CS 277: Database System Implementation

Notes 4: Indexing

Arthur Keller
Chapter 4

Indexing & Hashing

value → ? → value

record
Topics

• Conventional indexes
• B-trees
• Hashing schemes
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sparse 2nd level

Sequential File

| 10 | 10 |
| 90 | 20 |
| 170 | 30 |
| 250 | 40 |
| 330 | 50 |
| 410 | 70 |
| 490 | |
| 570 | |

| 170 | 90 |
| 190 | 110 |
| 210 | 130 |
| 230 | 150 |
| 30 | 50 |
| 490 | 60 |
| 570 | 70 |
| 100 | 80 |
| 130 | 90 |
• 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)
• if we want K3 block:
  get it at offset
  \((3-1)1024\)
  \(= 2048\) bytes

say:
1024 B per block
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
Duplicate keys

Dense index, one way to implement?
Duplicate keys

Dense index, better way?

10
20
30
40

10
10

10
20

20
30

30
30

40
45
Duplicate keys
Sparse index, one way?

careful if looking for 20 or 30!
## Duplicate keys

**Sparse index, another way?**

- place first new key from block

<table>
<thead>
<tr>
<th>10</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>30</td>
<td>20</td>
</tr>
</tbody>
</table>

Should this be 40?
Duplicate values, primary index

- Index may point to **first** instance of each value only

```
Index
  a

File
  a
  a
  .
  .
  b
```
Deletion from sparse index
Deletion from sparse index

– delete record 40
Deletion from sparse index

- delete record 30

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

<table>
<thead>
<tr>
<th>10</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>40</td>
</tr>
<tr>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>70</td>
<td>80</td>
</tr>
</tbody>
</table>
Deletion from sparse index
– delete records 30 & 40
Deletion from dense index
Deletion from dense index

- delete record 30

```
<table>
<thead>
<tr>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
</tr>
<tr>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>40</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>30</th>
<th>40</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
</tr>
<tr>
<td>70</td>
</tr>
<tr>
<td>80</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
</tr>
</tbody>
</table>
```
Insertion, sparse index case

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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...)

10
30
40
60

10
20

30

40
50

60
Insertion, dense index case

- Similar

- Often more expensive . . .
Secondary indexes

Sequence field

30
50
20
70
80
40
100
10
90
60
Secondary indexes

• Sparse index

does not make sense!
Secondary indexes

- Dense index

```
  10
  50
  90
  ...

  10
  20
  30
  40

  50
  60
  70
  ...
```

Sequence field

```
  30
  50

  20
  70

  80
  40

  100
  10
  90
  60
```
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
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

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

Problems:
- Need to add fields to records
- Need to follow chain to know records
Duplicate values & secondary indexes

10 -> 20
20 -> 30
30 -> 40
40
50
60 ...

20
10
20
40
10
40
10
40
30
40

buckets
Why “bucket” idea is useful

Indexes
Name: primary
Dept: secondary
Floor: secondary

Records
EMP (name,dept,floor,...)
Query: Get employees in 
(Toy Dept) \( \land \) (2nd floor)

→ Intersect toy bucket and 2nd Floor bucket to get set of matching EMP’s
This idea used in text information retrieval

Inverted lists

Documents

...the cat is fat ...

...was raining cats and dogs...

...Fido the 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

<table>
<thead>
<tr>
<th>type</th>
<th>position</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Author</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Abstract</td>
<td>57</td>
<td></td>
</tr>
</tbody>
</table>

cat →

<table>
<thead>
<tr>
<th>type</th>
<th>position</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Title</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

dog →

<table>
<thead>
<tr>
<th>type</th>
<th>position</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d_2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d_3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Posting: an entry in inverted list. Represents occurrence of term in article

Size of a list: 1 Rare words or miss-spellings
(in postings)

10^6 Common words

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

• Stop words
• Truncation
• Thesaurus
• Full text vs. Abstracts
• Vector model
Vector space model

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

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

\[ \text{PRODUCT} = 1 + \ldots = \text{score} \]
• Tricks to weigh scores + normalize

e.g.: Match on common word not as useful as match on rare words...
• How to process VS. Queries?

