Toward Elastic Memory Management for Cloud Data Analytics

Jingjing Wang and Magdalena Balazinska
University of Washington
Motivation

• Big data analytics systems
  – Maximally utilizing memory
Motivation

• Resource management in a shared cluster
  – Sharing & isolation

• Container-based scheduling (YARN, Docker)
  – Estimate the memory needs beforehand, however not easy
Motivation

• Resource management in a shared cluster
  – Sharing & isolation

• Container-based scheduling (YARN, Docker)
  – Estimate the memory needs beforehand, however not easy
Suboptimal yet Fixed Memory Limit

• Performance
  – Out-of-memory failures
  – Garbage collection
  – Performance degradation due to disk-spilling

• Wasteful over-provisioning
• Usability: tedious trial-and-error

Self-join, 10 million tuples, 2 int columns, one machine, one worker
Our Approach: From Fixed to Elastic

• Problem:
  – Multiple queries
  – Limited memory
  – Reducing OOM failure and GC time

• Approach:
  – Elastic memory allocation
Elastic Memory Allocation: Challenges

- Challenge 1: Container memory cap is fixed
- Challenge 2: Cost and benefit of a GC are unknown
- Challenge 3: Limited resources must be shared between queries
Elastic Memory Allocation: Challenges

• **Challenge 1:** Container memory cap is fixed
  – **Solution:** Enable dynamic memory adjustment

• Challenge 2: Cost and benefit of a GC are unknown

• Challenge 3: Limited resources must be shared between queries
Enable Dynamic Memory Adjustment

• We target Java-based containers
• Today: OpenJDK 7
  – Given a maximum heap size before launching JVM
  – Asks the OS for address space and keeps constant
• Observation:
  – Overcommitting memory + 64-bit address space
• Break the ceiling: We modify OpenJDK in two ways:
  – Commit a large address space in the beginning
  – Listen to instructions during runtime
Elastic Memory Allocation: Challenges

• Challenge 1: Container memory cap is fixed

• **Challenge 2: Cost and benefit of a GC are unknown**
  – **Solution**: Build models that predict heap sizes and GC time

• Challenge 3: Limited resources must be shared between queries
Estimate Garbage Collection Cost and Savings

• GC Cost and Savings predictors
  – Number and total size of live/dead objects
• Challenge:
  – Expensive to obtain number and size of live/dead objects
  – Requires traversing object graph
• Observation: Use query plan operator statistics instead
  – Size of hash tables, # of columns, data types, ...
  – Should be correlated with number and size of live/dead objects
• Build models for each operator
• Model for query plan: Sum of individual operator predictions
Single Operator Models

• Assumptions:
  – Two generations: young & old
  – Default collectors (−XX:+UseParallelGC)

# of tuples, total
# of tuples, delta since the last GC
# of keys, total
# of keys, delta since the last GC
Size of a tuple

Input features

Heap sizes:
Live/dead object sizes in the young/old generation

GC times:
User/sys times of the young/old collector

M5P from Weka

Values to predict
Single Operator Models: Training

- Collect training samples using synthetic data
  - Myria query, trigger GC at specific times
- Observation:
  - Not Linear, bad predictions in regions with insufficient training data
  - Uniform fine-grained sampling too expensive
- Collect adaptively and iteratively:
  - Collect samples using a coarse-grained grid
  - Do cross-validation and pick points with highest error rates
  - Split into finer-grained cells
Evaluation: Single Operator

- 10 million tuples, two int columns, 10 iterations
- Test set: randomly triggered GCs on a query with one operator
- 5 values, showing relative absolute error (RAE)
  - ylsize/ylsize: live/dead object size in the young generation
  - olsize/olsize: live/dead object size in the old generation
  - time: GC time

![Graph showing RAE over iterations](image)
Evaluation: Single Operator

- 10 million tuples, two `int` columns, 10 iterations
- Test set: randomly triggered GCs on a query with one operator
- 5 values, showing relative absolute error (RAE)
  - `ylsize/ydsize`: live/dead object size in the young generation
  - `olsize/odsize`: live/dead object size in the old generation
  - `time`: GC time

![Graph showing RAE (Percentages) over iterations for different values](chart.png)

Jingjing Wang - University of Washington
Evaluation: Multiple Operators

JoinAgg: one join + one aggregate

AggJoin: two aggregates + one join

Jingjing Wang - University of Washington
Elastic Memory Allocation: Challenges

• Challenge 1: Container memory cap is fixed

• Challenge 2: Cost and benefit of a GC are unknown

• Challenge 3: Limited resources must be shared between queries
  – Solution: Design algorithms to orchestrate multiple queries
Orchestrating: Adaptive Allocation

• At each timestep $t$, consider $[t, t + \delta t]$
• Knapsack problem:
  – Capacity: total memory
  – Items: queries
  – Weights/costs: a few possible combinations
    • Let it grow: larger heap size, zero GC time
    • Trigger a GC: smaller heap size, some GC time
    • Kill a query: no heap size, one failed query
  – Dynamic programming
    • Minimize: # of OOM then GC time
Orchestrating: Evaluation

- Four queries started at nearly the same time
  - 10 million tuples, two int columns
- # of completed queries:

Jingjing Wang - University of Washington
Orchestrating: Evaluation

GC Times

CPU Times

Query Times

Agg, Agg, Agg, Agg
Orchestrating: Evaluation

Join, Join, Join, Join
Orchestrating: Evaluation

GC Times

CPU Times

Query Times

Total Memory (GB)

10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Time (s)

50 100 150

Time (s)

200 250

Time (s)

200 250

AggJoin, Join, JoinAgg, Agg

Elastic

Fixed

Jingjing Wang - University of Washington
Conclusions

• Analytics in the cloud need orchestration
• Using hard memory limits is unnecessary
• Elastic memory allocation opens up a new world
• Preliminary results show big performance gains