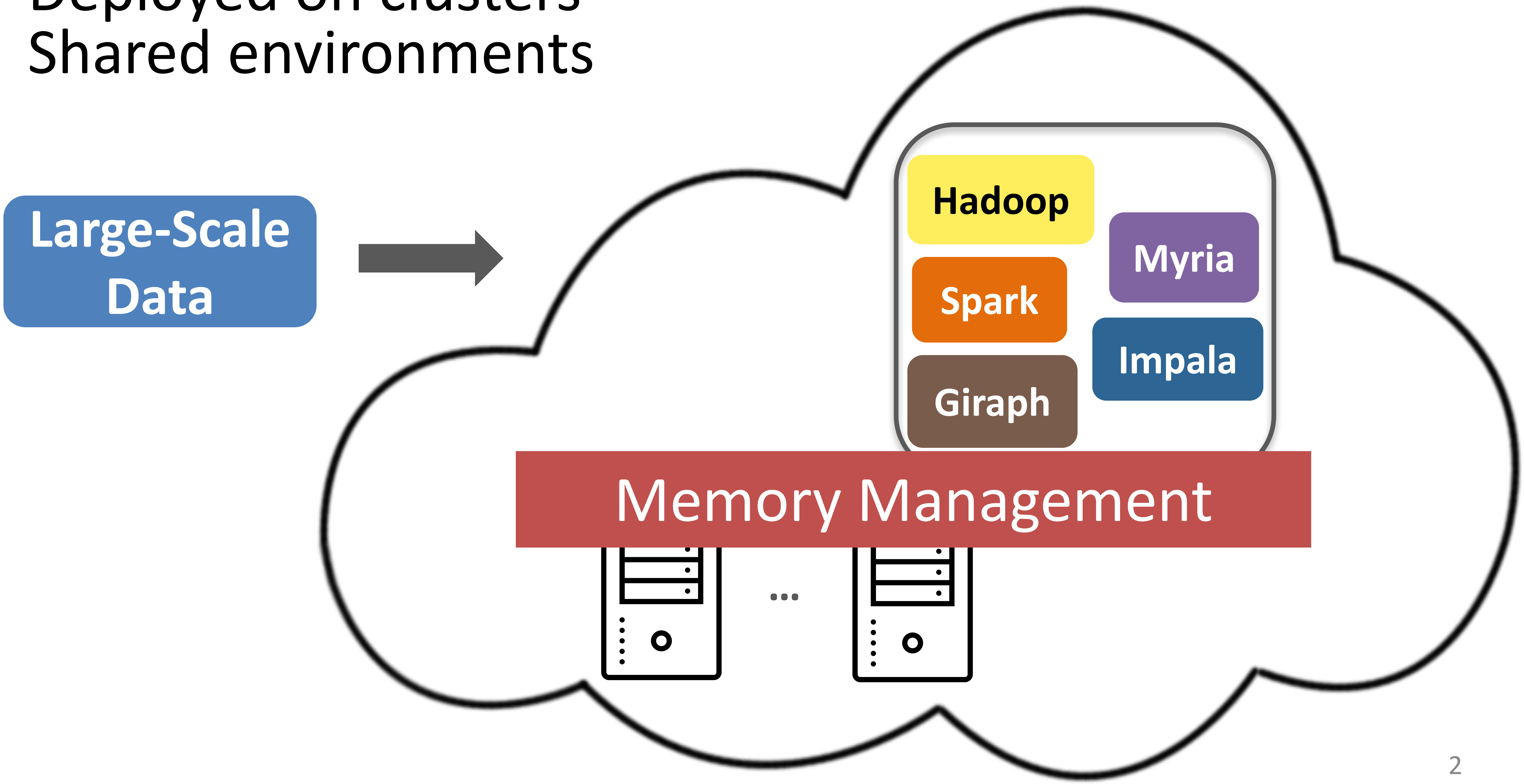
Elastic Memory Management for Cloud Data Analytics

