CS 5614: (Big) Data Management Systems

B. Aditya Prakash
Lecture #5: Hashing and Sorting
(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
Since space is NOT free:
- use M, instead of 999,999,999 slots
- hash function: \( h(key) = \text{slot-id} \)
Typically: each hash bucket is a page, holding many records:
Hashing

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

123; Smith; Main str.
Hashing

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

- $h(x) = (a*x+b) \mod M$
- eg., $h(ssn) = (ssn) \mod 1,000$
  - gives the last three digits of ssn
- $M$: size of hash table - choose a prime number, defensively (why?)
Division hashing

- 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

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

- \( \phi \): golden ratio \( (0.618\ldots = (\sqrt{5}-1)/2) \)
- 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)
- ...
Other hashing functions

- quadratic hashing (bad)
- ...
- conclusion: use division hashing
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 $\sim 90\%$ and
  - $M$: prime number

Eg., in our case: $M= \text{closest prime to } 50,000/10 / 0.9 = 5,555$
Collision resolution

Q: what is a ‘collision’?
A: ??

Prakash 2014
Collision resolution

123; Smith; Main str.

#h(123)

#0 page

M
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 (ie., put to next slot/bucket)
  - re-hashing
- separate chaining (ie., put links to overflow pages)
Collision resolution

linear probing:

123; Smith; Main str.

#0 page

#h(123)

M
re-hashing

123; Smith; Main str.

Collision resolution

h1()

h2()

#h(123)

M

#0 page
Collision resolution

separate chaining

123; Smith; Main str.
Design decisions - conclusions

- function: division hashing
  - $h(x) = (a \times x + b) \mod M$
- size $M$: ~90% util.; prime number.
- collision resolution: separate chaining
  - easier to implement (deletions!);
  - no danger of becoming full
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

123; Smith; Main str. → #h(123) → FULL

#0 page
Extendible hashing

solution:
split the bucket in two

123; Smith; Main str. -> #h(123)

#0 page

M
Extendible hashing

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

directory

00...
01...
10...
11...
101001...

0001...
0111...
10101...
10011...
1101...
10110...
1101...

Extendible hashing

directory

00...
01...
10...
11...
101001...

0001...
0111...

10101...
10011...
10110...

1101...

Extendible hashing

directory

split on 3-rd bit
Extendible hashing

directory

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

new page / bucket

1001...
10110...
1101...

Extendible hashing

Directory (doubled)

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

0001...
0111...
10011...
0111...
101001...
10101...
10110...

10011...

10101...
101001...
10110...

New page / bucket
Extendible hashing

BEFORE

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

101001...
10101...
10110...
1101...
1101...

AFTER

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

0001...
0111...
0001...
0111...
10011...
0111...
0111...
0001...

BEFORE

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

101001...
10101...
10110...
1101...
1101...

AFTER

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

0001...
0111...
0001...
0111...
10011...
0111...
0111...
0001...

Prakash 2014
Extendible hashing

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

- Motivation
- main idea
- search algo
- insertion/split algo
- deletion
Linear hashing

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

- Motivation: ext. 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!) - Eg.:
Initially: $h(x) = x \mod N$  \quad (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></td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>
Initially: $h(x) = x \mod N$  (N=4 here)

overflow of bucket#1

bucket- id

0 1 2 3

4 8 5 9 6 7 11

13
Linear hashing

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

bucket- id

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

overflow of bucket#1

Split #0, anyway!!!
Initially: \( h(x) = x \mod N \) (\( N=4 \) here)

Split #0, anyway!!!

Q: But, how?
Linear hashing

A: use two h.f.:  

\[ h_0(x) = x \mod N \]

\[ h_1(x) = x \mod (2*N) \]
Linear hashing - after split:

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

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

bucket- id

0  1  2  3  4

8  5  9  6  7  11  4

17
Linear hashing - after split:

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

$h_1(x) = x \mod (2*N)$
Linear hashing - searching?

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

bucket-id

<table>
<thead>
<tr>
<th></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>6</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

split ptr

overflow
Linear hashing - searching?

