Notes 4: Indexing

Arthur Keller

Topics
- Conventional indexes
- B-trees
- Hashing schemes

Sequential File

Dense Index

Sparse Index

Chapter 4

Indexing & Hashing

value → record

value
Sequential File

Sparse 2nd level

- Comment:
  \{FILE, INDEX\} may be contiguous or not (blocks chained)

Question:
- Can we build a dense, 2nd level index for a dense index?

Notes on pointers:
(1) Block pointer (sparse index) can be smaller than record pointer

Notes on pointers:
(2) If file is contiguous, then we can omit pointers (i.e., compute them)
Sparse vs. Dense Tradeoff

- **Sparse**: Less index space per record can keep more of index in memory
- **Dense**: Can tell if any record exists without accessing file

(Later:
- sparse better for insertions
- dense needed for secondary indexes)

Terms

- Index sequential file
- Search key (≠ primary key)
- Primary index (on Sequencing field)
- Secondary index
- Dense index (all Search Key values in)
- Sparse index
- Multi-level index

Next:

- Duplicate keys
- Deletion/Insertion
- Secondary indexes

Duplicate keys

Dense index, one way to implement?

| 10 | 10 |
| 20 | 20 |
| 30 | 30 |
| 40 | 40 |

Dense index, better way?

| 10 | 10 |
| 20 | 20 |
| 30 | 30 |
| 40 | 45 |
Sparse index, one way?

Duplicate keys

Sparse index, another way?

- place first new key from block

should this be 40?

Duplicate values, primary index

- Index may point to first instance of each value only

Deletion from sparse index

- delete record 40

Deletion from sparse index

- delete record 30
Deletion from sparse index
- delete records 30 & 40

Deletion from dense index
- delete record 30

Deletion from dense index
- delete records 30 & 40

Insertion, sparse index case
- insert record 34

- our lucky day!
  we have free space
  where we need it!

Insertion, sparse index case
- insert record 15

- Illustrated: Immediate reorganization
- Variation:
  - insert new block (chained file)
  - update index
Insertion, sparse index case

- insert record 25

overflow blocks (reorganize later...)

Insertion, dense index case

- Similar
- Often more expensive . . .

Secondary indexes

- Sparse index

does not make sense!

With secondary indexes:

- Lowest level is dense
- Other levels are sparse

Also: Pointers are record pointers (not block pointers; not computed)
Duplicate values & secondary indexes

Problem:
excess overhead!
- disk space
- search time

Another idea (suggested in class):
Chain records with same key?

Problems:
- Need to add fields to records
- Need to follow chain to know records

Why “bucket” idea is useful

Indexes Records
Name: primary EMP (name,dept,floor,....)
Dept: secondary
Floor: secondary
Query: Get employees in (Toy Dept) ∩ (2nd floor)

This idea used in text information retrieval

IR QUERIES
- Find articles with “cat” and “dog”
- Find articles with “cat” or “dog”
- Find articles with “cat” and not “dog”
- Find articles with “cat” in title
- Find articles with “cat” and “dog” within 5 words

Common technique: more info in inverted list

Posting: an entry in inverted list. Represents occurrence of term in article

IR DISCUSSION
- Stop words
- Truncation
- Thesaurus
- Full text vs. Abstracts
- Vector model
Vector space model

\[
\begin{align*}
\text{DOC} &= <1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ ... > \\
\text{Query} &= <0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ ... > \\
\text{PRODUCT} &= 1 + \ldots = \text{score}
\end{align*}
\]

• Tricks to weigh scores + normalize
  
e.g.: Match on common word not as useful as match on rare words...

• How to process VS. Queries?

\[
\begin{align*}
\text{w1} \ \text{w2} \ \text{w3} \ \text{w4} \ \text{w5} \ \text{w6} \ \ldots \\
\text{Q} &= <0 \ 0 \ 0 \ 1 \ 1 \ 0 \ \ldots >
\end{align*}
\]

• Try Altavista, Excite, Infoseek, Lycos...