Jingjing Wang and Magdalena Balazinska

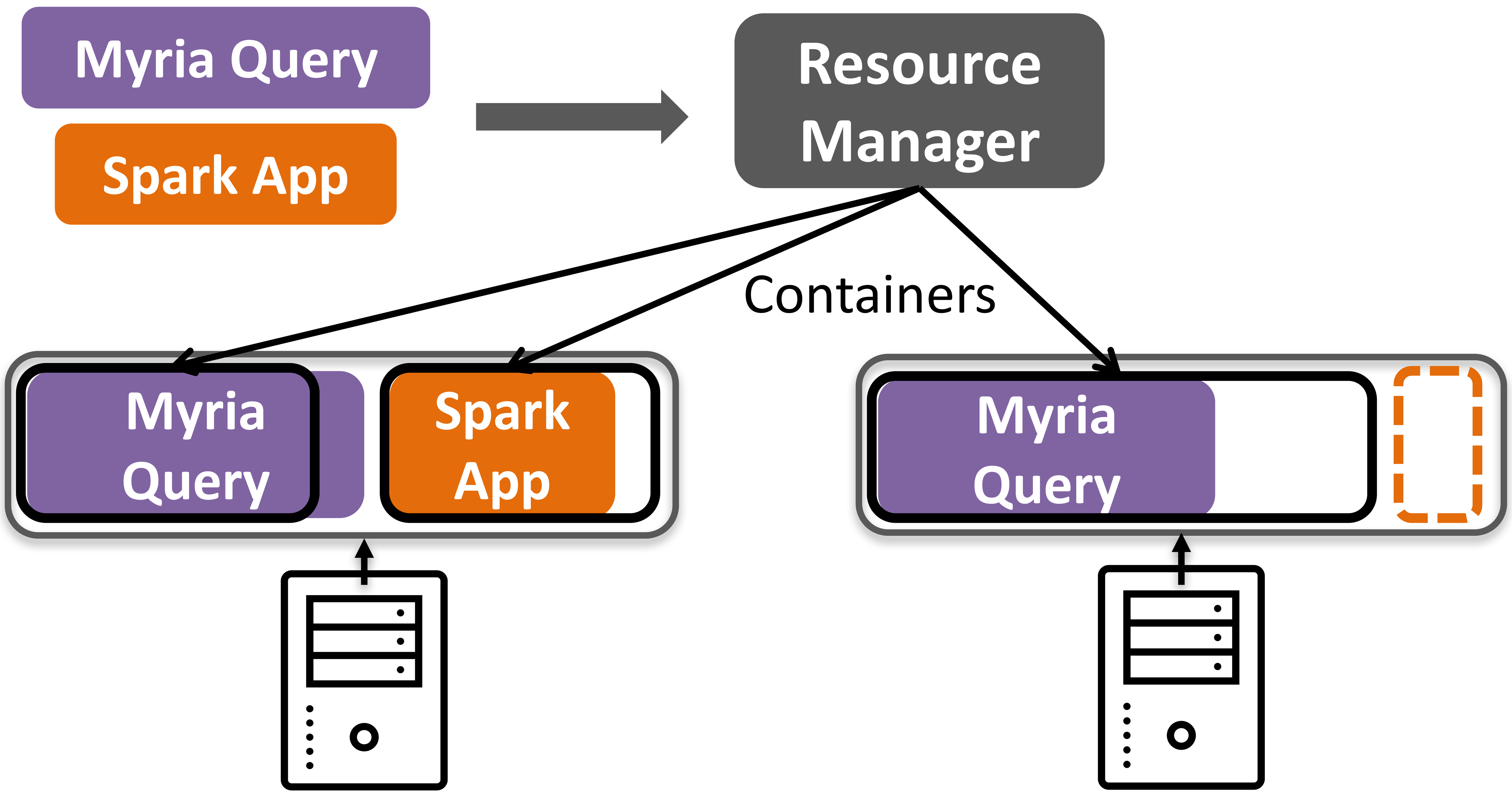
PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

