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

<table>
<thead>
<tr>
<th>10</th>
<th>30</th>
<th>70</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>50</td>
<td>70</td>
<td>20</td>
</tr>
<tr>
<td>170</td>
<td>70</td>
<td>70</td>
<td>30</td>
</tr>
<tr>
<td>250</td>
<td>70</td>
<td>70</td>
<td>40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>330</th>
<th>90</th>
<th>50</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>410</td>
<td>110</td>
<td>60</td>
<td>70</td>
</tr>
<tr>
<td>490</td>
<td>130</td>
<td>70</td>
<td>80</td>
</tr>
<tr>
<td>570</td>
<td>150</td>
<td>80</td>
<td>90</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>170</th>
<th>90</th>
<th>70</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>190</td>
<td>90</td>
<td>90</td>
<td>100</td>
</tr>
<tr>
<td>210</td>
<td>90</td>
<td>90</td>
<td>100</td>
</tr>
<tr>
<td>230</td>
<td>90</td>
<td>90</td>
<td>100</td>
</tr>
</tbody>
</table>
B-Trees (≠ binary tree)

- B-Tree is a multi-level index with **variable number of levels**
  - Many variations: B/B+/B*/B++/BB...
- **Height adapts** to table size
- Designed for **block-wise access**
- >50% space usage guaranteed
- **Always balanced**

Formally

- Assume index on primary key (no duplicates)
- **Internal nodes** contain pairs (key, TID) and pointers
- **Leaf nodes** only contain (key, TID)
- Block can hold **2k triples** (pointer, key, TID) plus 1 ptr
- Each internal node contains **between k and 2k** (key, TID)
  - And between k+1 and 2k+1 pointers to subtrees
    - Subtree left of pair (v,TID) contains only and all keys y<v
    - Subtree right of pair (v,TID) contains only and all keys y>v
    - Pairs are sorted: \( v_i < v_{i+1} \)
  - Exception: Root node
- **Thus, B-trees use always at least 50% of allocated space**

| \( p_0 \) | \( (v_0,t_0) \) | \( p_1 \) | \( (v_1,t_1) \) | \( p_2 \) | \( (v_2,t_2) \) | \( p_3 \) | ... | \( (v_{2k-1},t_{2k-1}) \) | \( p_{2k} \) |
Searching B-Trees

Find 9
1. Start with root node
2. Follow \( p_0 \)
3. Follow \( p_1 \)
4. Scan block - found

Find 60
1. Start with root node
2. Follow \( p_2 \)
3. Follow \( p_1 \)
4. Scan block - not found
Complexity

• B-trees are always balanced (how? Wait)
  – All paths from root to a leaves are of equal length
• Assume n keys; let \( r = |\text{key}| + |\text{TID}| + |\text{pointer}| \)
• Best case: All nodes are full (2k keys)
  – We have \( b \approx n/2k \) 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 keys
  – We need \( b \approx n/k \) blocks
  – Height of the tree \( h \approx \log_k(b) \)
  – Search requires between 1 and \( \log_k(b) \) IO
Example

- Assume $|\text{key}|=20$, $|\text{TID}|=16$, $|\text{pointer}|=8$, block size=4096
  $\Rightarrow r=44$
- Assume $n=1.000.000.000$ ($1E9$) records
- Gives between 46 and 92 index records per block
- Hence, we need 5 or 6 IO
  - Essentially all data is in the leaves
  - Very small changes to find key earlier
- Caching the first two levels (between 1+46 and 1+92 blocks), this reduces to 3 or 4 IO
Inserting into B-Trees

- In B-Trees, we always **insert into a leaf**
- We insert 5 (assume: 2\(k \times k = 2\))
  - For ease of exposition, we assume 2-5 keys in leaves and 1-2 keys in inner nodes

```plaintext
1 2 3 4 -
7 -
...
15 30
50 75
32 38 39 45 49
76 85 88 91 -
9 10 11 13 -
51 55 58 - -
```
Inserting into B-Trees

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

- Split overflow block and **propagate median** upwards
  - All values from old node plus new value minus median are evenly split between two new nodes
  - Thus, each has ~k keys
  - Median is pushed up to parent node and inserted there

```
1 2 3 4 5
7 
```

```
1 2 3 - -
9 10 11 13 -
```

```
4 7
```

```
5 6 - - -
```

```
15 30...
```

```
... ...
```
Inserting into B-Trees

- We insert 40
- Block is full – split and propagate 40, the median
- Propagating upwards leads to overflow in parent(s)
- Finally, the root node overflows
  - B-trees grow upwards
Intermediate 1
Intermediate 2
Final Tree
Longer Sequence of Insertions
Complexity Insertion

- Let $h$ be height of B-tree
- Cost for searching leaf node: $h$ IO
- If no split necessary: Total IO 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
Deleting Keys

- If found in internal node
  - Choose **smallest value from right subtree** and replace deleted value
    - This value must be in a leaf
    - Recall search trees: symmetric predecessor (or successor)
  - Delete value in leaf and **progress**

- If found in leaf
  - Delete value
  - **If blocks underflows** (<k keys), choose one of neighboring blocks
    - Must have the same parent node
  - If both blocks together have **more than 2k records**: Distribute values evenly; adapt between-key in parent node
  - Otherwise – **merge blocks**
    - One block with all leaf-records plus the median in parent
    - Remove middle value in parent block – which now might underflow
  - Might work **recursively up the tree**
Delete with Underflow

