Datenbanksysteme II: B / B+ / Prefix Trees

Ulf Leser
Content of this Lecture

- B Trees
- B+ Trees
- Index Structures for Strings
Recall: Multi-Level Index Files

Sparse 2nd level | Sparse 1st level | Sorted File

10 90 170 250
330 410 490 570

10 30 50 70
90 110 130 150
170 190 210 230

10 20
30 40
50 60
70 80
90 100
B-Trees

- B-Tree is a multi-level index with variable number of levels
- Height adapts to table growth / shrinkage
Formally

- Assume index on primary key (no duplicates)
- Internal nodes contain pairs (value, TID) and pointers
- **Leaf nodes** only contain (value, TID)
- Block can hold **2k triples** (pointer, value, TID) plus 1 ptr
- Each **internal node** contains between k and 2k (value, TID)
  - k+1 and 2k+1 pointers to subtrees
    - Subtree left of m contains only and all values y<V_m
    - Subtree right of m contains only and all values V_m with y>V_m
    - V_m < V_{m+1}
  - Exception: Root node
- **B-trees use always at least 50% of allocated space**

\[
\begin{array}{cccccccc}
  P_0 & V_1 & P_1 & \ldots & P_{2k-1} & V_{2k} & P_{2k}
\end{array}
\]
Searching B-Trees

Find 8
1. Start with root node
2. Follow p₀
3. Follow p₁
4. Scan - found

Find 60
1. Start with root node
2. Follow p₂
3. Follow p₁
4. Not found
Complexity

• B-trees are always balanced (how: Later)
  - All paths from root to a leaf are of equal length

• Assume n keys; let \( r = |\text{value}| + |\text{TID}| + |\text{pointer}| \)

• Best case: All nodes are full
  - We have \( b \approx n \times r / 2k \) blocks
    • Actually a little less, since leaves contain no pointers
  - Height of the tree \( h \approx \log_{2k}(b) \)
  - Search requires between 1 and \( \log_{2k}(b) \) IO

• Worst case: All nodes contain only \( k \) values
  - We need \( b \approx n \times r / k \) blocks
  - Height of the tree \( h \approx \log_k(b) \)
  - Search requires between 1 and \( \log_k(b) \) IO
Example

- Assume $|\text{value}|=20$, $|\text{TID}|=16$, $|\text{pointer}|=8$, block size=4096 => $r=44$
- Assume $n=1.000.000.000$ (1E9) records
- Gives between 46 and 92 index records per block
- Hence, we need between 1 and $5/6$ IO
- Caching the first two levels (between 1+46 and 1+92 blocks), this reduces to a maximum of $3/4$ IO
Inserting into B-Trees

- We insert 5 (note: 2*k=2)
Inserting into B-Trees

- We insert 6
- Block is full – we need to adapt
Inserting into B-Trees

- Split overflow block and *propagate middle value* upwards
  - All values from old node plus new value minus middle value are evenly split between two new nodes
  - Thus, each has ~k keys
  - Middle value is pushed up to parent node
Inserting into B-Trees

- We insert 40
- Block is full – split and propagate
- Propagating upwards leads to new overflow block
- Finally, the root node overflows
  - B-trees grow upwards
Intermediate 2

- 10 30
- 40 -
- 75 -
- 32 38 39 - -
- 50?
- 40 -
- 76 85 88 91 -
- 45 49 - - -
Final Tree
Longer Sequence of Insertions
Complexity of Insertion

• Let h be height of tree
• Cost for searching leaf node: h IO
• If no split necessary: Total cost = h+1 (writing)
• If split is necessary
  – Worst case – up to the root
  – We assume we cached ancestor blocks during traversal
  – We thus need to read them once and write them once
  – Total cost: (h+2)+2(h-1)+1 = 3h+1
    • Split on all levels and create new root node
    • Not caching adds one more h
Deleting Keys

