Indexing with Hashing

- primary key indexing
- B-trees and variants
- (static) hashing
- extendible hashing
- secondary key indexing
- spatial access methods
- text
- ...

(Static) Hashing

- Problem: “find EMP record with SSN=123”
- What if disk space was free, and time was at premium?

Hashing

Since space is NOT free:
- use M, instead of 999,999,999 slots
- hash function: h(key) = slot-id
- Typically: hash bucket = page with many records

Hashing

Can have clustering or non-clustering versions
Hashing

Can have clustering or non-clustering versions

EMP file

Hashing – Design Decisions

E.g., IRS, 200M tax returns, by SSN

1. formula \( h() \) for hashing function
2. size of hash table \( M \)
3. collision resolution method

Hashing Functions

- Goal:
  uniform spread of keys over hash buckets

- Popular choices:
  - Division hashing
  - Multiplication hashing

Division Hashing

\[ h(x) = (a \times x + b) \mod M \]

- e.g., \( h(SSN) = (SSN) \mod 1,000 \)
- gives the last three digits of SSN
- \( M \): size of hash table
- choose a prime number, defensively (why?)

Division Hashing (cont.)

- e.g., \( M=2 \): hash on driver-license number (dln), where last digit is ‘gender’ (0/1 = M/F)
- In an army unit with predominantly male soldiers
- Thus: avoid cases where \( M \) and keys have common divisors - prime \( M \) guards against that!

Multiplication Hashing

\[ h(x) = \lfloor \text{fractional-part-of} \ ( x \times \phi) \rfloor \times M \]

- \( \phi \): golden ratio (0.618... = (sqrt(5)-1)/2)
- in general, we need an irrational number
- advantage: \( M \) need not be a prime number
- but \( \phi \) must be irrational
Other Hashing Functions

- quadratic hashing (bad)
- ...
- conclusion: use division hashing

Design Decisions

1) formula \( h() \) for hashing function
2) size of hash table \( M \)
3) collision resolution method

Size of Hash Table

- e.g., 50,000 employees, 10 employees/page
- Q: \( M = ? \) pages/buckets/slots
- A: utilization \( \sim 90\% \) and \( M \): prime number
e.g., in our case: \( M = \) closest prime to \( 50,000/10 / 0.9 = 5,555 \)

Collision Resolution

- Q: what is a ‘collision’?
- A: ??
Collision Resolution (cont.)

- Q: what is a ‘collision’?
  - A: ??
- Q: why worry about collisions/overflows? (recall that buckets are ~90% full)
  - A: ‘birthday paradox’

Collision Resolution (cont.)

- Open addressing
  - linear probing (i.e., put to next slot/bucket)
  - re-hashing
- Separate chaining
  - put links to overflow pages

Collision Resolution (cont.)

- linear probing
  - #0 page
  - #h(123)
  - M
  - 123; Smith; Main str.

Collision Resolution (cont.)

- re-hashing
  - #0 page
  - h1()
  - h2()
  - #h(123)
  - M
  - 123; Smith; Main str.

Collision Resolution (cont.)

- separate chaining
  - 123; Smith; Main str.

Design Decisions - Conclusions

- Hash function: division hashing
  - \( h(x) = (a \times x + b) \mod M \)
- Table size \( M \): ~90% utilization; prime number
- Collision resolution: separate chaining
  - easier to implement (deletions!)
  - no danger of becoming full
Hashing vs B-trees

Hashing offers:
- speed! (O(1) avg. search time)

...but B-trees give:
- key ordering:
  - range queries
  - proximity queries
  - sequential scan
- O(log(N)) guarantees for search, ins./del.
- graceful growing/shrinking

Indexing

- B-trees
- hashing
  - Hashing vs B-trees
- Indices in SQL
  - extensible hashing

Indexing in SQL

- create index <index-name> on <relation-name> (<attribute-list>)
- create unique index <index-name> on <relation-name> (<attribute-list>)
- drop index <index-name>

Indexing- overview

- B-trees
- hashing
- Indices in SQL
  - extensible hashing
    - 'extendible' hashing [Fagin, Pipenger +]
    - 'linear' hashing [Litwin]

Problem with static hashing

- problem: overflow?
- problem: underflow? (underutilization)
Dynamic/Extendible Hashing

- Idea: shrink / expand hash table on demand.
- Dynamic hashing
- Details: how to grow gracefully, on overflow?
- Many solutions
- One of them: 'extendible hashing' [Fagin et al]

Extendible Hashing (cont.)