Large-Scale Cloud Data Analytics

- Memory intensive
- Deployed on clusters
- Shared environments



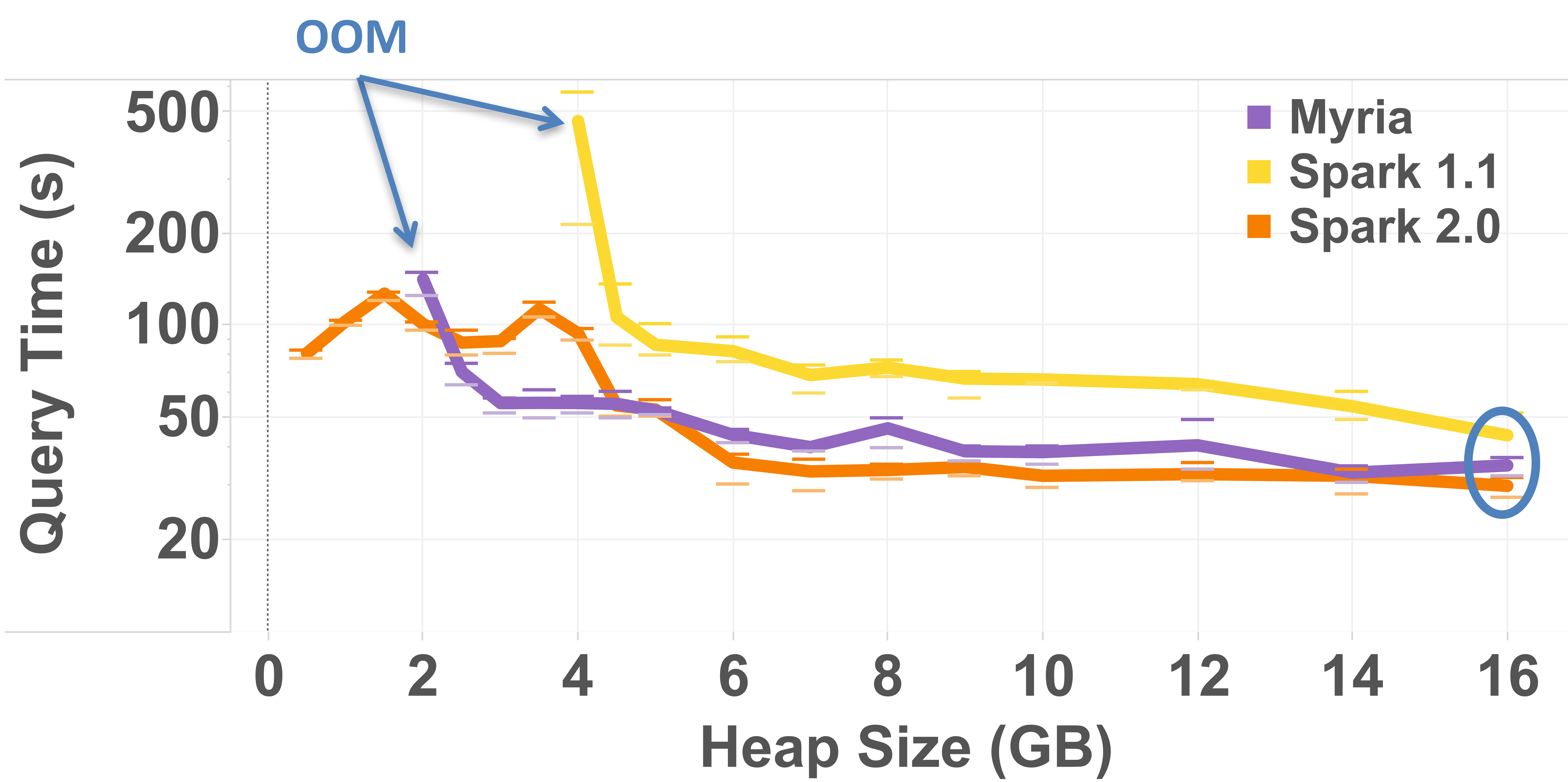
Container-Based Cloud Memory Management



- 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



A self-join query

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)

