Memory-Efficient GroupBy-Aggregate with Compressed Buffer Trees

Hrishikesh Amur
Karsten Schwan
Georgia Tech.

Wolf Richter
David G. Andersen
Athula Balachandran
Erik Zawadzki
Carnegie Mellon

Michael Kaminsky
Intel Labs.
Motivation

Increasing cost of memory

Importance of GroupBy-Aggregate
Need for Memory Efficiency
Decreasing memory capacity per core

1. Disaggregated Memory for Expansion and Sharing in Blade Servers, Lim et al., ISCA'09
DRAM is expensive

- Standard (S)
- Standard (L)
- Hi-memory (XXL)
- Hi-CPU (M)

Amazon EC2 proportional resource cost
GroupBy-Aggregate
GroupBy-Aggregate

GroupBy

Aggregate

key-value pair
GroupBy-Aggregate

GroupBy

key-value pair

Aggregate
GroupBy-Aggregate

GroupBy

Aggregate

key-value pair
MapReduce
MapReduce
MapReduce
MapReduce

Map

Cb.

Map

Cb.

Map

Cb.

Red.

Red.
MapReduce
Implementation of G-A

Map  |  Sort  |  Agg.
--- | --- | ---

Map  |  Hash  |  
--- | --- | ---
Implementation of G-A
Implementation of G-A
Implementation of G-A

Map  Sort  Agg.

Map  Hash  Aggregate as you hash
Implementation of G-A

Sort orders keys along with grouping them

Aggregate as you hash
Sort vs. Hash-based GA

Hash typically outperforms Sort for aggregation workloads\textsuperscript{1,2,3}

1. Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations, Yu et. al., SOSP'09
2. Tenzing: A SQL Implementation On The MapReduce Framework, Chattopadhyaya et al., VLDB'11
3. A Platform for Scalable One-Pass Analytics using MapReduce, Li et al., SIGMOD'11
Sort vs. Hash-based GA

Hash typically outperforms Sort for aggregation workloads\textsuperscript{1,2,3}

1. Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations, Yu et al., SOSP’09
2. Tenzing: A SQL Implementation On The MapReduce Framework, Chattopadhyaya et al., VLDB’11
3. A Platform for Scalable One-Pass Analytics using MapReduce, Li et al., SIGMOD’11

Hash-based G-A requires lots of memory
Hashtable Overheads

<table>
<thead>
<tr>
<th>Allocator</th>
<th>Per-entry memory (B) std::unordered_map</th>
<th>sparse_hash_map</th>
</tr>
</thead>
<tbody>
<tr>
<td>hoard [9]</td>
<td>64.9</td>
<td>67.8</td>
</tr>
<tr>
<td>tcmalloc [21]</td>
<td>57.2</td>
<td>43</td>
</tr>
<tr>
<td>jemalloc [20]</td>
<td>58.1</td>
<td>41</td>
</tr>
</tbody>
</table>

Dataset:
Key: 8B char array
Value: 4B integer
### Hashable Overheads

<table>
<thead>
<tr>
<th>Allocator</th>
<th>Per-entry memory (B) std::unordered_map</th>
<th>sparse_hash_map</th>
</tr>
</thead>
<tbody>
<tr>
<td>hoard [9]</td>
<td>64.9</td>
<td>67.8</td>
</tr>
<tr>
<td>tcmalloc [21]</td>
<td>57.2</td>
<td>43</td>
</tr>
<tr>
<td>jemalloc [20]</td>
<td>58.1</td>
<td>41</td>
</tr>
</tbody>
</table>

**Dataset:**
- **Key:** 8B char array
- **Value:** 4B integer

**Sources of Memory Overhead**

- **Allocator overhead for small heap objects**
- **Indirection overhead (64bit)**
- **Empty slots in hashtable**
How to build a memory-efficient and fast GroupBy-Aggregate?
Approach

Use Compression for Memory Efficiency
Can we compress hashtables?
Can we compress hashtables?

Strawman 1

Compress each entry
Can we compress hashtables?

Strawman 1
Compress each entry

Ineffective compression
Can we compress hashtables?
Can we compress hashtables?

Strawman 2
Compress a block
Can we compress hashtables?

Strawman 2
Compress a block
Frequent compression
Can we compress hashtables?
Can we compress hashtables?

Tension between efficiency of compression and performance
Can we compress hashtables?

Tension between efficiency of compression and performance

But we want both!
Compressed Buffer Trees
Compressed Buffer Trees (CBT)

- In-memory B-tree with each node augmented with a memory buffer
- Inspired by the buffer tree

**Terminology**

- **Partial Aggregation Object (PAO)**
  - User-defined key and value
  - Eg. (char*, uint32) for wordcount,
    (char*, vector<T>) for k-Nearest-Neighbor

<input token> \[\xrightarrow{\text{map()}}\] \[\xrightarrow{\text{reduce()}}\]
Insert PAO
1. Insert PAO

