CS 245: Database System Principles

Notes 4: Indexing

Hector Garcia-Molina

Topics
• Conventional indexes
• B-trees
• Hashing schemes

Sequential File

| 10 | 20 |
| 30 | 40 |
| 50 | 60 |
| 70 | 80 |
| 90 | 100 |

Dense Index

| 10 |
| 20 |
| 30 |
| 40 |
| 50 |
| 60 |
| 70 |
| 80 |
| 90 |
| 100 |

Sparse Index

| 10 |
| 20 |
| 30 |
| 40 |
| 50 |
| 60 |
| 70 |
| 80 |
| 90 |
| 100 |
**Sequential File**

```
10  10
20  20
30  30
40  40
50  50
60  60
70  70
80  80
90  90
100 100
```

**Sparse 2nd level**

```
10   10
90   90
170  170
250  250
330  330
410  410
490  490
570  570
```

- **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

   - BP
   - RP

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?
Duplicate keys
Dense index, better way?

Duplicate keys
Sparse index, one way?

Duplicate keys
Sparse index, another way?

Sparse index, one way?

Sparse index, another way?

Duplicate keys
Sparse index, another way?

- place first new key from block

Duplicate values, primary index

- Index may point to first instance of each value only

Summary

File

Index

a

a

. 

b
Deletion from sparse index

- delete record 40

Deletion from sparse index

- delete record 30

Deletion from sparse index

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

Deletion from dense index
- delete record 30

Deletion from dense index
- delete record 30
Insertion, sparse index case

- insert record 34

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

- insert record 15

• Illustrated: Immediate reorganization

• Variation:
  - insert new block (chained file)
  - update index
**Insertion, sparse index case**

- insert record 25

---

**Insertion, sparse index case**

- insert record 25

overflow blocks (reorganize later...)

---

**Insertion, dense index case**

- Similar
- Often more expensive . . .

---

**Secondary indexes**

- Sparse index

---

**Secondary indexes**

- Sparse index

does not make sense!
Secondary indexes

- Dense index

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...
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 option...

Duplicate values & secondary indexes

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

Problem:
  • Need to add fields to records
  • Need to follow chain to know records

Duplicate values & secondary indexes

Another idea (suggested in class):
Chain records with same key?
Why “bucket” idea is useful

Indexes | Records
---|---
Name: primary EMP (name,dept,floor,...) | Dept: secondary
Floor: secondary

Query: Get employees in (Toy Dept) \( \land \) (2nd floor)

\[ \text{Dept. index EMP Floor index} \]

\[ \text{Toy} \quad \text{Toy} \quad 2nd \]

→ Intersect toy bucket and 2nd Floor bucket to get set of matching EMP’s

This idea used in text information retrieval

Documents

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

Inverted lists

IR QUERIES

- Find articles with “cat” and “dog”
- Find articles with “cat” or “dog”
- Find articles with “cat” and not “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 miss-spellings
- Size of a posting: 10-15 bits (compressed)

**IR DISCUSSION**

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

**Vector space model**

\[
\begin{align*}
\text{w1 w2 w3 w4 w5 w6 w7 ...} \\
\text{DOC} &= \langle 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ ... \rangle \\
\text{Query} &= \langle 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ ... \rangle \\
\text{PRODUCT} &= 1 + \ldots = \text{score}
\end{align*}
\]
• Tricks to weigh scores + normalize
  e.g.: Match on common word not as useful as match on rare words...

• How to process V.S. Queries?
  \[ w_1 \ w_2 \ w_3 \ w_4 \ w_5 \ w_6 \ \ldots \]
  \[ Q = < 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ \ldots > \]

• Try Stanford Libraries
• Try Google, Yahoo, ...

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

Note: This index is called B+ tree, but Gradiance homeworks just call it B-tree.
In textbook’s notation \( n=3 \)

**Leaf:**

**Non-leaf:**

<table>
<thead>
<tr>
<th>Size of nodes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n+1 ) pointers</td>
</tr>
<tr>
<td>( n ) keys</td>
</tr>
</tbody>
</table>

Don’t want nodes to be too empty

- Use at least

  \[ \lceil (n+1)/2 \rceil \] pointers

  \[ \lfloor (n+1)/2 \rfloor \] 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”

<table>
<thead>
<tr>
<th>(3) Number of pointers/keys for B+tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max ptrs</td>
</tr>
<tr>
<td>Non-leaf (non-root)</td>
</tr>
<tr>
<td>Leaf (non-leaf)</td>
</tr>
<tr>
<td>Root</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

(a) Insert key = 7
(c) Insert key = 160

(d) New root, insert 45
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!
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!