- **Search key**
  - If found in internal node
    - Follow pointer left from node
    - Choose largest value from left subtree and replace deleted value
      - This value must be in a leaf
    - Delete value in leaf (and progress)
  - If found in leaf
    - Delete value
    - If blocks underflows, choose one of neighboring blocks
    - If both blocks together have more than 2k records
      - Distribute values evenly; adapt between-key in parent node
    - Otherwise – merge blocks
      - Store in one block all records plus value in parent between those blocks
      - Delete other block
    - Might work recursively up the tree
Delete with Underflow

- Delete 40
Delete with Underflow

- Borrow from right subtree
Delete with Underflow

- Underflow
- merge blocks

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>30</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>-</td>
<td></td>
<td>50</td>
<td>-</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>39</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>38</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>45</th>
<th>49</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>51</td>
<td>55</td>
<td>58</td>
<td>-</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>75</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>85</td>
<td>88</td>
<td>91</td>
<td>-</td>
</tr>
</tbody>
</table>
```
Delete with Underflow

- Delete 45
- Underflow
- No local repair

```
/ \                        /     \       /
/   \                      /       \     /     \      
/     \                    /         \   /       \     /     \   
10    -                      50    -       30                        39    -       
     /\                        /\      /\                  /\      
    /  \                      /  \    /  \                /  \    /  \     
   /    \                    /    \  /    \              /    \  /    \   
  /      \                  /      \ /      \            /      \ /      \  
 /        \                /        \ /        \          /        \ /        \ 
/          \              /          \ /          \        /          \ /          \ 
/            \            /            \ /            \      /            \ /            \ 
\          \            \          \ /          \        \          \ /          \        
 \        \            \        \ /        \        \        \ /        \        \ 
  \      \            \      \ /      \        \      \ /      \        \      
   \    \            \    \ /    \        \    \ /    \        \    \ /    \ 
    \  \            \  \ /  \        \  \ /  \        \  \ /  \ 
      \ \            \ \ /   \        \ \ /   \        \ \ /   \ 
        \   \            \   \ /     \        \   \ /     \     \ 
          \    \            \    \ /       \        \    \ /       \
            \     \            \     \ /         \        \     \ /         \\n              \      \            \      \ /           \        \      \ /           \\n                \       \            \       \ /            \        \       \ /            \\n```

```
Delete with Underflow

- Delete 45
- Merge blocks

![Diagram showing deletion and merging of blocks]
Delete with Underflow

- Up the tree
Complexity of Deleting Keys

- Going down costs mostly $h+1$
  - If found in leaf, it costs $h$
  - If found in internal node, we anyway read $h$ blocks to choose largest value in subtree and write internal node
- If no underflow, total cost is $h+2$
- If local underflow, total cost is $h+6$
  - Checking left and right neighbor, writing block and chosen neighbor, writing parent
- If blocks underflow bottom-up, total cost is at most $3h-2$
  - Similar argument as for insertion
B-trees on Non-Unique Attributes

• Option 1: Compact representation
  - Store (value, TID₁, TID₂, ... TIDₙ)
  - Difficult to handle – internal nodes cannot keep fixed number of pairs any more
  - Requires internal overflow blocks
• Option 2: Verbose representation
  - Treat duplicates as different values
  - Constraints on keys chance from “<” to “≤”
  - Generates a tree although a list would suffice
• Better: B* trees
Content of this Lecture

- B Trees
- B+ Trees
- Index Structures for Strings
B+ Trees

• Dense index on heap-structured data file
• **Only records in leaves** are pairs (value, TID)
• Records in **internal nodes contain only values**
  - Values demark borders between subtrees
  - Concrete values need not exist in the database - only **signposts**
• Leaves are connected by pointers (faster **range queries**)

---

**B+ Tree as dense index**

**Data file organized as heap file**
Operations

• Searching
  - Essentially the same as for B trees
  - But will always go down to leaf - slightly worse IO complexity

• Insertion
  - Essentially the same as for B trees
  - Values are only inserted at leave nodes
  - When block is split, no value moves upwards