Summary so far

• Conventional index
  
  – Basic Ideas: sparse, dense, multi-level...
  – Duplicate Keys
  – Deletion/Insertion
  – Secondary indexes
    – Buckets of Postings List

Conventional indexes

Advantage:
  - Simple
  - Index is sequential file
    good for scans

Disadvantage:
  - Inserts expensive, and/or
    Lose sequentiality & balance
Example Index (sequential)

Continuous free space

overflow area (not sequential)

Outline:

- Conventional indexes
- B-Trees \( \Rightarrow \) NEXT
- Hashing schemes

• NEXT: Another type of index
  - Give up on sequentiality of index
  - Try to get "balance"

B+Tree Example

Sample non-leaf

Sample leaf node:
In textbook’s notation  

n = 3 

Leaf:  

Non-leaf:  

Size of nodes: 

\[ n+1 \text{ pointers} \]  
\[ n \text{ keys} \]  

(fixed) 

Don’t want nodes to be too empty  

- Use at least 

\[ \left\lceil \frac{n+1}{2} \right\rceil \text{ pointers} \]  

\[ \left\lfloor \frac{n+1}{2} \right\rfloor \text{ pointers to data} \]  

B+tree rules  

Tree of order \( n \)  

(1) All leaves at same lowest level  

(balanced tree)  

(2) Pointers in leaves point to records  

except for “sequence pointer”  

(3) Number of pointers/keys for B+tree 

<table>
<thead>
<tr>
<th></th>
<th>Max ptrs</th>
<th>Max keys</th>
<th>Min ptrs - data</th>
<th>Min keys</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-leaf (non-root)</td>
<td>( n+1 )</td>
<td>( n )</td>
<td>( \left\lceil \frac{n+1}{2} \right\rceil )</td>
<td>( \left\lfloor \frac{n+1}{2} \right\rfloor ) - 1</td>
<td></td>
</tr>
<tr>
<td>Leaf (non-root)</td>
<td>( n+1 )</td>
<td>( n )</td>
<td>( \left\lceil \frac{n+1}{2} \right\rceil )</td>
<td>( \left\lfloor \frac{n+1}{2} \right\rfloor )</td>
<td></td>
</tr>
<tr>
<td>Root</td>
<td>( n+1 )</td>
<td>( n )</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Insert into B+tree
(a) simple case
   - space available in leaf
(b) leaf overflow
(c) non-leaf overflow
(d) new root

(a) Insert key = 32
\[ n=3 \]

(b) Coalesce with neighbor (sibling)
(c) Re-distribute keys
(d) Cases (b) or (c) at non-leaf

Deletion from B+tree
(a) Simple case - no example
(b) Coalesce with neighbor (sibling)
(c) Re-distribute keys
(d) Cases (b) or (c) at non-leaf
(b) Coalesce with sibling
- Delete 50

(c) Redistribute keys
- Delete 50

(d) Non-leaf coalesce
- Delete 37

B+tree deletions in practice
- Often, coalescing is not implemented
  - Too hard and not worth it!

Comparison: B-trees vs. static indexed sequential file

Ref #1: Held & Stonebraker
  "B-Trees Re-examined"
  CACM, Feb. 1978

Ref #1 claims:
- Concurrency control harder in B-Trees
- B-tree consumes more space

For their comparison:
  block = 512 bytes
  key = pointer = 4 bytes
  4 data records per block
Example: 1 block static index

127 keys

\[(127 + 1) \times 4 = 512 \text{ Bytes}\]

-> pointers in index implicit! up to 127 blocks

Example: 1 block B-tree

63 keys

\[63 \times (4 + 4) + 8 = 512 \text{ Bytes}\]

-> pointers needed in B-tree blocks because index is not contiguous

Size comparison

<table>
<thead>
<tr>
<th>Static Index</th>
<th>Ref. #1</th>
</tr>
</thead>
<tbody>
<tr>
<td># data blocks</td>
<td>height</td>
</tr>
<tr>
<td>2 (\rightarrow) 127</td>
<td>2</td>
</tr>
<tr>
<td>128 (\rightarrow) 16,129</td>
<td>3</td>
</tr>
<tr>
<td>16,130 (\rightarrow) 2,048,383</td>
<td>4</td>
</tr>
</tbody>
</table>
| 250,048 \(\rightarrow\) 15,752,961 | 5

Ref. #1 analysis claims

- For an 8,000 block file, after 32,000 inserts after 16,000 lookups

\[\Rightarrow\] Static index saves enough accesses to allow for reorganization

Ref. #1 conclusion Static index better!!

