External Memory Multimaps

Elaine Angelino (Harvard)
Michael Goodrich (UC Irvine)
Michael Mitzenmacher (Harvard)
Justin Thaler (Harvard)
Multimaps

• Store key-value pairs, but each key can have many values.
  – And distribution can be very skewed.
  – So you need something better than a standard hash table.
Operations

• **Container of key-value pairs C.**
  • insert(k,v) : insert key-value pair (k,v)
  • remove(k,v) : remove key-value pair (k,v)
  • isMember(k,v) : is (k,v) in the collection?
  • findAll(k) : return the set of all key-value pairs in C having key k
  • removeAll(k) : remove from C all key-value pairs having key k
  • count(k) : return count of all key-value pairs in C having key k
Motivations

• Now a standard abstraction.
  – Multimaps appear in C++ Standard Template Library, Google Java Collections Library, Apache Common Collections API.
  – Somebody thinks they’re useful!
• Extend inverted files/inverted indices.
  – Used in Web search
• Natural representation for graphical data.
• Geometric hashing, etc.
Possible Approaches

• C++ Standard Template Library uses red-black trees.
  – $O(\log n)$ worst-case operations.

• What about hashing?
  – Can we support all operations in worst-case constant time with high probability?
Our Work:
External Memory Multimaps

• For big data sets, number of memory accesses is paramount.
  – Each memory block has size $B$ (may be $\omega(1)$).
  – Minimize the number of memory blocks that must be touched – I/O operations – especially for findAll.

• Seek external memory schemes with theoretical performance guarantees, viable in practice.
## Results

<table>
<thead>
<tr>
<th>Method</th>
<th>I/O Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(k,v)</td>
<td>expected O(1)</td>
</tr>
<tr>
<td>remove(k,v)</td>
<td>O(1)</td>
</tr>
<tr>
<td>isMember(k,v)</td>
<td>O(1)</td>
</tr>
<tr>
<td>findAll(k)</td>
<td>O(1+count(k)/B)</td>
</tr>
<tr>
<td>removeAll(k)</td>
<td>O(1)</td>
</tr>
<tr>
<td>count(k)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

Results are *unamortized*, with linear space.
Methodology

• Utilize two data structures.
• External-memory cuckoo hashing (fast lookups, removals, inserts)
  – Make use of large buckets.
  – Keep expected insertion time small.
  – Lookups and removals are constant time in the worst-case.
• External-memory multiquotes (fast findAlls and removeAlls)
  – Keep values associated with a key in a queue.
  – Need to keep entire queue in contiguous memory, while using O(n) space.
Primary structure is a cuckoo hash table, points to multiqueues. Key-value dictionary is a cuckoo hash table, points to (k,v) pairs.
Our Solution, Step-by-Step

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>isMember(k,v)</td>
</tr>
<tr>
<td>remove(k,v)</td>
</tr>
<tr>
<td>insert(k,v)</td>
</tr>
<tr>
<td>findAll(k)</td>
</tr>
<tr>
<td>removeAll(k)</td>
</tr>
<tr>
<td>count(k)</td>
</tr>
</tbody>
</table>
Our Solution, Step-by-Step

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isMember(k,v)</td>
<td>Keep (k, v) pairs in a cuckoo hash table.</td>
</tr>
<tr>
<td>remove(k,v)</td>
<td>Supports lookups and removals in worst case O(1) time.</td>
</tr>
<tr>
<td>insert(k,v)</td>
<td>Is it suitable for external memory?</td>
</tr>
</tbody>
</table>

• Keep (k, v) pairs in a cuckoo hash table.
  • Supports lookups and removals in worst case O(1) time.
  • Is it suitable for external memory?
External-Memory Cuckoo Hashing

• Would like each block to correspond to a bucket, but don’t know how to analyze random-walk cuckoo hashing with large buckets.

• For theory guarantees, turn to BFS-based cuckoo hashing [Dietzfelbinger and Weidling 2007]. But then inserts are only fast for constant-sized buckets.
  • Can simulate constant-sized buckets in external memory even when block size is super-constant.

• Experimentally, random walk cuckoo hashing works great!
Our Solution, Step-by-Step

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>findAll(k)</td>
</tr>
</tbody>
</table>

• Can we just keep a list or array for each key of all associated values?

