB\(^+$\)-tree: wasted memory
- Another idea: reduce the size of node groups
- **Segmented CSB\(^+$\)-tree:**
  - Each node has more than one pointer (for example, two pointers) to node groups storing its children
Empirical study: setup

• Setting
  • Ultra Sparc II machine:
    ▸ L1 data cache: 16Kb, line size: 32 bytes
    ▸ L2 cache: 1Mb, line size: 64 bytes
  • Node size = L2 cache line size
    ▸ $B^+$-trees: 7 keys, 8 child pointers
    ▸ CSB$^+$-tree: 14 keys

• Implementation tricks:
  • Recursive decent iteratively – avoiding function calls
  • Unwinding of a loop for binary searching in a node:
    ▸ Instead a tree of if-then-else statements, hard-coding the search tree for a given node size
Empirical study

• Search performance

• All variants of the CSB\(^+\)-tree beat the regular B\(^+\)-tree
Empirical study

- Insert performance

- Insertions are expensive in CSB\(^{+}\)-tree, except the full CSB\(^{+}\)-tree
Empirical study

- Overall workload performance

- Full CSB\(^+\)-tree is best across the board
Conclusions

• Full CSB⁺-tree is best in all aspects except for space
• (Partial) pointer elimination is a general technique, that can be applied to other index structures
  ▪ Less effective, when keys are large (e.g., R-tree)
Evaluation

• Positive:
  ▪ Well written paper
  ▪ Careful implementation and performance experiments
  ▪ Repeatable performance experiments

• Negative:
  ▪ Too many implementation details! (not all are necessary)
    ▫ For example, #ifdef on page 482
  ▪ Could have more examples
  ▪ Different types of queries are not explored (range/point)