Ref #2: M. Stonebraker, “Retrospective on a database system,” TODS, June 1980

Ref. #2 conclusion B-trees better!!

- DBA does not know when to reorganize
- DBA does not know how full to load pages of new index

Ref. #2 conclusion B-trees better!!
• Buffering
  – B-tree: has fixed buffer requirements
  – Static index: must read several overflow blocks to be efficient
    (large & variable size buffers needed for this)

Ref. #2 conclusion B-trees better!!

• Speaking of buffering...
  Is LRU a good policy for B+tree buffers?
  → Of course not!
  → Should try to keep root in memory at all times
    (and perhaps some nodes from second level)

Interesting problem:
For B+tree, how large should n be?

\[ n \text{ is number of keys / node} \]

Sample assumptions:
(1) Time to read node from disk is 
    \((70+0.05n)\) msec.
(2) Once block in memory, use binary search to locate key:
    \((a + b \log_2 n)\) msec.
    For some constants \(a, b\): Assume \(a << 70\)
(3) Assume B+tree is full, i.e.,
    # nodes to examine is \(\log_n N\)
    where \(N = \#\) records

\[ f(n) = \text{time to find a record} \]

FIND \(n_{opt}\) by \(f'(n) = 0\)
Answer is \(n_{opt} = \text{“few hundred”}\)
(see homework for details)

What happens to \(n_{opt}\) as
  • Disk gets faster?
  • CPU get faster?
Variation on B+tree: B-tree (no +)

- Idea:
  - Avoid duplicate keys
  - Have record pointers in non-leaf nodes

B-tree example

- sequence pointers not useful now!
  (but keep space for simplicity)

Note on inserts

- Say we insert record with key = 25

So, for B-trees:

<table>
<thead>
<tr>
<th></th>
<th>MAX</th>
<th>MIN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tree Ptrs</td>
<td>Rec Ptrs</td>
</tr>
<tr>
<td>Non-leaf non-root</td>
<td>n+1</td>
<td>n</td>
</tr>
<tr>
<td>Leaf non-root</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>Root non-leaf</td>
<td>n+1</td>
<td>n</td>
</tr>
<tr>
<td>Root Leaf</td>
<td>1</td>
<td>n</td>
</tr>
</tbody>
</table>

Tradeoffs:

- B-trees have faster lookup than B+trees
- in B-tree, non-leaf & leaf different sizes
- in B-tree, deletion more complicated

∴ B+trees preferred!
But note:
• If blocks are fixed size (due to disk and buffering restrictions)
  Then lookup for B+tree is actually better!!

Example:
- Pointers 4 bytes
- Keys 4 bytes
- Blocks 100 bytes (just example)
- Look at full 2 level tree

Example:
- Pointers 4 bytes
- Keys 4 bytes
- Blocks 100 bytes (just example)
- Look at full 2 level tree

B-tree:
Root has 8 keys + 8 record pointers + 9 son pointers
= 8x4 + 8x4 + 9x4 = 100 bytes
Each of 9 sons: 12 rec. pointers (+12 keys)
= 12x(4+4) + 4 = 100 bytes
2-level B-tree, Max # records = 12x9 + 8 = 116

B+tree:
Root has 12 keys + 13 son pointers
= 12x4 + 13x4 = 100 bytes
Each of 13 sons: 12 rec. ptrs (+12 keys)
= 12x(4 +4) + 4 = 100 bytes
2-level B+tree, Max # records = 13x12 = 156

So...

8 records

B+
156 records

B
108 records
Total = 116

• Conclusion:
  – For fixed block size,
  – B+ tree is better because it is bushier

Outline/summary
• Conventional Indexes
  • Sparse vs. dense
  • Primary vs. secondary
• B trees
  • B+trees vs. B-trees
  • B+trees vs. indexed sequential
• Hashing schemes --> Next