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>B-tree</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; 2,048,383</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 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,
"Retrospection 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

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?

$n$ is number of keys / node

Sample assumptions:
(1) Time to read node from disk is $(S+Tn)$ msec.

Sample assumptions:
(1) Time to read node from disk is $(S+Tn)$ msec.
(2) Once block in memory, use binary search to locate key:
    $(a + b \log_2 n)$ msec.
    For some constants $a, b$: Assume $a << S$

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

Can get:
$f(n) = \text{time to find a record}$

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

Exercise

$$f(n) = \log_n N \times [S + T \times n + a + b \times \log_2(n)]$$

<table>
<thead>
<tr>
<th>$S$</th>
<th>14000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>0.2</td>
</tr>
<tr>
<td>$a$</td>
<td>0</td>
</tr>
<tr>
<td>$b$</td>
<td>0.002</td>
</tr>
<tr>
<td>$N$</td>
<td>10000000</td>
</tr>
</tbody>
</table>

N=10 million records

Variation on B+tree: B-tree (no +)

- Idea:
  - Avoid duplicate keys
  - Have record pointers in non-leaf nodes
to record with keys < K1
K1 P1
K2 P2
K3 P3

with K1
K1<x<K2
K2<x<k3

> k3
to keys

to record with K2

to record with K3

B-tree example

n=2

• 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>Non-leaf</th>
<th>MAX</th>
<th></th>
<th>MIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Root</td>
<td></td>
<td></td>
<td></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

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
= 8x4 + 8x4 + 9x4 = 100 bytes

Each of 9 sons: 12 rec. pointers (+12 keys)
= 12x(4+4) + 4 = 100 bytes
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...

B+

156 records

B

8 records

108 records

Total = 116

---

B+

8 records

156 records

B

108 records

Total = 116

• Conclusion:
- For fixed block size,
- B+ tree is better because it is bushier
An Interesting Problem...

- What is a good index structure when:
  - records tend to be inserted with keys that are larger than existing values?
    (e.g., banking records with growing data/time)
  - we want to remove older data

One Solution: Multiple Indexes

- Example: I1, I2

<table>
<thead>
<tr>
<th>day</th>
<th>days indexed</th>
<th>days indexed</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>1, 2, 3, 4, 5</td>
<td>6, 7, 8, 9, 10</td>
</tr>
<tr>
<td>I2</td>
<td>11, 12, 3, 4, 5</td>
<td>6, 7, 8, 9, 10</td>
</tr>
</tbody>
</table>

- advantage: deletions/insertions from smaller index
- disadvantage: query multiple indexes

Another Solution (Wave Indexes)

<table>
<thead>
<tr>
<th>day</th>
<th>I1</th>
<th>I2</th>
<th>I3</th>
<th>I4</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1, 2, 3</td>
<td>4, 5, 6</td>
<td>7, 8, 9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>1, 2, 3</td>
<td>4, 5, 6</td>
<td>7, 8, 9</td>
<td>10, 11, 12</td>
</tr>
<tr>
<td>12</td>
<td>1, 2, 3</td>
<td>4, 5, 6</td>
<td>7, 8, 9</td>
<td>10, 11, 12</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>4, 5, 6</td>
<td>7, 8, 9</td>
<td>10, 11, 12</td>
</tr>
<tr>
<td>14</td>
<td>13, 14</td>
<td>4, 5, 6</td>
<td>7, 8, 9</td>
<td>10, 11, 12</td>
</tr>
<tr>
<td>15</td>
<td>13, 14, 15</td>
<td>4, 5, 6</td>
<td>7, 8, 9</td>
<td>10, 11, 12</td>
</tr>
<tr>
<td>16</td>
<td>13, 14, 15</td>
<td>16</td>
<td>7, 8, 9</td>
<td>10, 11, 12</td>
</tr>
</tbody>
</table>

- advantage: no deletions
- disadvantage: approximate windows

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