In detail:
- Keep a directory, with pointers to hash-buckets

Q: how to divide contents of bucket in two?
A: hash each key into a very long bit string; keep only as many bits as needed

Eventually:
Extendible Hashing (cont.)

directory

00...
01...
10...
11...

new page / bucket

001...
011...
1011...
1101...

Directory doubles on demand
or halves, on shrinking files
needs ‘local’ and ‘global’ depth

BEFORE

AFTER

001...
011...
1011...
1101...

000...
001...
010...
011...
100...
101...
110...
111...

Indexing

- B-trees
- hashing
- Hashing vs B-trees
- Indices in SQL
- extendible hashing
  - ‘extensible’ hashing [Fagin, Pipenger +]
  - ‘linear’ hashing [Litwin]

Linear Hashing - Outline

- Motivation
- main idea
- search algorithm
- insertion/split algorithm
- deletion
- performance analysis
- variations
Linear Hashing

- Motivation: external hashing needs directory etc etc; which doubles (ouch!)
- Q: can we do something simpler, with smoother growth?
- A: split buckets from left to right, regardless of which one overflowed ('crazy', but it works well!)
  - E.g.:

Initially: \( h(x) = x \mod N \) (\( N=4 \) here)

Assume capacity: 3 records/bucket
Insert key '17'

<table>
<thead>
<tr>
<th>bucket-id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td></td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Split #0, anyway!!!

Q: But, how?

A: use two h.f.:

\( h0(x) = x \mod N \)

\( h1(x) = x \mod (2N) \)

Split #0, anyway!!!

<table>
<thead>
<tr>
<th>bucket-id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td></td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Linear Hashing (after split)

A: use two h.f.: $h_0(x) = x \mod N$

$h_1(x) = x \mod (2 \cdot N)$

<table>
<thead>
<tr>
<th>bucket- id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
<td>overflow</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Linear Hashing (after split)

A: use two h.f.: $h_0(x) = x \mod N$

$h_1(x) = x \mod (2 \cdot N)$

<table>
<thead>
<tr>
<th>bucket- id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
<td>overflow</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Linear Hashing: Outline

- Motivation
- main idea
- search algorithm
- insertion/split algorithm
- deletion
- performance analysis
- variations

### Linear Hashing: Searching

$h_0(x) = x \mod N$ (for the un-split buckets)

$h_1(x) = x \mod (2 \cdot N)$ (for the split ones)

<table>
<thead>
<tr>
<th>bucket- id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
<td>overflow</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Linear Hashing: Searching

Q1: find key ‘6’?

Q2: find key ‘4’?

Q3: key ‘8’?
Linear Hashing: Searching

Algorithm to find key ‘k’:
- compute $b = h_0(k)$;
  - if $b < \text{split ptr}$, compute $b = h_1(k)$
- search bucket $b$

Linear Hashing: Insertion

Algorithm: insert key ‘k’
- compute appropriate bucket ‘b’
- if the overflow criterion is true
  - split the bucket of ‘split ptr’
  - split ptr ++ (*)

Split pointer

Linear Hashing: Outline

- Motivation
- main idea
- search algo
- insertion/split algo
- deletion
- performance analysis
- variations

Linear Hashing: Insertion

- notice: overflow criterion is up to us!!
- Q: suggestions?
- A1: space utilization > u-max
- A2: avg length of overflow chains > max-len
- A3: ....
Linear Hashing: split now?

\[ h_0(x) = x \mod N \] (for the un-split buckets)
\[ h_1(x) = x \mod (2N) \] (for the split ones)

split pointer

\[ \begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\end{array} \]

