Database Applications (15-415)

DBMS Internals- Part V
Lecture 15, March 15, 2015

Mohammad Hammoud
Today...

- Last Session:
  - DBMS Internals- Part IV
    - Tree-based (i.e., B+ Tree) and Hash-based (i.e., Extendible Hashing) indexes

- Today’s Session:
  - DBMS Internals- Part V
    - Hash-based indexes (Cont’d) and External Sorting

- Announcements:
  - Project 2 is due today by midnight. Student demos will be conducted on Tuesday/Thursday
  - PS3 is now posted and it is due on March 26 by midnight
  - Project 3 will be posted by Thursday
DBMS Layers

Queries

Query Optimization and Execution

Relational Operators

Files and Access Methods

Buffer Management

Disk Space Management

Transaction Manager

Lock Manager

Recovery Manager

DB
Outline

- Linear Hashing
- Why Sorting?
- In-Memory vs. External Sorting
- A Simple 2-Way External Merge Sorting
- General External Merge Sorting
- Optimizations: Replacement Sorting, Blocked I/O and Double Buffering
Linear Hashing

- Another way of adapting gracefully to insertions and deletions (i.e., pursuing dynamic hashing) is to use Linear Hashing (LH)

- In contrast to Extendible Hashing, LH
  - Does not require a directory
  - Deals naturally with collisions
  - Offers a lot of flexibility w.r.t the timing of bucket split (allowing trading off greater overflow chains for higher average space utilization)
How Linear Hashing Works?

- LH uses a family of hash functions $h_0, h_1, h_2, \ldots$
  - $h_i(key) = h(key) \mod(2^iN)$; $N =$ initial # buckets
  - $h$ is some hash function (range is not 0 to $N-1$)

- If $N = 2^{d0}$, for some $d0$, $h_i$ consists of applying $h$ and looking at the last $di$ bits, where $di = d0 + i$

- $h_{i+1}$ doubles the range of $h_i$ (similar to directory doubling)
How Linear Hashing Works? (Cont’d)

- LH uses overflow pages, and chooses buckets to split in a *round-robin* fashion

- Splitting proceeds in “rounds”
  - A round ends when all $N_R$ (for round $R$) initial buckets are split
  - Buckets 0 to Next-1 have been split; Next to $N_R$ yet to be split
  - Current round number is referred to as $Level$
Linear Hashing: Searching For Entries

- To find a bucket for data entry $r$, find $h_{\text{Level}}(r)$:
  - If $h_{\text{Level}}(r)$ in range `Next to $N_R$`, $r$ belongs there
  - Else, $r$ could belong to bucket $h_{\text{Level}}(r)$ or bucket $h_{\text{Level}}(r) + N_R$; must apply $h_{\text{Level}+1}(r)$ to find out

- Example: search for 5*

```
Level = 0  h0
5* = 101  01
```

```
<table>
<thead>
<tr>
<th></th>
<th>h</th>
<th>h</th>
<th>PRIMARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>PAGES</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>32<em>44</em>36*</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01</td>
<td>9<em>25</em>5*</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
<td>14<em>18</em>10<em>30</em></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>11</td>
<td>31<em>35</em>7<em>11</em></td>
</tr>
</tbody>
</table>
```
Linear Hashing: Inserting Entries

- Find bucket as in search
  - If the bucket to insert the data entry into is full:
    - Add an overflow page and insert data entry
    - *(Maybe)* Split Next bucket and increment Next

- Some points to Keep in mind:
  - Unlike Extendible Hashing, when an insert triggers a split, the bucket into which the data entry is inserted is not necessarily the bucket that is split
  - As in Static Hashing, an overflow page is added to store the newly inserted data entry
  - However, since the bucket to split is chosen in a round-robin fashion, eventually *all* buckets will be split
Linear Hashing: Inserting Entries