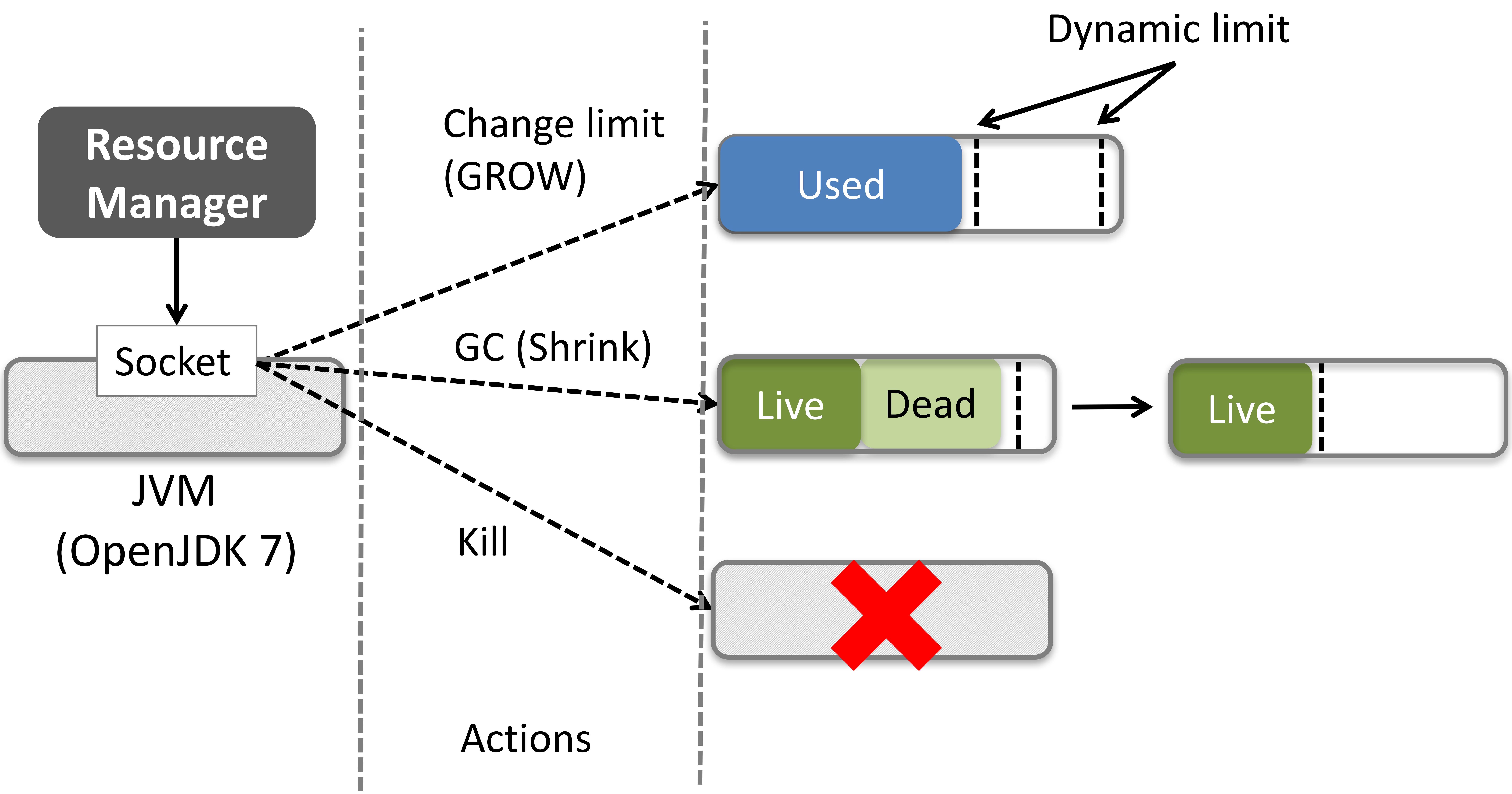
Our Approach: ElasticMem

- **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



Our Approach: ElasticMem

- Make container memory limits dynamic

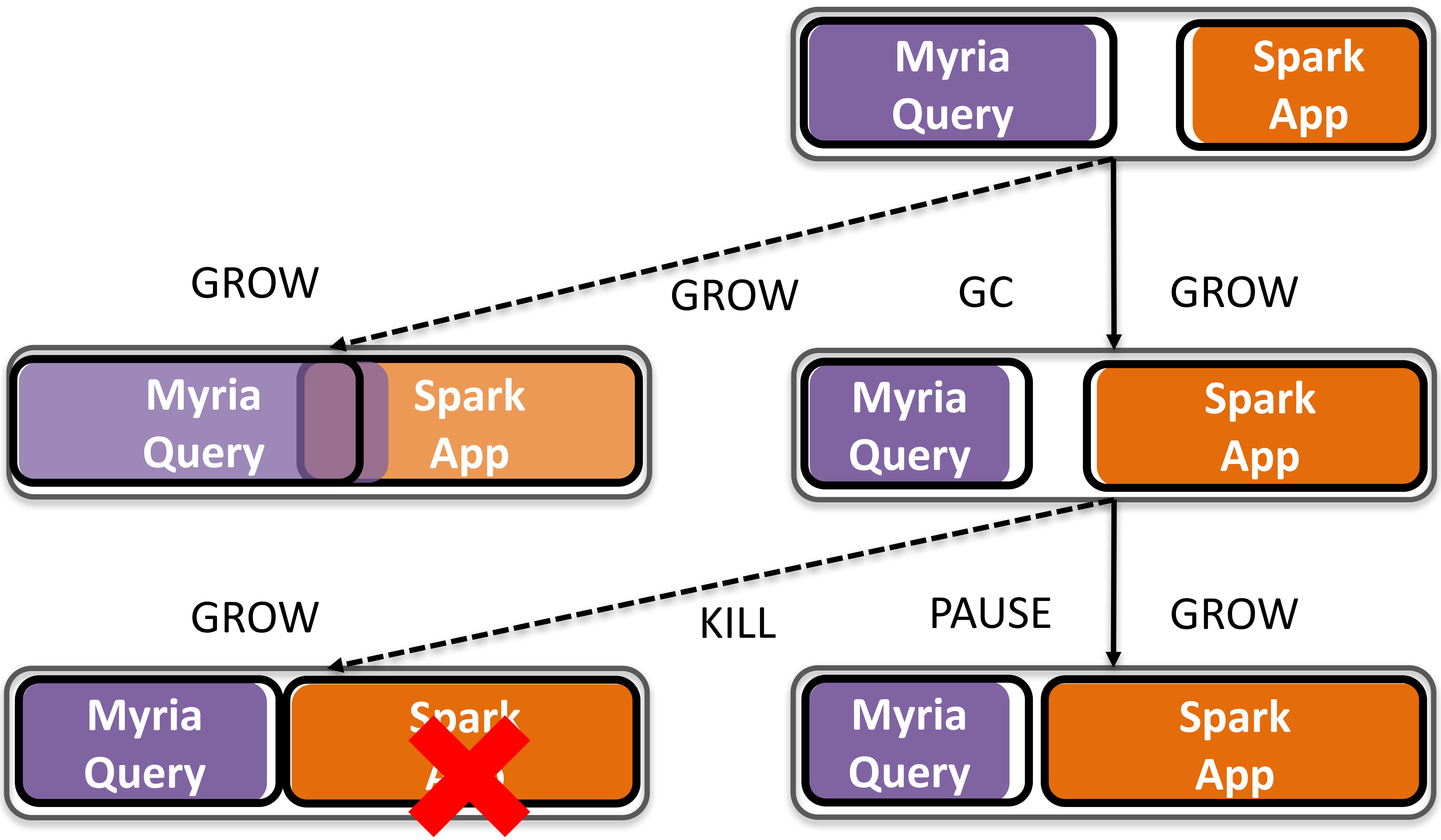
- **Allocate memory to multiple applications**
  - **Perform actions: garbage collection, change mem limits, etc**

