Classical External Sort III

Dmitri Loguinov
Texas A&M University

September 15, 2015
**Agenda**

- External merge sort
  - Replacement selection
- Distribution sort
  - Heterogeneous fan-out
- Wrap-up
External Merge Sort

- Replacement selection (RS)
  - Suppose RAM can hold $R$ key-value pairs
  - Previous methods fill this up, sort the result, then evict the entire buffer to disk as one run
  - But we can do better by evicting incrementally – new arrival pushes one existing (key, value) pair to disk

- Since the runs must be sorted, eviction applies to the smallest key in RAM that is larger than the last evicted
  - Implemented as two heaps $H_1, H_2$

- If new key is larger than $H_1$.root(), it gets added to $H_1$
  - Otherwise, goes into $H_2$

- Eviction applies to $H_1$.root() in all cases
  - What is the average run length if all keys are unique?
External Merge Sort 10

• Analysis shows asymptotically 2R (double RAM size)
  – Formulas are more complicated for the first few runs
  – Run 1 length = (e-1)R \approx 1.72R
  – Run 2 length = e(e-2)R \approx 1.95R
  – Run 3 length \approx 1.996R
  – and so on, oscillating around 2 thereafter

• How to implement RS with one heap?
• RS doesn’t eliminate duplicates in RAM
  – Can this be improved upon?
• IHT (Incremental Hash Table)
  – Maintains a hand h that points to the next bin to be evicted
  – If the incoming key is not in the hash table and more space is needed, entire chain pointed to by h is evicted
**Distribution Sort**

- Our homework has 1.8B keys, only 86M are unique
  - That’s a 95% duplicate rate
  - Among the considered methods thus far, IHT is likely to produce the longest runs, highest compaction, smallest number of runs \( r \), and least I/O

- Distribution (bucket) sort
  - Use the next \( k \) bits (left to right) to split the input to \( m=2^k \) files
  - Caveat: before the split, run the keys through a cache

- Recursively split buckets in DFS order
  - Until the leaves fit in RAM
  - Load, sort, and append to output
• Bucket sort is hard to analyze
  - The available RAM for the cache is \((P – mM)\), where \(P\) is the physical RAM and \(M\) is each I/O buffer
  - Then, \(R = (P – mM) / (E+K+V)\)

• **Tradeoff**: more buckets means less-effective caching

• Additionally, \(m\) can be heterogeneous across levels
  - There exists an optimal vector \((m_1, m_2, \ldots)\) that achieves the lowest I/O

• **Example**: input size \(n = 128M\), \(m_1 = 4\), \(m_2 = 8\)
  - If all keys are unique, first-level buckets are 32M keys, second-level are 4M; no differences from \(m_1 = 8\), \(m_2 = 4\)
  - But what if there are duplicate keys? Does \(m_1 = 8\), \(m_2 = 4\) perform better?
Distribution Sort 3

- **Rule of thumb**: a heterogeneous sequence \((m_1, m_2, \ldots)\) performs best if the largest fan-out goes first
  - Couples the largest cache with the smallest buckets

- **Example**: 60 GB IRLbot host graph
  - 6.8B edges, 640M unique (9%), \(M = 64\) MB, \(P = 1\) GB RAM
  - Consider two scenarios (2,8) and (8,2) splits; \(K+V=12\) bytes

```
(2,8) (8,2) (9)
(640M, 6.8B) (640M, 6.8B) (640M, 6.8B)

c=0.8 → cache 74M c=0.9 → cache 43M c=0.95 → cache 37M

B1: (320M, 2.7B) B1: (80M, 765M) B1: (71M, 717M)

cache 43M ↔ c=0.7 cache 74M ↔ c=0.15 cache 74M ↔ c=0.1

B11: (40M, 236M) B11: (40M, 57M) append to output
```
Distribution Sort 4

- Example (continued)
  - Actual implementation, variation of LRU cache
  - RAM size = 817 MB (1% of space for all key-value pairs)
  - The last bucket must be $\leq 68.2M$ keys

<table>
<thead>
<tr>
<th>$m_1$</th>
<th>$m_2$</th>
<th>level 1 bucket</th>
<th>level 2 bucket</th>
<th>total I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>880M</td>
<td>163M</td>
<td>8220M</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>688M</td>
<td>81M</td>
<td>8080M</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>419M</td>
<td>19M</td>
<td>9140M</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>420M</td>
<td>44M</td>
<td>8134M</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>1270M</td>
<td>159M</td>
<td>10164M</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1270M</td>
<td>252M</td>
<td>9118M</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>689M</td>
<td>140M</td>
<td>7742M</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>576M</td>
<td>67.8M</td>
<td>7790M</td>
</tr>
</tbody>
</table>
Distribution Sort 5

• Highlighted in gray are configurations that finish in two levels, of which (5,3) is optimal
  - Instead of upper-bits, we can also divide the interval \([0, 2^{64})\) into \(m\) equal-size blocks
  - Set \(D = 2^{64} / m\) and \(\text{bucket} = \text{floor} (\text{key} / D)\)
  - Requires integer division instead of bit shifts

• Best result for other methods
  - Hash tables use \(E = 9.2\) bytes of overhead per key
  - To be efficient, LRU requires \(E > 80\) bytes, in which case bucket sort is not competitive
  - CLOCK is an approximate LRU cache with \(E = 27\), but this also doesn’t beat IHT

<table>
<thead>
<tr>
<th>Method</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge sort</td>
<td>11052</td>
</tr>
<tr>
<td>hash table</td>
<td>10976</td>
</tr>
<tr>
<td>IHT</td>
<td>8508</td>
</tr>
</tbody>
</table>
• We use a custom LRU implementation with E=1 byte, which is when it actually beats the other methods
  – Hash tables don’t admit reduction in E down to this level
• In the best case, bucket sort is 2.6x more I/O efficient than merge sort on IRLbot graphs
• STL data structures are notoriously bloated
  – Keep an eye on your E (debug mode adds 50 bytes per item)
  – Task Manager->View->Select Columns->Peak Working Set
• What is a working set?
  – Set of pages in RAM your program has written to
  – Commit size is the amount of RAM allocated by the OS to your process, but not all of it may be used