Today...

Welcome Back from Spring Break!
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 1 grades are out
  - Midterm grades are out
  - Project 2 is due on March 13 by midnight.
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
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

Using B+ Trees for External Sorting
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)
  - $h_{i+1}$ doubles the range of $h_i$ (this is similar to directory doubling)

- If $N = 2^{d_0}$, for some $d_0$, $h_i$ consists of applying $h$ and looking at the last $d_i$ bits, where $d_i = d_0 + i$
How Linear Hashing Works? (Cont’d)

- LH uses overflow pages, and chooses buckets to split in a \textit{round-robin} fashion

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

- To find bucket for data entry $r$, find $h_{\text{Level}}(r)$:
  - If $h_{\text{Level}}(r)$ in range `$\text{Next to } N_R$`, $r$ 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 $\Rightarrow$ h0
  $5^* = 101 \Rightarrow 01$
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 ➔ h0
43* = 101011 ➔ 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 \(\Rightarrow\) h0
  43* = 101011 \(\Rightarrow\) 11

Almost there...
Linear Hashing: Inserting Entries

- Example: insert 43*

Level = 0 ➞ h0
43* = 101011 ➞ 11

FINAL STATE!
Linear Hashing: Inserting Entries

- Another Example: insert 50*

  Level = 0 → h0
  50* = 110010 → 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 \(\rightarrow 10\)

Level = 1

- PRIMARY PAGES

Next = 0

- OVERFLOW PAGES

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
- Using B+ Trees for External Sorting
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
- Using B+ Trees for External Sorting
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!
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
- Using B+ Trees for 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

Input File

PASS 0

PASS 1

PASS 2

PASS 3

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

Input File

1-Page Runs

2-Page Runs

4-Page Runs

8-Page Runs

PASS 0

PASS 1

PASS 2

PASS 3

Formula Check:

\[ 2N \times \left( \lceil \log_2 N \rceil + 1 \right) \]

\[ = (2 \times 8) \times (3 + 1) = 64 \text{ I/Os} \]

Correct!
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
- Using B+ Trees for External Sorting
B-Way Merge Sort

- 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 × Number of passes

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

- 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

Using B+ Trees for External Sorting
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

**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 access

- 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

Diagram:

```
Disk  ---| 3 Main memory buffers ---|  Disk
      |                     |
      v                     v
INPUT 1  ---|  OUTPUT  ---|  INPUT 2

```

```
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-out 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]

Disk → INPUT 1 → OUTPUT → Disk
Disk → INPUT 2 → OUTPUT → Disk
Disk → INPUT B-1 → OUTPUT → Disk

B Main memory 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!

**Diagram:**
- **Disk** → **INPUT 1** → **INPUT 1’** → **INPUT 2** → **INPUT 2’** → ... → **INPUT k** → **INPUT k’** → **OUTPUT** → **OUTPUT’** → **Disk**
- **B main memory buffers, k-way merge**
- *b* block size
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
- Using B+ Trees for External Sorting
Using B+ Trees for External Sorting

- **Scenario**: the relation to be sorted has a B+ tree index on its primary key

- **IDEA**: retrieve records in order by traversing leaf pages

- **Is this a good idea?**
  - What if the B+ tree is clustered?
  - What if the B+ tree is un-clustered?
  - What about different indexing alternatives?
Using Clustered B+ Trees for Sorting

- What if Alternative (1) is in use?
  - **Cost:** root to the left-most leaf, then retrieve all leaf pages

- What if Alternative (2) or (3) is in use?
  - **Cost:** root to the left-most leaf, then fetch each page just once
Using Un-clustered B+ Trees for Sorting

What if Alternative (1) is in use?
- **Cost:** root to the left-most leaf, then retrieve all leaf pages

What if Alternative (2) or (3) is in use?
- **Cost:** root to the left-most leaf, then fetch pages
  - Worst-case: 1 I/O per each data record!
Using B+ Trees for External Sorting

- **Scenario**: the relation to be sorted has a B+ tree index on its primary key

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

- **Is this a good idea?**
  - What if the B+ tree is clustered?
    - Good idea!
  - What if the B+ tree is un-clustered?
    - Could be a very bad idea!
Summary

- External sorting is important; a DBMS may dedicate part of its buffer pool for sorting!

- External merge sort minimizes disk I/O cost:
  - Pass 0: Produces sorted runs of size $B$ (# buffer pages).
  - Later passes: merge runs
  - # of runs merged at a time depends on $B$, and block size
    - Larger block size means less I/O cost per page
    - Larger block size means smaller # runs merged
  - In practice, # of runs is rarely more than 2 or 3

- Clustered B+ tree is good for sorting; un-clustered tree is usually very bad!
Next Class

Queries

Query Optimization and Execution

Relational Operators

Files and Access Methods

Buffer Management

Disk Space Management

Transaction Manager

Lock Manager

Recovery Manager

DB