• Need to keep entire list in *contiguous* memory, to minimize I/O complexity of printing entire list.
  • While maintaining linear space.
  • Avoid dynamic memory allocation if possible.

• To find k’s list quickly, keep a *second* cuckoo hash table of (k, (ptr(k))) pairs, where ptr(k) is a pointer to the front of k’s list.
Multiqueues

- Recall: must keep each queue in contiguous memory.
- Ideally, would just assign one block to each $k$. But this uses superlinear space for $B=\omega(1)$.
- Instead, force many queues to share a block of memory.
  - If a block overflows, need to split the block into two.
  - If a block becomes under-populated, need to merge it with another block.
- If a queue gets big enough, it deserves its own block(s) of memory.
  - Mark the queue as heavy and give it its own block(s).
  - If a heavy queue’s size falls due to deletions, return it to sharing its block with other queues (heavy-to-light transition).
- Merges, splits, light-to-heavy, and heavy-to-light transitions are expensive! Make sure they don’t happen often.
Our Solution, Step-by-Step

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>removeAll(k)</td>
<td>To do a removeAll, just delete k from the hash table (takes O(1) I/Os in the worst case).</td>
</tr>
<tr>
<td>count(k)</td>
<td>Clean up memory used by k’s queue in background of subsequent operations (de-amortization).</td>
</tr>
</tbody>
</table>

• Store count(k) with (k, ptr(k)) pair in Primary Structure, for constant-time access.
De-Amortization

– Deamortization surprisingly tricky, many cases to consider.
– Idea: Procrastinate. When splitting/merging, only move a few (k,v) pairs at once.
– Can move rest of them in the background as other operations happen.
– No groundbreaking new techniques, but need to be methodical and a little clever.
Simulation Results

• Insert 1 million keys, then alternate insertion/deletion operations from Zipfian distribution.
  – Tested under range of parameters.
• Memory usage loads of 0.33 – 0.40.
• Average of less than 3 I/Os per operation
• Maximum of less than 50 I/Os per operation.
  – For deamortized implementation.
  – Original amortized implementation, maximum was ~10 times worse.
  – Can trade off space-usage vs I/O cost by tweaking how aggressive you are in splitting or merging multiqueues.
  – Deamortizing made us really think about constant-factor I/O costs that are significant in practice, and how to only move things when absolutely necessary.
Space Usage For De-Amortized Implementation
Simulation Results

Results for basic (non-deamortized) implementation

<table>
<thead>
<tr>
<th>Split parameter</th>
<th>Merge Parameter</th>
<th>Mean I/Os</th>
<th>Std Dev I/Os</th>
<th>Max I/Os</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>3.53</td>
<td>4.24</td>
<td>639</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3.52</td>
<td>4.59</td>
<td>625</td>
</tr>
</tbody>
</table>

Results for deamortized implementation

<table>
<thead>
<tr>
<th>Split parameter</th>
<th>Merge Parameter</th>
<th>Mean I/Os</th>
<th>Std Dev I/Os</th>
<th>Max I/Os</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>2.96</td>
<td>1.75</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2.99</td>
<td>1.83</td>
<td>43</td>
</tr>
</tbody>
</table>
Related/Follow-up Work

• Fully-Deamortized Cuckoo Hashing for Cache-Oblivious Maps and Multimaps [Goodrich, Hirschberg, Mitzenmacher, Thaler, 2011].
  – Focuses on both in-memory cuckoo hashing (lower failure prob.) and cache-oblivious external memory multimaps (cache-oblivious solutions assumes O(1)-time worst-case cache-oblivious dynamic memory allocation mechanism).

• Cuckoo Hashing with Paging [Dietzfelbinger, Mitzenmacher, Rink, 2011]. Experimental study of new external-memory cuckoo hashing scheme meant to achieve look-ups with just a single I/O.
Conclusions

• Multimaps seem like an under-studied, interesting data structure.
• External memory variation natural, offers some challenges.
• We offer a solution based on cuckoo hash tables, multiqueues.
  – Theoretically sound, empirically reasonable.
• In practice, can compare against other solutions.
  – Preliminary experiments suggest we’re competitive with a joint hash table/B-Tree implementation in commercial products (that lacks the same theoretical guarantees).
  – Further optimizations?