Outline

Goal: ‘Find similar / interesting things’

• Intro to DB

• Indexing - similarity search

• Data Mining

Indexing - Detailed outline

• 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

A: Brilliant idea: key-to-address transformation:

123; Smith; Main str

#0 page

#123 page

#999,999,999

Hashing

Since space is NOT free:

• use $M$, instead of 999,999,999 slots

• hash function: $h(key) = slot-id$
**Hashing**

Typically: each hash bucket is a page, holding many records:

Notice: could have clustering, or non-clustering versions:

- eg., IRS, 200M tax returns, by SSN

**Indexing- overview**

- B-trees
- hashing
  - hashing functions
  - size of hash table
  - collision resolution
- Hashing vs B-trees
- Indices in SQL

**Design decisions**

1) formula $h()$ for hashing function
2) size of hash table $M$
3) collision resolution method
Design decisions - functions

- **Goal:** uniform spread of keys over hash buckets
- **Popular choices:**
  - Division hashing
  - Multiplication hashing

Division hashing

- eg., $M=2$; 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

- $h(x) = \lfloor {\text{fractional-part-of} \,( x \cdot \varphi ) } \rfloor \cdot M$
  - $\varphi$: golden ratio (0.618... = (sqrt(5)-1)/2)
  - in general, we need an irrational number
  - advantage: $M$ need not be a prime number
  - but $\varphi$ 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

- eg., 50,000 employees, 10 employee-records / page
- Q: $M = ??$ pages/buckets/slots

Size of hash table

- eg., 50,000 employees, 10 employees/page
- Q: $M = ??$ pages/buckets/slots
- A: utilization ~ 90% and
  - $M$: prime number
  
  Eg., in our case: $M = $ closest prime to $50,000 / 10 / 0.9 = 5,555$

Design decisions

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

Collision resolution

- Q: what is a ‘collision’?
- A: ??

Collision resolution

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

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

Collision resolution

- linear probing:
  - $h(x) = (ax+b) \mod M$
  - easier to implement (deletions!)
  - no danger of becoming full

Collision resolution

- separate chaining:
  - $h(x) = (ax+b) \mod M$
  - easier to implement (deletions!)
  - no danger of becoming full

Design decisions - conclusions

- function: division hashing
  - $h(x) = (ax+b) \mod M$

- size $M$: ~90% util.; prime number.

- collision resolution: separate chaining
  - easier to implement (deletions!);
  - no danger of becoming full

Indexing - overview

- B-trees
- hashing
  - Hashing vs B-trees
- Indices in SQL
- extendible hashing
Hashing vs B-trees:

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

Hashing vs B-trees:

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

Hashing vs B-trees:

thus:
- B-trees are implemented in most systems

footnotes:
- hashing is rarely implemented (why not?)
- 'dbm' and 'ndbm' of UNIX: offer one or both

Indexing- overview

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

- eg.,
  create index ssn-index on STUDENT (ssn)
- or (eg., on Takes(ssn,cid,grade)):
  create index sc-index on Takes (ssn, c-id)
**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)

**Solution: 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**

solution:
split the bucket in two

123: Smith; Main str.

123; Smith; Main str.

in detail:
- keep a directory, with ptrs 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

- Summary: directory doubles on demand
- or halves, on shrinking files
- needs ‘local’ and ‘global’ depth

Indexing- overview

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

Linear hashing - overview

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

Linear hashing

Motivation: ext. hashing needs directory etc etc; which doubles (ouch!)
Q: can we do something simpler, with smoother growth?

Linear hashing

Initially: \( h(x) = x \mod N \) \((N=4 \text{ here})\)
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>6</td>
<td>7</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>
Linear hashing

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

\[
\begin{array}{cccc}
\text{bucket- id} & 0 & 1 & 2 & 3 \\
4 & 8 & 5 & 9 & 13 \\
6 & 7 & 11 & &
\end{array}
\]

overflow of bucket#1

\[
\begin{array}{cccc}
\text{bucket- id} & 0 & 1 & 2 & 3 \\
4 & 8 & 5 & 9 & 13 \\
6 & 7 & 11 & &
\end{array}
\]

Linear hashing

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

Q: But, how?

\[
\begin{array}{cccc}
\text{bucket- id} & 0 & 1 & 2 & 3 \\
4 & 8 & 5 & 9 & 13 \\
6 & 7 & 11 & &
\end{array}
\]

Split #0, anyway!!!

\[
\begin{array}{cccc}
\text{bucket- id} & 0 & 1 & 2 & 3 \\
4 & 8 & 5 & 9 & 13 \\
6 & 7 & 11 & &
\end{array}
\]

Linear hashing - after split:

A: use two h.f.: \( h0(x) = x \mod N \)
\( h1(x) = x \mod (2*N) \)

\[
\begin{array}{cccc}
\text{bucket- id} & 0 & 1 & 2 & 3 & 4 \\
8 & 5 & 9 & 13 & 6 \\
7 & 11 & 4 & &
\end{array}
\]

\[
\begin{array}{cccc}
\text{bucket- id} & 0 & 1 & 2 & 3 & 4 \\
8 & 5 & 9 & 13 & 6 \\
7 & 11 & 4 & &
\end{array}
\]

overflow
Linear hashing - overview

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

Linear hashing - after split:

A: use two h.f.: 
\[ h_0(x) = x \mod N \]
\[ h_1(x) = x \mod (2*N) \]

<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></td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>6</td>
</tr>
</tbody>
</table>

17 overflow

Linear hashing - searching?

\[ h_0(x) = x \mod N \] (for the un-split buckets)
\[ h_1(x) = x \mod (2*N) \] (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></td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>6</td>
</tr>
</tbody>
</table>

17 overflow

Linear hashing - searching?

Algo 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?

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

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

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

\[ h_1(x) = x \mod (2^N) \] (for the split buckets)

**Linear hashing - observations**

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

- \( h_0(x) = x \mod (N \cdot 2^n) \)
- \( h_{n+1}(x) = x \mod (N \cdot 2^{n+1}) \)

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

**Linear hashing - overview**

- Motivation
- Main idea
- Search algo
- Insertion/split algo
- Deletion
- Performance analysis
- Variations
Linear hashing - deletion?

- reverse of insertion:

Linear hashing - deletion?

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

Linear hashing - how to contract?

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

split ptr

0 1 2 3 4 5 6

Linear hashing - how to contract?

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

split ptr

0 1 2 3 4 5

Linear hashing - overview

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

Linear hashing - performance

- [Larson, TODS 1982]
  - [Larson, TODS 1982]

split: if \( u > u_0 \) (say \( u_0 = 0.85 \))

\[ \text{search-time (avg # of d.a.)} \]

1.01 d.a.

\[ \text{split: if } u > u_0 \]

\[ \text{(say } u_0 = 0.85) \]

R 2R

# records
Linear hashing - performance

- [Larson, TODS 1983]

split: if \( u > u_0 \)
(say \( u_0 \approx 0.85 \))

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

1.01 d.a.

??

# records
R
2R

Q: How to shorten the maximum?

search-time

eg., 1.3 d.a.

eg., 1.01 d.a.

# records
R
2R
Linear hashing - overview

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

Linear hashing - performance

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

Linear hashing - variations

Two split pointers! On split:

```
  ↓  ↓  ↓  ↓
0   1   2   3
```

Linear hashing - variations

2nd split:

```
  ↓  ↓
0   1   2   3
```
Linear hashing - variations

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

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

Linear hashing - variations

Q: how to do the third split?

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

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 for primary key indexing


References, cont’d

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