Q1: find key ‘6’?  Q2: find key ‘4’?  Q3: key ‘8’?

bucket-id

0  1  2  3  4

8  5  9  6  7  11  13  4

overflow

split ptr

Prakash 2014
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 $b$
Algo: insert key ‘k’

• compute appropriate bucket ‘b’

• if the **overflow criterion** is true
  
  • split the bucket of ‘split-ptr’

  • split-ptr ++ (*)
Linear hashing - insertion?

- notice: overflow criterion is up to us!!
- Q: suggestions?
Linear hashing - insertion?

- notice: overflow criterion is up to us!!
- Q: suggestions?
- A1: space utilization $\geq u\text{-max}$
Linear hashing - insertion?

- 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 - insertion?

Algo: insert key ‘k’

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

what if we reach the right edge??
Linear hashing - split now?

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

$h_0(x) = x \mod N$  \hspace{1cm} (for the un-split buckets) $h_1(x) = x \mod (2N)$  \hspace{1cm} (for the splitted ones)
Linear hashing - split now?

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

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

split ptr

0 1 2 3 4 5 6 7
Linear hashing - split now?

This state is called ‘full expansion’
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 - deletion?

- reverse of insertion:
Linear hashing - deletion?

- reverse of insertion:
- if the underflow criterion is met
  – contract!
Linear hashing - how to contract?

\[ h_0(x) = \text{mod } N \quad \text{(for the un-split buckets)} \]
\[ h_1(x) = \text{mod } (2*N) \quad \text{(for the split ones)} \]
Linear hashing - how to contract?

\[ h_0(x) = \text{mod } N \quad \text{(for the un-split buckets)} \]
\[ h_1(x) = \text{mod } (2N) \quad \text{(for the splitten ones)} \]
Hashing - pros?
Hashing - pros?

- Speed,
  - on exact match queries
  - on the average
B(+) - trees - pros?
B(+-) - trees - pros?

- Speed on search:
  - exact match queries, worst case
  - range queries
  - nearest-neighbor queries

- Speed on insertion + deletion

- smooth growing and shrinking (no re-org)

Prakash 2014
Conclusions

- B-trees and variants: in all DBMSs
- hash indices: in some
  - (but hashing in useful for joins...)

Prakash 2014

VT CS 5614
SORTING
Why Sort?
Why Sort?

- `select ... order by`
  - e.g., find students in increasing `gpa` order
- `bulk loading` B+ tree index.
- `duplicate elimination` (select distinct)
- `select ... group by`
- `Sort-merge` join algorithm involves sorting.
Outline

- two-way merge sort
- external merge sort
- fine-tunings
- B+ trees for sorting
2-Way Sort: Requires 3 Buffers

- **Pass 0**: Read a page, sort it, write it.
  - only one buffer page is used
- **Pass 1, 2, 3, …, etc.**: requires 3 buffer pages
  - merge pairs of runs into runs twice as long
  - three buffer pages used.

![Diagram of 2-Way Sort](image)
Two-Way External Merge Sort

- Each pass we read + write each page in file.
Two-Way External Merge Sort

- Each pass we read + write each page in file.
Two-Way External Merge Sort

- Each pass we read + write each page in file.
- Each pass we read + write each page in file.
Two-Way External Merge Sort

- Each pass we read + write each page in file.
- N pages in the file =>
  \[ \lceil \log_2 N \rceil + 1 \]
- So total cost is:
  \[ 2N \left( \lceil \log_2 N \rceil + 1 \right) \]
- Idea: Divide and conquer: sort subfiles and merge
External merge sort

B > 3 buffers

- Q1: how to sort?
- Q2: cost?
General External Merge Sort

$B > 3$ buffer pages. How to sort a file with $N$ pages?
General External Merge Sort

– Pass 0: use $B$ buffer pages. Produce $\lceil N / B \rceil$ sorted runs of $B$ pages each.
– Pass 1, 2, ..., etc.: merge $B-1$ runs.
Sorting

– create sorted runs of size B (how many?)
– merge them (how?)