    - But “parent value” might change
    - Different methods for choosing new value for parent node

• Deletion
  - Deletion in internal node cannot occur
  - When blocks are merged, no values are moved down
    - But values in parent node are deleted as well
Advantages

• Simpler operations
• No TIDs in internal nodes (main advantage of B+ trees)
  – More records per page in internal nodes
  – Increased fan-out, reduced average height
• Not necessarily “real” values in internal nodes
  – Can save further space – Prefix B+ Tree (later)
• Linked leaves
  – Faster range queries – traversal need not go up/down the tree
  – Optimally, leaves are in sequential order on disk
• Can we increase space usage guarantee beyond 50%?
B* tree: Improving Space Usage

- Don’t split upon overflow: Move values to neighbor blocks
  - More complex operations, higher IO cost, but on average lower height of trees
- When splitting, make three out of two
  - Don’t split single block
  - Choose sibling and generate three new blocks from two old ones
  - Guaranteed 66% space usage
IO versus RAM model

- Consider again searching a B+ tree
- How many comparisons are we performing??
IO versus RAM model

• How many comparisons are we performing?
  - We scan \( h = \log_k(n) \) blocks
  - In each block, we can perform binary search on all keys
  - That is \( \log(k) \) comparisons
  - Together: \( \log_k(n) \times \log(k) \)
  - Imagine \( k = 50, \ n = 1E9 \): \( 6 \times 6 = 36 \) comparisons

• Using a **binary tree**
  - \( \log(n) \) comparisons: 29 comparisons
  - Better as main memory data structure, but worse IO

• Larger blocks: \( k = 10.000 \): 42 comparisons (but only 3 IO)

• **Increasing block size** = IO reduction, CPU increase
B+ Trees and Hashing

• Hashing faster for some applications
  - Can lead to O(1) IO
  - Requires relatively static data
    • All extensible schemes may degenerate
  - Requires perfectly fitting hash function – domain dependence

• B+ trees
  - Very few IO if upper levels are cached
  - Adapts to skewed (non-uniformly distributed) data
  - More robust, no selection of hash function required
Loading a B+ Tree

• What happens in case of

    create index myidx on LARGETABLE( id);
Loading a B+ Tree

- What happens in case of
  \[\text{create index myidx on LARGETABLE( id);}\]

- Performing record-after-record insertion
  - Each insertion has \(3h+2 = O(\log_k(b))\) block I/O
  - Altogether: \(O(n*\log_k(b))\)

- Blocks are read and written in arbitrary order
  - Very likely: bad cache-hit ratio

- Space usage will be anywhere between 50 and 100%

- Can’t we do better?
Bulk-Loading a B+ Tree

- First **sort records**
  - $O(n \cdot \log_m(n))$, where $m$ is number of records fitting into memory
  - Clearly, $m \gg k$

- Insert in **sorted order** using normal insertion
  - Tree builds from lower left to upper right
  - **Caching will work very well**
  - But space usage will be only around 50%

- **Alternative**
  - Compute **structure in advance**
    - Every 2k’th record we need a separating key
    - Every 2k’th separating key we need a next-level separating key
    - ... 
  - Can be generated and written in linear time
Content of this Lecture

• B Trees
• B+ Trees
• Index Structures for Strings
  - Prefix B+ Tree
  - Prefix Tree
  - PETER
  - PEARL
Prefix B+ Trees

- Consider **string values as keys**
- Keys for int. nodes: Smallest key from right-hand subtree
  - Leads to internal key as large as leaf keys
- Prefix B+ trees – **Shortest string** separating greatest key in left-hand subtree from smallest key in right-hand subtree

**Advantage:** Reduced space usage
**Disadvantage:** Overhead for computing values
Prefix Tree

- If we index many strings with many common prefixes
  - … as in Information Retrieval …
  - Why store common prefixes multiple times?
- Prefix trees
  - Store common prefix / substring in internal nodes
  - Searching a key $k$ requires at most $|k|$ character comparisons
Indexing Strings

• Note: Prefix/Patricia trees are main memory structures
  - Not balanced – no guaranteed worst-case IO
  - How to optimally layout internal nodes on blocks?

• More index structures for strings
  - Keyword trees – searching for many patterns simultaneously
    • Necessary for joins on strings
    • Persistent keyword trees – challenge
  - Suffix trees – indexing all substrings of a string
    • Necessary e.g. to search genomic sequences
    • Persistent suffix trees – challenge in advancement
PETER

- Computes joins / search on large collections of long strings much faster than traditional DB technology
- Also handles similarity search / similarity joins
- Open source
Prefix-Trees (also called Tries)

- Given a set $S$ of strings
- Build a tree with
  - Labeled nodes
  - Outgoing edges have different label
  - Every $s \in S$ is spelled on exactly one path from root
  - Mark all nodes where an $s$ ends
- **Common prefixes** are represented only once