- Example: insert 43*

Level = 0 $\Rightarrow$ h0
43* = 101011 $\Rightarrow$ 11

Add an overflow page and insert data entry
Linear Hashing: Inserting Entries

- Example: insert 43*

  Level = 0 → h0
  43* = 101011 → 11

  Split Next bucket and increment Next
Linear Hashing: Inserting Entries

- Example: insert 43*

  Level = 0  ➔  h0
  43* = 101011  ➔  11

  Almost there...
Linear Hashing: Inserting Entries

- Example: insert 43*

Level = 0 $\rightarrow$ $h_0$
$43^* = 101011 \rightarrow 11$

**FINAL STATE!**
Another Example: insert 50*

Level = 0 \(\Rightarrow\) h0
50* = 110010 \(\Rightarrow\) 10

Add an overflow page and insert data entry
Linear Hashing: Inserting Entries

Another Example: insert 50*

Level = 0 ➔ h0
50* = 110010 ➔ 10

Split Next bucket and increment Next
Linear Hashing: Inserting Entries

- Another Example: insert 50*

Level = 0 ➞ h0
50* = 110010 ➞ 10

Almost there...
Linear Hashing: Inserting Entries

- Another Example: insert 50*

Level = 0 \(\Rightarrow\) \(h_0\)
50* = 110010 \(\Rightarrow\) 10

Almost there...
Linear Hashing: Inserting Entries

- Another Example: insert 50*

Level = 0  $\rightarrow$ h0
50* = 110010  $\rightarrow$ 10

Final State!
Linear Hashing: Deleting Entries

- Deletion is essentially the inverse of insertion

- If the last bucket in the file is empty, it can be removed and Next can be decremented

- If Next is zero and the last bucket becomes empty
  - Next is made to point to bucket $M/2 - 1$ (where $M$ is the current number of buckets)
  - Level is decremented
  - The empty bucket is removed

- The insertion examples can be worked out backwards as examples of deletions!
But, before we will discuss “Sorting”
Outline

- Linear Hashing
- Why Sorting?
- In-Memory vs. External Sorting
- A Simple 2-Way External Merge Sorting
- General External Merge Sorting
- Optimizations: Replacement Sorting, Blocked I/O and Double Buffering
When Does A DBMS Sort Data?

- Users may want answers in some order
  - `SELECT FROM student ORDER BY name`
  - `SELECT S.rating, MIN(S.age) FROM Sailors S GROUP BY S.rating`

- *Bulk loading* a B+ tree index involves sorting

- Sorting is useful in eliminating duplicates records

- The *Sort-Merge* Join algorithm involves sorting *(next session!)*
Outline

- Linear Hashing
- Why Sorting?
- In-Memory vs. External Sorting
- A Simple 2-Way External Merge Sorting
- General External Merge Sorting
- Optimizations: Replacement Sorting, Blocked I/O and Double Buffering
In-Memory vs. External Sorting

- Assume we want to sort 60GB of data on a machine with only 8GB of RAM
  - In-Memory Sort (e.g., Quicksort)?
    - Yes, but data do not fit in memory
    - What about relying on virtual memory?

- In this case, external sorting is needed
  - In-memory sorting is orthogonal to external sorting!
A Simple Two-Way Merge Sort

- **IDEA**: Sort sub-files that can fit in memory and merge

- Let us refer to each sorted sub-file as a *run*

- **Algorithm:**
  - **Pass 1**: Read a page into memory, sort it, write it
    - 1-page runs are produced
  - **Passes 2, 3, etc.**: Merge *pairs* (hence, 2-way) of runs to produce longer runs until only one run is left
A Simple Two-Way Merge Sort

- **Algorithm:**
  - **Pass 1:** Read a page into memory, sort it, write it
    - How many buffer pages are needed? **ONE**
  - **Passes 2, 3, etc.,**: Merge *pairs* (hence, 2-way) of runs to produce longer runs until only one run is left
    - How many buffer pages are needed? **THREE**
2-Way Merge Sort: An Example