- 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

Action	Value.KILL	Value.NOOP	Value.cost
KILL	1	0	N/A
NOOP	0	1	N/A
Others	0	0	time / space

- 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

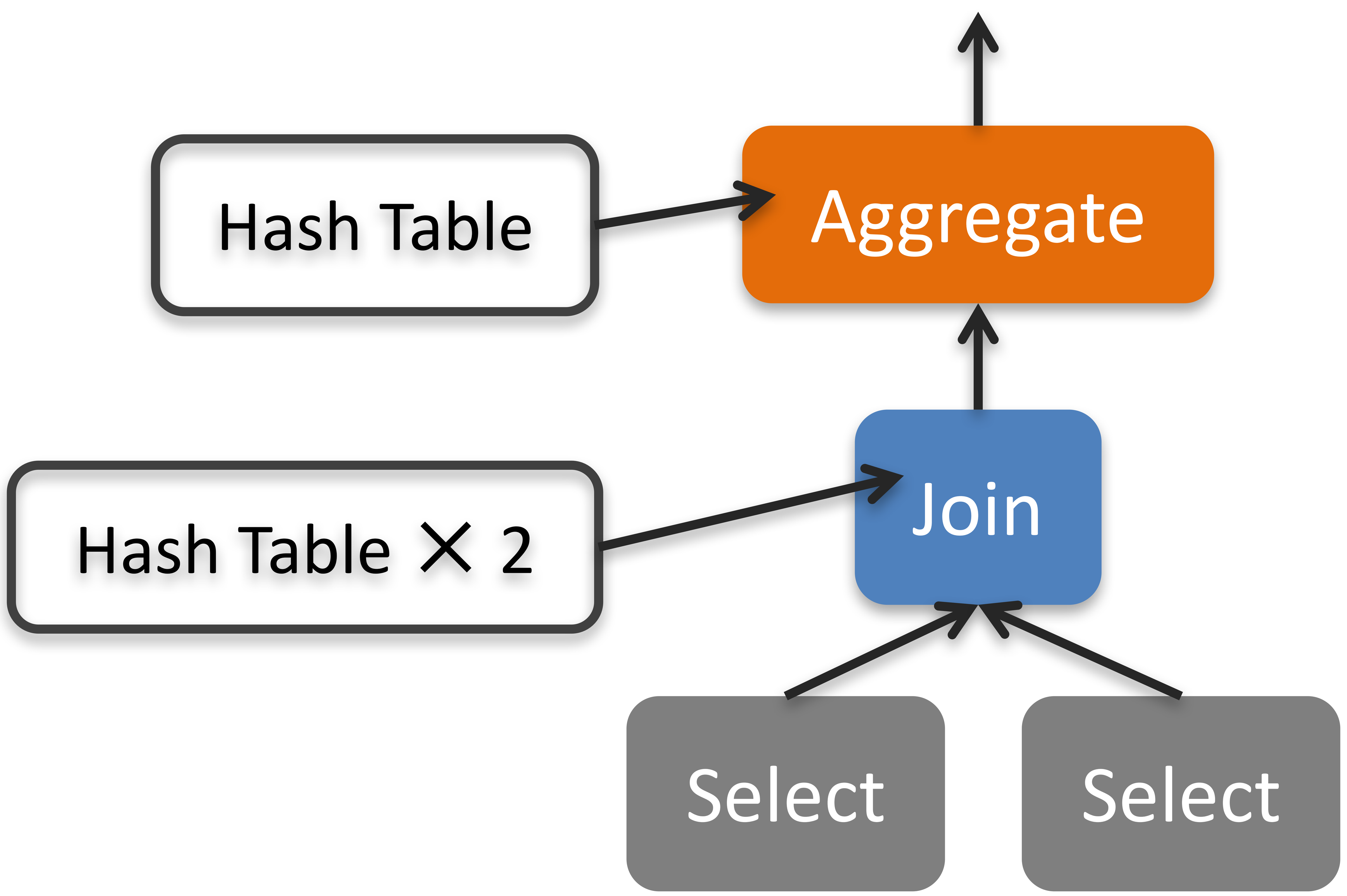
- Make container memory limits dynamic

