Elastic Memory Management for Cloud Data Analytics

Jingjing Wang and Magdalena Balazinska
Large-Scale Cloud Data Analytics

- Memory intensive
- Deployed on clusters
- Shared environments

Large-Scale Data

Memory Management

- Hadoop
- Myria
- Spark
- Impala
- Giraph
Container-Based Cloud Memory Management

- Hard limits: Good for isolation but lack flexibility
- Estimating memory usage before execution is hard
Inaccurate Memory Estimates Affect Performance

- Application failures due to out-of-memory
- Performance degradation due to garbage collection
Our Approach: ElasticMem

• Make container memory limits dynamic

• Allocate memory to multiple applications
  – Perform actions: garbage collection, change mem limits, etc

• Predict how memory actions affect performance
  – Use predictions to drive memory allocation decisions

• Our focus: analytical (relational) queries in Java-based containers (JVM)
Our Approach: ElasticMem

• Make container memory limits dynamic

• Allocate memory to multiple applications
  – Perform actions: garbage collection, change mem limits, etc

• Predict how memory actions affect performance
  – Use predictions to drive memory allocation decisions
Implementing Dynamic Heap Adjustment in a JVM

- OpenJDK has a rigid design:
  - Reserve heap space based on user-specified value
  - Cannot be changed during runtime

- But memory overcommitting + 64-bit address space opens up an opportunity
  - Reserve and commit a large address space
    - Does not physically occupy memory
  - Adjust limits according to actual usage
Implementing Dynamic Heap Adjustment in a JVM

- **Resource Manager**
- **Socket**
- **JVM (OpenJDK 7)**
- **Change limit (GROW)**
- **GC (Shrink)**
- **Actions**
- **Dynamic limit**
- **Used**
- **Live**
- **Dead**
- **Kill**

Resources Manager connected to various components, indicating actions such as changing limits, garbage collection, and killing actions. The diagram illustrates the dynamic adjustment of the heap in a JVM.
Our Approach: ElasticMem

• Make container memory limits dynamic

• Allocate memory to multiple applications
  – Perform actions: garbage collection, change mem limits, etc

• Predict how memory actions affect performance
  – Use predictions to drive memory allocation decisions
Dynamic Memory Allocation

• Problem description:
  – Multiple queries sharing memory
  – At each timestep, allocate memory by performing actions
  – Goal: Reduce query times and failures

• 0-1 knapsack problem:
  – Capacity: total memory
  – Items: JVM memory usages after performing actions
  – Item value: defined on multiple attributes
Dynamic Memory Allocation
Values of Actions and States

- Kill (KILL): # of killed queries, fewer is better
- Pause (NOOP): # of paused queries, fewer is better
- Cost to acquire more memory (cost)
  - Time/space efficiency

<table>
<thead>
<tr>
<th>Action</th>
<th>Value.KILL</th>
<th>Value.NOOP</th>
<th>Value.cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>KILL</td>
<td>1</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>NOOP</td>
<td>0</td>
<td>1</td>
<td>N/A</td>
</tr>
<tr>
<td>Others</td>
<td>0</td>
<td>0</td>
<td>time / space</td>
</tr>
</tbody>
</table>

- Value of a state: sum of action values
- Lexicographic order
Values of Actions: Time and Space

• Increase memory limit (GROW):
  – Space: estimated heap growth
    • Maximum heap usage change in the past few timesteps
  – Time: acquiring and accessing memory from OS
    • Run a calibration program

• Reclaim memory (GC actions)
  – Space: size of recycled memory
  – Time: GC time
  – How to predict them from heap states?
Our Approach: ElasticMem

• Make container memory limits dynamic

• Allocate memory to multiple applications
  – Perform actions: garbage collection, change mem limits, etc

• Predict how memory actions affect performance
  – Use predictions to drive memory allocation decisions
Build Performance Models from Heap States

- Our focus: analytical (relational) queries
  - Large in-memory data structures

- Pick hash tables as our focus
  - Predict time & space for different GCs from stats
Features of Hash Tables

- # of tuples
- # of keys
- Schema information
  - # of long columns
  - # of String columns
  - Sum of lengths of String
Features of Hash Tables

- # of tuples & # of keys: Total & delta since last GC
- 7 features to collect, 4 values to predict
Evaluation: GC Models

• Model: M5P in Weka
• Training: generate hash tables with specific feature values
Evaluation: GC Models

- Testing: 17 TPC-H queries, randomly trigger GCs

<table>
<thead>
<tr>
<th>Values to Predict</th>
<th>Relative Absolute Errors (RAE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total size of live object in the young generation (y_{\text{live}})</td>
<td>23%</td>
</tr>
<tr>
<td>Total size of live object in the old generation (o_{\text{live}})</td>
<td>6%</td>
</tr>
<tr>
<td>Time for a young generation GC (\text{gc}_y)</td>
<td>25%</td>
</tr>
<tr>
<td>Time for an old generation GC (\text{gc}_o)</td>
<td>22%</td>
</tr>
</tbody>
</table>
Evaluation: Scheduling

• One Amazon EC2 r3.4xlarge instance
• 4 most memory intensive TPC-H queries with scale factors 1 and 2 on Myria

• Original: OpenJDK 7u85
  – Serial execution / fully parallel
• Elastic: our approach
  – Resubmit: resubmitting killed queries serially after all queries complete
Compare to Serial:
Much Less Time in Most Cases

# of completed queries

<table>
<thead>
<tr>
<th>Total Memory (GB)</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>70</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed Time (s)</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Elastic-Resubmit, U=1/12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original, Serial</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Compare to Fully Parallel: Less Failures, Less Time

• Advantages of ElasticMem:
  – Automatically adjusts concurrency level
  – Faster query executions and fewer failures
  – Low overhead in case serial execution is necessary
GC Time Reduction: Up to 80%

- Different memory increments $U$:  
  - Fixed ($U=500\text{MB}$) or dynamic ($U=1/12$ of free space)
- When memory is abundant, careful tuning of $U$ is not required
Other Results

• Query time saving up to 30%
• Elastic methods use memory more efficiently
ElasticMem: Conclusion

• Scheduling with hard memory limits is inefficient
• Avoid using containers with hard limits by modifying JVM
• Design a scheduling algorithm to allocate memory across multiple applications in real time
  – Build models to predict GC time and space saving
  – Reduce query time up to 30%, GC time up to 80%, use memory more efficiently