```
cattga, gatt, agtactc, ga, agaatc
```
Searching Prefix-Trees

- Search t in S
- Recursively match t with a path starting from root
  - If no further match: $t \notin S$
  - If matched completely: $t \in S$

- Search complexity
  - Only depends on depth of S
  - Independent from $|S|$
Compressed Prefix Trees

- More complex implementation
- Different kinds of edges/nodes
Large Prefix Trees

- **Unique suffixes** are stored (sorted) on disk
  - Minus a preview

- **Tree of common prefixes** is kept in main memory
  - Many (failed) searches never access disc
  - At most one disc access per search
Similarity Search on Prefix-Trees

• In similarity search, a mismatch doesn’t mean that $t \not\in S$
• **Several mismatches** are allowed
  - Depending on error threshold
• Idea
  - Depth-first search on the tree as usual
  - Keep a **counter for the n\# of mismatches** spent in the prefix so far
  - If counter exceeds threshold – stop search in this branch
  - **Apply tricks** to stop although n\# of mismatches not yet too big
Example: Search

Hamming distance search for $t = \text{CTGAAATTTGGT}$, $k=1$
Example: Search

Hamming distance search for t = CTGAAATTGGT, k=1
Example: Search

Hamming distance search for \( t = CTGAAATTTGGT, k=1 \)
Example: Search

Hamming distance search for \( t = \text{CTGAAATTTGGT} \), \( k=1 \)
Example: Search

Hamming distance search for $t = \text{CTGAAATTGGT}$, $k=1$
Example: Search

Hamming distance search for $t = \text{CTGAAATTTGGT}$, $k=1$
(Similarity) Joins on Prefix Trees

- We compare growing prefixes with growing prefixes
- Essentially: Compute intersection of two trees
- Traverse both trees in parallel
  - Upon (sufficiently many) mismatches, entire subtrees are pruned
- Same idea for exact and similarity join
Setup

- **Data:** Several EST data sets from dbEST
  - **Search:** All strings of one data set in another data set
  - **Join:** One data set against another one
  - **Varying similarity thresholds**
- **(Linear) Index creation** not included
Search: Comparing to Flamingo (2011)

- Flamingo: Library for approximate string matching
  - http://flamingo.ics.uci.edu/
  - Based on an inverted index on q-grams
  - Uses length and charsum filter
PETER inside a RDBMS

- We integrated PETER into a commercial RDBMS using its extensible indexing interface
  - Joins: table functions
  - Tree stored in separate file, suffixes stored in table

- Hope
  - As search complexity is independent of $|S|$, ...
    - we might even beat B+ trees for exact search on very large $|S|$
    - we might even beat hash/merge for exact join of very large data sets

- First hope not fulfilled
  - API does not allow caching of tree – index reload for every search
  - Large penalty for context switch through API
    - Especially for JAVA!
Similarity Search RDBMS

- Peter (behind extensible indexing interface) versus UDF implementing hamming / edit distance calculations
- Difference: 2-3 orders of magnitude, independent of data set, threshold, or search pattern length
(Similarity) Join inside RDBMS

- **PETER** (behind extensible indexing interface) versus build-in join (exact join, hash and merge) or UDF
- **Similarity join**
  - Join T3 with T2e, k=2, inside RDBMS: Stopped after 24 h
  - Same join with PETER: 1 minute
- **Exact join**
  - For long strings, PETER is significantly faster even when compared to commercial join implementations
PEARL: Multi-Threaded PETER

Room for Improvement

Fig. 7. PeARL speed-up for similarity search on k=2.
Why?

Fig. 2. MapReduce workflow of similarity joins in PeARL.