\[ w_1 \ w_2 \ w_3 \ w_4 \ w_5 \ w_6 \ \ldots \]
\[ Q = \langle 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ \ldots \rangle \]
• 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)

<table>
<thead>
<tr>
<th>10</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>33</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

overflow area
(not sequential)

continuous
free space
Outline:

- Conventional indexes
- B-Trees
- Hashing schemes
• NEXT: Another type of index
  – Give up on sequentiality of index
  – Try to get “balance”
B+Tree Example

Root

3 5 11

30 35

100 101 110

120 130

150 156 1579

180 200

n=3
Sample non-leaf

- to keys
- < 57
- 57 ≤ k < 81
- 81 ≤ k < 95
- ≥ 95
Sample leaf node:

From non-leaf node to next leaf in sequence.

To record with key 57
To record with key 81
To record with key 85
In textbook’s notation

n = 3

Leaf:

Non-leaf:
Size of nodes: 

\[
\begin{align*}
\text{n+1 pointers} & \quad \text{(fixed)} \\
\text{n keys} & 
\end{align*}
\]
Don’t want nodes to be too empty

- Use at least

  Non-leaf: \[\lceil \frac{n+1}{2} \rceil \] pointers

  Leaf: \[\lfloor \frac{n+1}{2} \rfloor \] pointers to data
n=3

Non-leaf

Leaf

Full node

min. node

counts even if null
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>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Non-leaf</strong></td>
<td>n+1</td>
<td>n</td>
<td>⌈(n+1)/2⌉</td>
<td>⌈(n+1)/2⌉- 1</td>
</tr>
<tr>
<td>(non-root)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Leaf</strong></td>
<td>n+1</td>
<td>n</td>
<td>⌊(n+1)/2⌋</td>
<td>⌊(n+1)/2⌋</td>
</tr>
<tr>
<td>(non-root)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Root</strong></td>
<td>n+1</td>
<td>n</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></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
(a) Insert key = 7
(c) Insert key = 160

```
100 160
120 150 180
150 156 179
180
180 200
```

n=3
(d) New root, insert 45

n=3
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

\[ n=4 \]
(c) Redistribute keys
   – Delete 50

\[ n=4 \]
(d) Non-leaf coalesce
- Delete 37

new root

n=4
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)4 = 512 Bytes
-> pointers in index implicit!

1 data block

up to 127 blocks
Example: 1 block B-tree

63 keys

63x(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>B-tree</th>
</tr>
</thead>
<tbody>
<tr>
<td># data blocks</td>
<td># data blocks</td>
</tr>
<tr>
<td>height</td>
<td>height</td>
</tr>
</tbody>
</table>

| 2 -> 127 | 2 |
| 128 -> 16,129 | 3 |
| 16,130 -> 2,048,383 | 4 |
| 2 -> 63 | 2 |
| 64 -> 3968 | 3 |
| 3969 -> 250,047 | 4 |
| 250,048 -> 15,752,961 | 5 |
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!!
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
• 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?

\( n \) 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
Can get:

\[ f(n) = \text{time to find a record} \]
- 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
to record with K1

K1 P1

to record with K2

K2 P2

to record with K3

K3 P3

< K1< K1 < x < K2

to keys

K2 < x < k3

to keys

> k3

to keys

B-tree example

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

• Say we insert record with key = 25

Afterwards:
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</td>
<td>(n+1)</td>
<td>n</td>
</tr>
<tr>
<td>non-root</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leaf</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>non-root</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Root</td>
<td>n+1</td>
<td>n</td>
</tr>
<tr>
<td>non-leaf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Root</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>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
B-tree:

Root has 8 keys + 8 record pointers + 9 son pointers
= 8\times4 + 8\times4 + 9\times4 = 100 \text{ bytes}

Each of 9 sons: 12 rec. pointers (+12 keys)
= 12\times(4+4) + 4 = 100 \text{ bytes}

2-level B-tree, Max # records = 12\times9 + 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...

- 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