- 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**

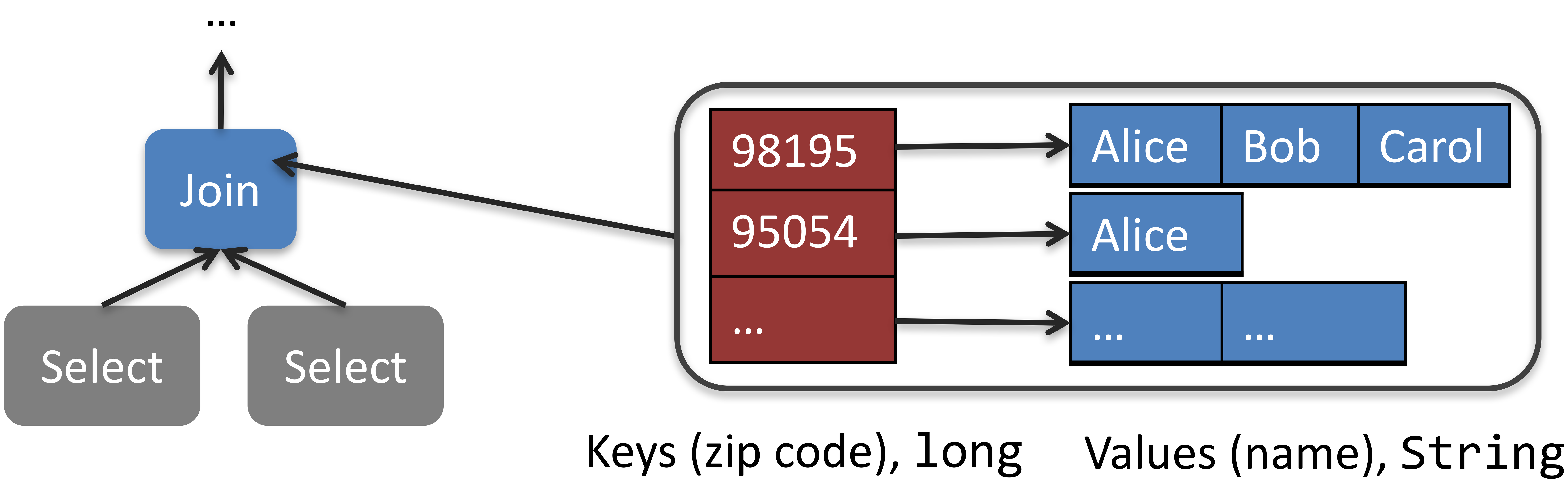
Build Performance Models from Heap States

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



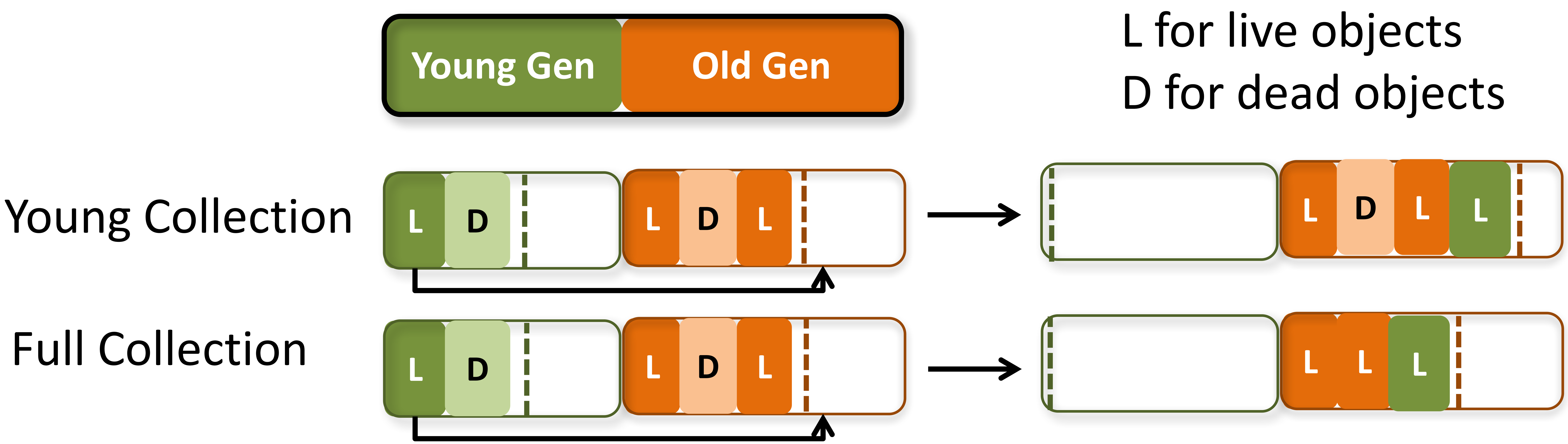
- Pick hash tables as our focus
  - Predict time & space for different GCs from stats

