**CS 245: Database System Principles**

**Chapter 4**

**Indexing & Hashing**

value → ? → record

<table>
<thead>
<tr>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Notes 4: Indexing**

Hector Garcia-Molina
(Some modifications by Chris Olston)

**Topics**

- Conventional indexes
- B-trees
- Hashing schemes

**First...**

- Primary Indexes
  - For your ordered files only!

**Sequential File**

| 10 | 20 |
| 35 |
| 50 |
| 70 |
| 90 |

**Dense Index**

| 00 |
| 10 |
| 20 |

**Sequential File**

| 00 |
| 10 |
| 20 |
- 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 file” (sequential)
- 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? (pointer to each record)
Duplicate keys
Dense index, better way?

Duplicate keys
Sparse index, one way?
(pointer to each block)

Duplicate keys
Sparse index, another way?
place first new key from block
should this be 40?

Summary
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 40
**Insertion, sparse index case**

- insert record 15

```
  10  20  30  40  50
  20  30  40  50  60
```

- Illustrated: Immediate reorganization
- Variation:
  - Insert new block (chained)
  - Possibly update index

---

**Insertion, sparse index case**

- insert record 25

```
  10  20  30  40  50
  20  30  40  50  60
  25
```

overflow blocks (reorganize later...)

---

**Insertion, dense index case**

- Similar
- Often more expensive...

---

**Secondary indexes**

- Sparse index

```
  30  20  10  50  60
  20  40  60
```

does not make sense!

---

**Secondary indexes**

- Dense index

```
  10  20  30  40  50
  20  30  40  50  60
```

spare high level

---
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
one option...

Problem:
excess overhead!
- disk space
- search time

Duplicate values & secondary indexes
another option...

Problem:
variable size
records in
index!

Duplicate values & secondary indexes

Duplicate values & secondary indexes
Why "bucket" idea is useful

Indexes
Name: primary
Dept: secondary
Floor: secondary

Records
EMP (name, dept, floor, ...)

This idea used in text information retrieval

Documents
...the cat is fat ...
...was raining cats and dogs...
...Fido the dog ...

Inverted lists

cat

dog

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

Size of a list: 1 Rare words or (in # of postings) miss-spellings
10^6 Common words

Size of a posting: 10-15 bits (compressed)
**IR DISCUSSION**

- Stop words
- Stemming
- Thesaurus
- Full text vs. Abstracts
- Vector model

**Vector space model**

\[
\text{DOC} = \langle w_1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \rangle
\]

\[
\text{Query} = \langle 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \rangle
\]

\[
\text{PRODUCT} = 1 + \ldots = \text{score}
\]

**Tricks to weigh scores + normalize**

e.g.: Match on common word not as useful as match on rare words... (generalization of stop words)

⇒ Take into account document frequency

**How to process V.S. Queries?**

\[
\text{Q} = \langle 0 \ 0 \ 0 \ 1 \ 1 \ 0 \rangle
\]

**Summary so far**

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

- Try Folio
- Try AltaVista, Excite, Infoseek, Lycos...
Conventional indexes

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

**Disadvantage:**
- Inserts expensive, and/or
  - Lose sequentiality & balance

---

Outline:

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

---

Example

**Index (sequential)**

- Continuous
- Free space
- Overflow area (not sequential)

---

**NEXT:** Another type of index
- Give up on sequentiality of index
- Try to get “balance”

---

**B+Tree Example**

\( n=3 \)

---

**Sample non-leaf**

- to keys
- to keys
- to keys
- to keys

- \(< 57 \)
- \(57 \leq k < 81 \)
- \(81 \leq k < 95 \)
- \(\geq 95 \)
Range queries

- We’ve seen how B+Trees can support fast lookups for equality predicates.
- What about range predicates?
  - $20 \leq K \leq 40$
  - $K \geq 20$
  - $K \leq 30$

Don’t want nodes to be too empty

- Use at least
  - Non-leaf: \( \lceil (n+1)/2 \rceil \) pointers
  - Leaf: \( \lfloor (n+1)/2 \rfloor \) pointers to data
**B+tree rules**

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 active ptrs</th>
<th>Min keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-leaf (not root)</td>
<td>n+1</td>
<td>n</td>
<td>⌈(n+1)/2⌉</td>
<td>⌈(n+1)/2⌉+1</td>
</tr>
<tr>
<td>Leaf (not root)</td>
<td>n+1</td>
<td>⌈(n+1)/2⌉</td>
<td>⌈(n+1)/2⌉</td>
<td></td>
</tr>
<tr>
<td>Root</td>
<td>n+1</td>
<td>n</td>
<td>1</td>
<td>1</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

**Insert key = 7**

**Insert key = 32**

**Insert key = 160**
B-Tree property

- Grows "up"
  - by splitting toward root
- Stays balanced

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

(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!

- **Middle ground:**
  - 1. Try to steal from neighbor (redistribute)
  - 2. If nothing available at neighbors, give up

---

**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**

<table>
<thead>
<tr>
<th>Keys</th>
<th>1 data block</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td></td>
</tr>
<tr>
<td>k2</td>
<td></td>
</tr>
<tr>
<td>k3</td>
<td></td>
</tr>
</tbody>
</table>

127 keys  
(127+1)/4 = 512 Bytes  
-> pointers in index implicit!  
up to 127 blocks

---

**Example: 1 block B-tree**

<table>
<thead>
<tr>
<th>Keys</th>
<th>1 data block</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td></td>
</tr>
<tr>
<td>k2</td>
<td></td>
</tr>
<tr>
<td>k3</td>
<td></td>
</tr>
</tbody>
</table>

63 keys  
63*(4+4)+8 = 512 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 -&gt; 127</td>
<td>2</td>
</tr>
<tr>
<td>128 -&gt; 16,129</td>
<td>3</td>
</tr>
<tr>
<td>16,130 -&gt; 2,048,303</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># data blocks</th>
<th>height</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 -&gt; 63</td>
<td>2</td>
</tr>
<tr>
<td>64 -&gt; 3968</td>
<td>3</td>
</tr>
<tr>
<td>5909 -&gt; 250,047</td>
<td>4</td>
</tr>
<tr>
<td>250,048 -&gt; 15,752,961</td>
<td>5</td>
</tr>
</tbody>
</table>
Ref. #1 analysis claims

- For an 8,000 block file, 
  after 32,000 inserts 
  after 16,000 lookups
⇒ 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
- B-Trees are self-tuning ("no-knobs")

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)

• 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?

$n$ 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

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

\[ n_{opt} \]

\[ n \]

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

\[ \gg \text{What happens to } n_{opt} \text{ as} \]
- 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 nearly 
(but keep space for simplicity)

\[ n=2 \]
Tradeoffs:

- B-trees have faster lookup than B+ trees (will get back to this ...)
- in B-tree, non-leaf & leaf different sizes
- in B-tree, deletion more complicated
- in B-tree, no leaf sequence pointers

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

[Chug, chug, chug ...]

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

Punch line...

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