B

...
Sorting

- create sorted runs of size $B$
- merge first $B-1$ runs into a sorted run of $(B-1) \cdot B$, ...

```
   B
    ...   ...   ...
    ...   ...   ...
    ...   ...   ...
```
Sorting

- How many steps we need to do?
  ‘i’, where \( B^i (B-1)^i > N \)
- How many reads/writes per step? \( N+N \)
Cost of External Merge Sort

- Number of passes: \(1 + \left\lceil \log_{B^{-1}} \left\lceil \frac{N}{B} \right\rceil \right\rceil\)
- Cost = \(2N \times \text{(\# of passes)}\)
Cost of External Merge Sort

- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: \( \left\lfloor \frac{108}{5} \right\rfloor = 22 \) sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: \( \left\lfloor \frac{22}{4} \right\rfloor = 6 \) sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

Formula check: \( \lceil \log_4 22 \rceil = 3 \ldots + 1 \rightarrow \text{4 passes} \) \( \checkmark \)
### Number of Passes of External Sort

(I/O cost is $2N \times$ number of passes)

<table>
<thead>
<tr>
<th>N</th>
<th>$B=3$</th>
<th>$B=5$</th>
<th>$B=9$</th>
<th>$B=17$</th>
<th>$B=129$</th>
<th>$B=257$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10,000</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10,000,000</td>
<td>23</td>
<td>12</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>100,000,000</td>
<td>26</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>30</td>
<td>15</td>
<td>10</td>
<td>8</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>
Internal Sort Algorithm

- Quicksort is a fast way to sort in memory.
Blocked I/O & double-buffering

- So far, we assumed random disk access
- Cost changes, if we consider that runs are written (and read) sequentially
- What could we do to exploit it?
Blocked I/O & double-buffering

- So far, we assumed random disk access
- Cost changes, if we consider that runs are written (and read) sequentially
- What could we do to exploit it?
- A1: Blocked I/O (exchange a few r.d.a for several sequential ones)
- A2: double-buffering
Double Buffering

- To reduce wait time for I/O request to complete, can *prefetch* into `shadow block`.
  - Potentially, more passes; in practice, most files still sorted in 2-3 passes.
Using B+ Trees for Sorting

- **Scenario**: Table to be sorted has B+ tree index on sorting column(s).

- **Idea**: Can retrieve records in order by traversing leaf pages.

- *Is this a good idea?*

- **Cases to consider:**
  - B+ tree is clustered
  - B+ tree is *not* clustered
Using B+ Trees for Sorting

- Scenario: Table to be sorted has B+ tree index on sorting column(s).
- Idea: Can retrieve records in order by traversing leaf pages.
- Is this a good idea?
- Cases to consider:
  - B+ tree is clustered Good idea!
  - B+ tree is not clustered Could be a very bad idea!
Clustered B+ Tree Used for Sorting

- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)

Always better than external sorting!
Alternative (2) for data entries; each data entry contains \textit{rid} of a data record. In general, \textit{one I/O per data record}!
## External Sorting vs. Unclustered Index

<table>
<thead>
<tr>
<th>N</th>
<th>Sorting</th>
<th>p=1</th>
<th>p=10</th>
<th>p=100</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>200</td>
<td>100</td>
<td>1,000</td>
<td>10,000</td>
</tr>
<tr>
<td>1,000</td>
<td>2,000</td>
<td>1,000</td>
<td>10,000</td>
<td>100,000</td>
</tr>
<tr>
<td>10,000</td>
<td>40,000</td>
<td>10,000</td>
<td>100,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>100,000</td>
<td>600,000</td>
<td>100,000</td>
<td>1,000,000</td>
<td>10,000,000</td>
</tr>
<tr>
<td>1,000,000</td>
<td>8,000,000</td>
<td>1,000,000</td>
<td>10,000,000</td>
<td>100,000,000</td>
</tr>
<tr>
<td>10,000,000</td>
<td>80,000,000</td>
<td>10,000,000</td>
<td>100,000,000</td>
<td>1,000,000,000</td>
</tr>
</tbody>
</table>

\( p \): \# of records per page  
\( B=1,000 \) and block size=32 for sorting  
\( p=100 \) is the more realistic value.
Summary

- External sorting is important
- External merge sort minimizes disk I/O cost:
  - Pass 0: Produces sorted runs of size $B$ (# buffer pages).
  - Later passes: merge runs.
- Clustered B+ tree is good for sorting; unclustered tree is usually very bad.