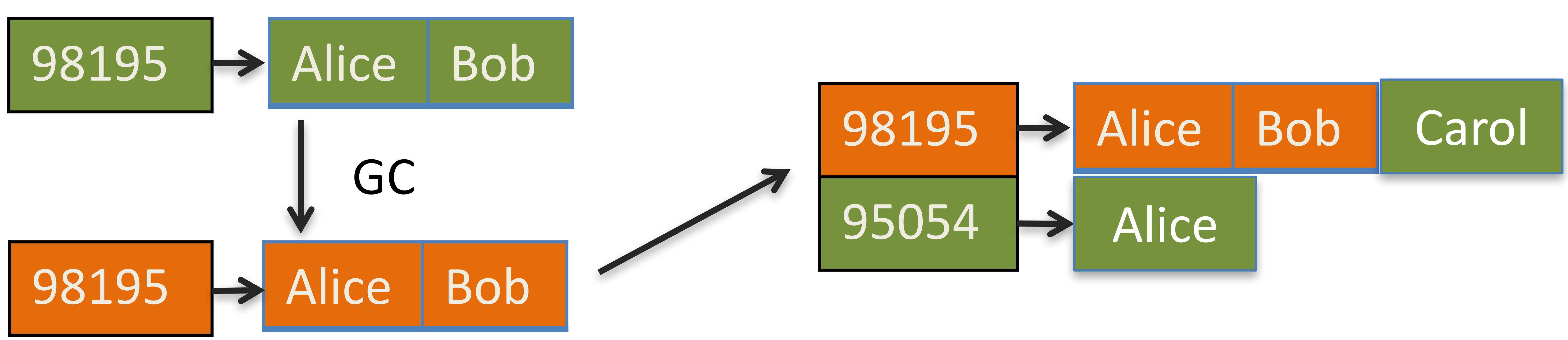
Features of Hash Tables



- # of tuples
- # of keys
- Schema information
  - # of `long` columns
  - # of `String` columns
  - Sum of lengths of `String`

Features of Hash Tables

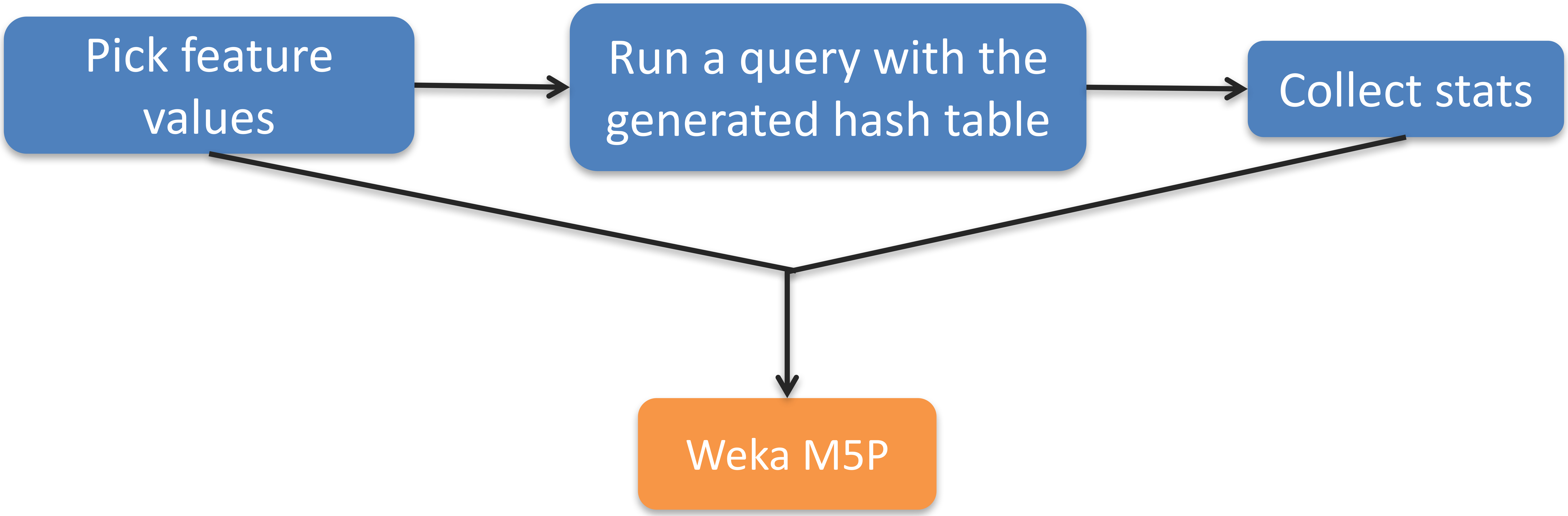




- # of tuples & # of keys: Total & delta since last GC
- 7 features to collect, 4 values to predict