- Delete 40
Delete with Underflow

- Move symmetric successor
- Underflow in leaf
Delete with Underflow

- Merge with left neighbor

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>30</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>39</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>38</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>45</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>76</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>51</td>
<td>55</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>91</td>
</tr>
</tbody>
</table>
```

…
Delete with Underflow

- Delete 45
- Underflow
- No local repair

```
30
39
10
...
32 38 - - - -
39 -
75 -
...
...
51 55 58 - - -
49 - - - - -
76 85 88 91 -
```
Delete with Underflow

- Merge blocks
- Parent underflows
Delete with Underflow

- Up the tree
Complexity of Deleting Keys

• Going down costs $h$ IO
  – If key found in leaf, it costs $h$ to read and 1 to write
  – If found in internal node, we still have to read $h$ blocks to choose replacement value from leaf

• If no underflow, total cost is $h+1$

• If local underflow (with merge), total cost is $\sim h+4$
  – Checking left and right neighbor, writing block and chosen neighbor, writing parent

• If blocks underflow bottom-up, total cost is at most $4h-2$
  – If left and right neighbors have to be checked at each level
B-trees on Non-Unique Attributes

- If duplicates exist

- Option 1: Compact representation
  - Store \((\text{value, TID}_1, \text{TID}_2, \ldots \text{TID}_n)\)
  - Difficult—internal nodes don’t have fixed number of pairs any more
  - Requires internal overflow blocks

- Option 2: Verbose representation
  - Treat duplicates as different values
  - Constraints on keys change from “<” to “≤”
  - Extreme case: 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
- **Internal nodes contain only values** and pointers
  - Values demark borders between subtrees
  - Concrete values need not exist as keys - only **signposts**
- Leaves are chained for faster **range queries**
Operations

- **Searching**
  - Essentially the same as for B trees
  - But will always go down to leaf – *marginally worse IO complexity*
- **Insertion**
  - Essentially the same as for B trees
  - When block is split, no value moves upwards
    - Parent block still changes – *new signpost*
    - Typical choice: \( \text{avg}(v_{\text{median}-1}, v_{\text{median}+1}) \)
- **Deletion**
  - Deletion in *internal node* cannot occur
  - When blocks are merged, no values are moved up
    - But signposts in parent node are deleted as well
Advantages

- Simpler operations
- Higher fan-out, lower IO complexity
  - No TIDs in internal nodes - more pointers in internal nodes
  - Much reduced height (base of log() changes)
- Smoother balancing: Chose signposts carefully
  - Choose such that future inserts are evenly distributed
- Linked leaves
  - Faster range queries – traversal need not go up/down the tree
  - Optimally, leaves are in sequential order on disk
B* tree: Improving Space Usage

• Can we increase space usage guarantee beyond 50%?
• Don’t split upon overflow: Move values to neighbor blocks as long as possible
  – More complex operations, need to look into neighbors
  – We only split when all neighbors and the current block is full
• When splitting, make three out of two
  – We only split when all neighbors are full – choose one
  – Generate three new blocks from the two full old ones
  – Each new block has 4/3k keys: Guaranteed 66% space usage

B+ Trees and Hashing

- Hashing faster for some applications
  - Can lead to O(1) IO
  - Assumes good hash function
  - Requires domain knowledge

- B+ trees
  - Very few IO if upper levels are cached
  - Adapts to skewed (non-uniformly distributed) data
  - Domain-independent
  - Also supports range queries
Loading a B+ Tree

- What happens in case of

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

• What happens in case of

        create index myidx on LARGETABLE( id);

• Naïve: Record-by-record insertion
  – Each insertion has $3h+1 = O(\log_k(b))$ block IO
  – 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
  - $O(n \times \log_m(n))$, 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 signposts as large as keys
- Prefix B+ trees – Shortest string separating largest key in left-hand subtree from smallest key in right-hand subtree

Advantages: Reduced space, higher fan-out
Disadvantages: Overhead for computing signpost
Prefix Tree / Patricia tree / Trie

- 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

- Prefix trees traditionally are **main memory structures**
  - How to **optimally layout** internal nodes on blocks?
  - **Not balanced** – no guaranteed worst-case IO
- 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
- There are many similar index structures
  - PRETTY, PRETTY+, MASSJoin, ...
Prefix-Trees

- 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 a string 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
- **Tree of common prefixes** is kept in **main memory**
  - Most failing searches never access disc
  - At most one disc IO per search
  - [If tree fits in main memory]
Similarity Search on Prefix-Trees

- In similarity search, a mismatch doesn’t mean that \( t \notin S \)
- Several mismatches might be allowed
  - Depending on error threshold
  - Depending on similarity function
- Idea
  - Depth-first search on the tree as usual
  - Keep a counter for the number of errors occurring in the prefix so far
  - If counter exceeds threshold – stop search in this branch
  - Pruning: Try to stop earlier by clever “guessing”
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 $t = \text{CTGAAATTGGT}$, $k=1$
Example: Search

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

Hamming distance search for $t = CTGAAATTGGT$, $k=1$

d($CTGAAATTGGT$, $CTGAGATTTGGT$) = 1
Example: Search

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

- We compare **growing prefixes with growing prefixes**
- Exact and similarity join
- Essentially: Compute **intersection of two trees**
  - Only labeled nodes are interesting
- Traverse both trees in parallel
  - Upon (sufficiently many) mismatches, entire subtrees are pruned
Evaluation

- **Data**: Several EST data sets from dbEST
  - **Search**: All strings of one data set in another data set
  - **Join**: One data set against another data set
  - **Varying similarity thresholds**

- **(Linear) Index creation** not included in measurements
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 beat B+ trees for exact search on **very large** $|S|$  
    - we might 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!
String Similarity Search in a 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 than 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.