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  10
90  30
170 50
250 70

330 90
410 110
490 130
570 150

170 170
190 210
210 230

50 70
60 80
90 100

10 20
30 40
50 60
70 80
90 100
B-Trees (≠ binary tree)

- B-Tree is a multi-level index with **variable number of levels**
  - Many variations: B/B+/B*/B++/…
- **Height adapts** to table growth / shrinkage
- Optimized for **block-based access (disc)**
- >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)
  - Plus 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**
Searching B-Trees

Find 9
1. Start with root node
2. Follow p₀
3. Follow p₁
4. Scan (binsearch) - found

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

- **B-trees are always balanced** (how: Later)
  - All paths from root to a leaves are of equal length
- **Assume n keys; let r=|key|+|TID|+|pointer|**
- **Best case: All nodes are full (2k keys)**
  - We have $b \approx \frac{n}{2k}$ 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 $b \approx \frac{n}{k}$ 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 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 (assume: $2\times k = 2$)
  - For ease of exposition, we assume 2-5 keys in leaves and 1-2 keys in inner nodes
Inserting into B-Trees

- We insert 6
- Block is full – we need to split
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 note overflows
  - B-trees grow upwards
Intermediate 1

...  

...  

32 38 39  -  -  

50  75  

40?  

45 49  -  -  -  

76 85 88 91  -  

51 55 58  -  -  
Intermediate 2
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 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
    • Works as well for largest value from left subtree
  - 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*
    • One block with records plus middle value 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

- Borrow from right subtree
- Underflow

```
30
10 -
...
```

```
50 -
45 -
...
```

```
32 38 39 - -
```

```
75 -
45 -
...
```

```
76 85 88 91 -
```

```
51 55 58 - -
```

```
49 - - - - -
```

• Borrow from right subtree
• Underflow
Delete with Underflow

- Merge with left neighbor
Delete with Underflow

- Delete 45
- Underflow
- No local repair
Delete with Underflow

- Merge blocks
- Parent underflows

```
  30
  50
  75
  76 85 88 91
  51 55 58
```

```
  32 38 39 49
  10 -
  50 -
  ...  
  ...  
  32 38 39 49 -
```

```
  ...  
  ...  
  ...  
```

```
  76 85 88 91 -
  51 55 58 - -
```
Delete with Underflow

- Up the tree
Complexity of Deleting Keys

• Going down costs $h+1$ IO at most
  - 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 $h+2$
• If local underflow (with merge), total cost is $\sim h+6$
  - 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
  - Similar argument as for insertion
B-trees on Non-Unique Attributes

- **Option 1: Compact representation**
  - Store (value, TID₁, TID₂, ... TIDₙ)
  - 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
  - Keys are only inserted at leaf nodes
  - 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
  - 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
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 as 4/3k keys: Guaranteed 66% space usage

B+ Trees and Hashing

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

- B+ trees
  - Very few IO if upper levels are cached
  - Adapts to skewed (non-uniformly distributed) data
  - More robust, domain-independent
  - Also support 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

\[
\text{create index myidx on LARGETABLE( id);}
\]

• Naïve: **Record-by-record** insertion
  - Each insertion has \(3h+2 = O(\log_k(b))\) 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
  - $O(n \cdot \log_m(n))$, 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 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 usage, higher fan-out
Disadvantages: Overhead for computing signpost (more IO)
Variable-length records in internal nodes
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

• Prefix/Patricia 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

- 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: $t \not\in S$
  - 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
- **Idea**
  - Depth-first search on the tree as usual
  - Keep a counter for the number of mismatches spent in the prefix so far
  - If counter exceeds threshold – stop search in this branch
  - **Pruning:** Try to stop early
Example: Search

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

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

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

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

$$d(CTGAAATTTGCT, CTGAAATTTGCT) > 1$$
Example: Search

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

Hamming distance search for \( t = CTGAAATTGGT \), \( k=1 \)
(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
- Exact and similarity join
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

<table>
<thead>
<tr>
<th>Set</th>
<th># EST strings</th>
<th>avg. string length</th>
<th>min/max length</th>
<th># tree nodes</th>
<th># ext. suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>307,542</td>
<td>348</td>
<td>$14/3,615$</td>
<td>589,062</td>
<td>293,764</td>
</tr>
<tr>
<td>$T_2$</td>
<td>736,305</td>
<td>387</td>
<td>$12/3,707$</td>
<td>1,482,709</td>
<td>689,590</td>
</tr>
<tr>
<td>$T_{2a}$</td>
<td>368,152</td>
<td>382</td>
<td>$12/2,774$</td>
<td>711,632</td>
<td>352,872</td>
</tr>
<tr>
<td>$T_{2b}$</td>
<td>184,076</td>
<td>385</td>
<td>$22/2,774$</td>
<td>349,329</td>
<td>177,846</td>
</tr>
<tr>
<td>$T_{2c}$</td>
<td>92,038</td>
<td>383</td>
<td>$25/2,774$</td>
<td>171,964</td>
<td>89,198</td>
</tr>
<tr>
<td>$T_{2d}$</td>
<td>46,019</td>
<td>381</td>
<td>$28/2,774$</td>
<td>84,954</td>
<td>44,716</td>
</tr>
<tr>
<td>$T_{2e}$</td>
<td>23,009</td>
<td>373</td>
<td>$31/878$</td>
<td>42,375</td>
<td>22,366</td>
</tr>
<tr>
<td>$T_3$</td>
<td>10,000</td>
<td>536</td>
<td>$16/3,707$</td>
<td>16,310</td>
<td>8,774</td>
</tr>
<tr>
<td>TX</td>
<td>5,000,000</td>
<td>359</td>
<td>14/3,247</td>
<td>10,478,214</td>
<td>4,834,231</td>
</tr>
</tbody>
</table>
Search: Comparing to Flamingo (2011)

- Flamingo: Library for approximate string matching
  - 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.