PASS 0

PASS 1

PASS 2

PASS 3

Input File

1-Page Runs

2-Page Runs

4-Page Runs

8-Page Runs
2-Way Merge Sort: I/O Cost Analysis

- If the number of pages in the input file is $2^k$
  - How many runs are produced in pass 0 and of what size?
    - $2^k$ 1-page runs
  - How many runs are produced in pass 1 and of what size?
    - $2^{k-1}$ 2-page runs
  - How many runs are produced in pass 2 and of what size?
    - $2^{k-2}$ 4-page runs
  - How many runs are produced in pass $k$ and of what size?
    - $2^{k-k}$ $2^k$-page runs (or 1 run of size $2^k$)
  - For $N$ number of pages, how many passes are incurred?
    - $\lceil \log_2 N \rceil + 1$
  - How many pages do we read and write in each pass?
    - $2N$
  - What is the overall cost?
    - $2N \times (\lceil \log_2 N \rceil + 1)$
2-Way Merge Sort: An Example

\[ \lceil \log_2 8 \rceil + 1 \]
= 4 passes

Formula Check:

\[ 2N \times \left( \lceil \log_2 N \rceil + 1 \right) \]
= (2 \times 8) \times (3 + 1) = 64 I/Os
Correct!
Outline

1. Linear Hashing
2. Why Sorting?
3. In-Memory vs. External Sorting
4. A Simple 2-Way External Merge Sorting
5. General External Merge Sorting
6. Optimizations: Replacement Sorting, Blocked I/O and Double Buffering
How can we sort a file with $N$ pages using $B$ buffer pages?

- **Pass 0**: use $B$ buffer pages
  - This will produce $\left\lceil \frac{N}{B} \right\rceil$ sorted B-page runs
- **Pass 2, ..., etc.**: merge $B-1$ runs
B-Way Merge Sort: I/O Cost Analysis

- I/O cost = \(2N \times \text{Number of passes}\)

- Number of passes = \(1 + \left\lceil \log_{B-1} \left(\frac{N}{B}\right) \right\rceil\)

- Assume the previous example (i.e., 8 pages), but using 5 buffer pages (instead of 2)
  - I/O cost = 32 (as opposed to 64)

- Therefore, increasing the number of buffer pages minimizes the number of passes and accordingly the I/O cost!
## Number of Passes of B-Way Sort

<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>

High Fan-in during merging is crucial!

How else can we minimize I/O cost?
Outline

- Linear Hashing
- Why Sorting?
- In-Memory vs. External Sorting
- A Simple 2-Way External Merge Sorting
- General External Merge Sorting
- Optimizations: Replacement Sorting, Blocked I/O and Double Buffering
Replacement Sort

- With a more aggressive implementation of B-way sort, we can write out runs of \(~2 \times B\) internally sorted pages
  - This is referred to as replacement sort

**IDEA:** Pick the tuple in the current set with the smallest value that is greater than the largest value in the output buffer and append it to the output buffer
Replacement Sort

- With a more aggressive implementation of B-way sort, we can write out runs of $\sim 2 \times B$ internally sorted pages
  - This is referred to as replacement sort

When do we terminate the current run and start a new one?
Blocked I/O and Double Buffering

- So far, we assumed random disk accesses

- Would cost change if we assume that reads and writes are done sequentially?
  - Yes

- How can we incorporate this fact into our cost model?
  - Use bigger units (this is referred to as Blocked I/O)
  - Mask I/O delays through pre-fetching (this is referred to as double buffering)
Blocked I/O

- Normally, we go with ‘B’ buffers of size (say) 1 page
Blocked I/O

- Normally, we go with ‘B’ buffers of size (say) 1 page
- INSTEAD: let us go with $B/b$ buffers, of size ‘$b$’ pages
Blocked I/O

- Normally, we go with ‘$B$’ buffers of size (say) 1 page
- INSTEAD: let us go with $B/b$ buffers, of size ‘$b$’ pages

- What is the main advantage?
  - Fewer random accesses (as some of the page will be arranged sequentially!)

- What is the main disadvantage?
  - Smaller fan-in and accordingly larger number of passes!
Double Buffering

- Normally, when, say ‘INPUT1’ is exhausted
  - We issue a ‘read’ request and
  - We wait ...

![Diagram of double buffering with multiple input and output buffers.]
Double Buffering

- INSTEAD: *pre-fetch* INPUT1’ into a `shadow block`
  - When INPUT1 is exhausted, issue a ‘read’
  - BUT, also proceed with INPUT1’
  - Thus, the CPU can never go idle!
Next Class

Queries

Query Optimization and Execution

Relational Operators

Files and Access Methods

Buffer Management

Disk Space Management

DB

Transaction Manager

Lock Manager

Recovery Manager

Continue…