2. Full root:
1. Insert **PAO**

2. Full root:
   a. sorted
1. Insert PAO

2. Full root:
   a. sorted
   b. aggregated
1. Insert **PAO**

2. Full root:
   - a. sorted
   - b. aggregated
   - c. spilled
① Insert PAO

② Full root:
   a. sorted
   b. aggregated
   c. spilled
1. Insert PAO

2. Full root:
   a. sorted
   b. aggregated
   c. spilled
① Insert PAO

② Full root:
   a. sorted
   b. aggregated
   c. spilled

③ Copied fragments are compressed
1. Insert PAO

2. Full root:
   a. sorted
   b. aggregated
   c. spilled

3. Copied fragments are compressed

4. Further spills create more fragments
Copied fragments are compressed

Further spills create more fragments

Insert PAO

1. Full root:
   a. sorted
   b. aggregated
   c. spilled

2. Full node:
3. Copied fragments are compressed.

4. Further spills create more fragments.

1. Insert PAO

2. Full root:
   a. sorted
   b. aggregated
   c. spilled

5. Full node:
   a. decomp.
Copied fragments are compressed.

Further spills create more fragments.

Full root:
- a. sorted
- b. aggregated
- c. spilled

Full node:
- a. decomp.
- b. merged
1. Insert PAO

3. Copied fragments are compressed

4. Further spills create more fragments

2. Full root:
   a. sorted
   b. aggregated
   c. spilled

5. Full node:
   a. decomp.
   b. merged
   c. aggregated
① Insert **PAO**

③ Copied fragments are compressed

④ Further spills create more fragments

② Full root:
  a. sorted
  b. aggregated
  c. spilled

⑤ Full node:
  a. decomp.
  b. merged
  c. aggregated
  d. spilled

---

Monday, December 3, 2012
Copied fragments are compressed.

Further spills create more fragments.

Insert PAO

Full root:
- sorted
- aggregated
- spilled

Full node:
- decomp.
- merged
- aggregated
- spilled
⑥ After all inserts, tree is flushed
After all inserts, tree is flushed
6 After all inserts, tree is flushed
After all inserts, tree is flushed
6 After all inserts, tree is flushed
6 After all inserts, tree is flushed
6. After all inserts, tree is flushed
After all inserts, tree is flushed

Aggregated results available in leaves
CBT Operation (recap)

- PAO\textsuperscript{s} always inserted into root buffer
- If root full, sort PAO\textsuperscript{s}, aggregate and spill
- Spilled buffer fragments are compressed in memory
- If child is full, decompress fragments, merge and spill recursively
- Flush tree at the end
Compressed Buffer Tree
Compressed Buffer Tree

Memory efficiency through compression

Being emptied
Compressed Buffer Tree

Memory efficiency through compression
Compressed Buffer Tree

Memory efficiency through compression

Effective compression through use of large buffers
Compressed Buffer Tree

Memory efficiency through compression

Effective compression through use of large buffers

Strawman 2

Frequent compression
Compressed Buffer Tree

Memory efficiency through compression

Effective compression through use of large buffers
Compressed Buffer Tree

- Memory efficiency through compression
- Effective compression through use of large buffers
- High performance through buffering
Implementation

![Bar graph showing aggregation throughput and CPU time for various implementation methods.]

**Aggregation throughput (×10^6 keys/s):**
- Baseline
- +Comp
- +AsynSort
- +AsynComp
- +Prio
- +DoubBuf
- +StructBuf
- +SpecComp

**CPU time (×10^{-2} cpu-seconds/s):**
- Monday, December 3, 2012
# Performance

## Microbenchmark

<table>
<thead>
<tr>
<th>Application</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wordcount</td>
<td>Key: Random 8B char array</td>
</tr>
<tr>
<td></td>
<td>Value: 4B uint</td>
</tr>
</tbody>
</table>

## Applications:

<table>
<thead>
<tr>
<th>Application</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tri-gram count</td>
<td>Project Gutenberg ebooks</td>
</tr>
<tr>
<td>Clustering</td>
<td>MIT Tiny Image Dataset</td>
</tr>
<tr>
<td>Pagerank</td>
<td>Twitter follower network</td>
</tr>
</tbody>
</table>
Memory Usage: CBT vs. HT

- CBT: Compressed Buffer Tree
- HT: Google sparse_hash_map

Number of unique keys in dataset (×10^6):
- 11
- 23
- 35
- 47
- 59
- 71

Per-key Memory (B):
- Memory (left y-axis)
- Throughput (right y-axis)

Aggregation throughput (×10^6 keys/s):
- Memory Usage: CBT vs. HT

Monday, December 3, 2012
Throughput: CBT vs. HT-C

Application: Wordcount
Dataset:
Key: 8B char array
Value: 4B uint

<table>
<thead>
<tr>
<th></th>
<th>CBT</th>
<th>HT-C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compressed Buffer Tree</td>
<td>TBB concurrent_hash_map</td>
</tr>
</tbody>
</table>

Monday, December 3, 2012
Performance

- CBT: Compressed Buffer Tree
- HT: Google sparse_hash_map
- HT-C: TBB concurrent_hash_map

Per-key Memory (B)
- Memory (left y-axis)
- Throughput (right y-axis)

Aggregation throughput ($\times 10^6$ keys/s)

Applications:
- Trigram
- Clustering
- Pagerank

Monday, December 3, 2012
CBT: Summary

Memory efficiency through compression

Effective compression through use of large buffers

High performance through buffering
Thanks!