Linear Hashing: split now?

\[ h_0(x) = x \mod N \] (for the un-split buckets)
\[ h_1(x) = x \mod (2N) \] (for the split ones)

split pointer

\[ \begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\end{array} \]

Linear Hashing: split now?

\[ h_0(x) = x \mod N \] (for the un-split buckets)
\[ h_1(x) = x \mod (2N) \] (for the split ones)

split pointer

\[ \begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\end{array} \]

Linear Hashing: split now?

\[ h_0(x) = x \mod N \] (for the un-split buckets)
\[ h_1(x) = x \mod (2N) \] (for the split ones)

split pointer

\[ \begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\end{array} \]

Linear Hashing: split now?

\[ h_0(x) = x \mod N \] (for the un-split buckets)
\[ h_1(x) = x \mod (2N) \] (for the split ones)

split pointer

\[ \begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\end{array} \]

Linear Hashing: Observations

In general, at any point of time, we have at most two h.f. active, of the form:

- \[ h_n(x) = x \mod (N \times 2^n) \]
- \[ h_{n+1}(x) = x \mod (N \times 2^{n+1}) \]

(after a full expansion, we have only one h.f.)

Linear Hashing: Outline

- Motivation
- Main idea
- Search algorithm
- Insertion/split algorithm
- Deletion
- Performance analysis
- Variations
Linear Hashing: Deletion

- reverse of insertion:
  - if the underflow criterion is met
  - contract!

Linear Hashing: how to contract?

- \( h_0(x) = \text{mod } N \) (for the un-split buckets)
- \( h_1(x) = \text{mod } (2N) \) (for the split ones)

Linear Hashing: Outline

- Motivation
- main idea
- search algorithm
- insertion/split algorithm
- deletion
- performance analysis
- variations

Linear Hashing: Performance

- [Larson, TODS 1982]
  - search-time (avg # of d.a.)
  - split: if \( u > u_0 \)
    (say \( u_0 = 0.85 \))
    - 1.01 d.a.
  - \# records
  - \( R \) 2R

Linear Hashing: Performance

- [Larson, TODS 1983]
  - search-time (avg # of d.a.)
  - split: if \( u > u_0 \)
    (say \( u_0 = 0.85 \))
  - 1.01 d.a.
Linear Hashing: Performance

- [Larson, TODS 1983]

Search-time (avg # of d.a.)

- split: if $u > u_0$
  - (say $u_0 = 0.85$)

# records

1.01 d.a. vs. 2R

Q: How to shorten the maximum?

- Q: How to shorten the maximum?
  - A: 2-3 splits - partial expansions!
Linear Hashing: Performance

- Q: How to shorten the maximum?
- A: 2-3 splits - partial expansions!

Linear Hashing: Outline

- Motivation
- main idea
- search algorithm
- insertion/split algorithm
- deletion
- performance analysis
- variations

Linear Hashing: Variations

Two split pointers! On split:

0 1 2 3

2nd split:

0 1 2 3

2nd split: Partial expansion! (50% larger table)

0 1 2 3 4 5
Linear Hashing: Variations

Q: how to do the third split?

0 1 2 3 4 5

Linear Hashing: Variations

Q: how to do the third split?
A: 3-to-4 splits now!

0 1 2 3 4 5

Linear Hashing: Performance

- Q1: Which of the two red peaks is higher?
- Q2: Why?

search-time

R

2R

# records

Other Hashing Variations

- ‘order preserving’
- ‘perfect hashing’ (no collisions!) [Ed. Fox, et al]

Primary key indexing: Conclusions

- hashing is O(1) on the average for search
- linear hashing: elegant way to grow a hash table
- B-trees: major contenders for primary-key indexing (O(log(N) w.c.))

References: primary key indexing

References (cont.)

- [Litwin] Litwin, W., (1980), Linear Hashing: A New Tool for File and Table Addressing, VLDB, Montreal, Canada, 1980