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

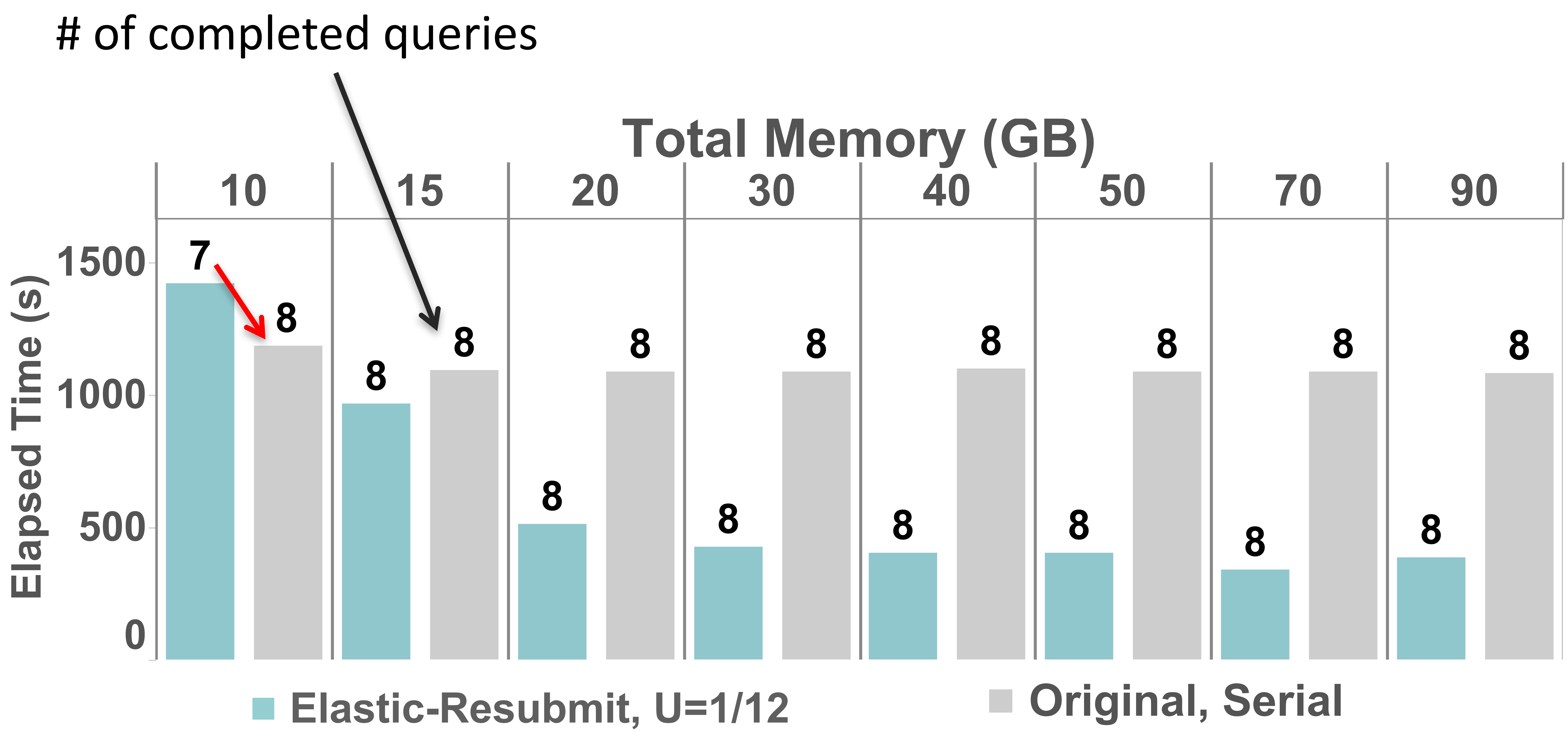
Values to Predict	Relative Absolute Errors (RAE)
Total size of live object in the young generation (y_{live})	23%
Total size of live object in the old generation (o_{live})	6%
Time for a young generation GC (gc_y)	25%
Time for an old generation GC (gc_o)	22%

Evaluation: Scheduling

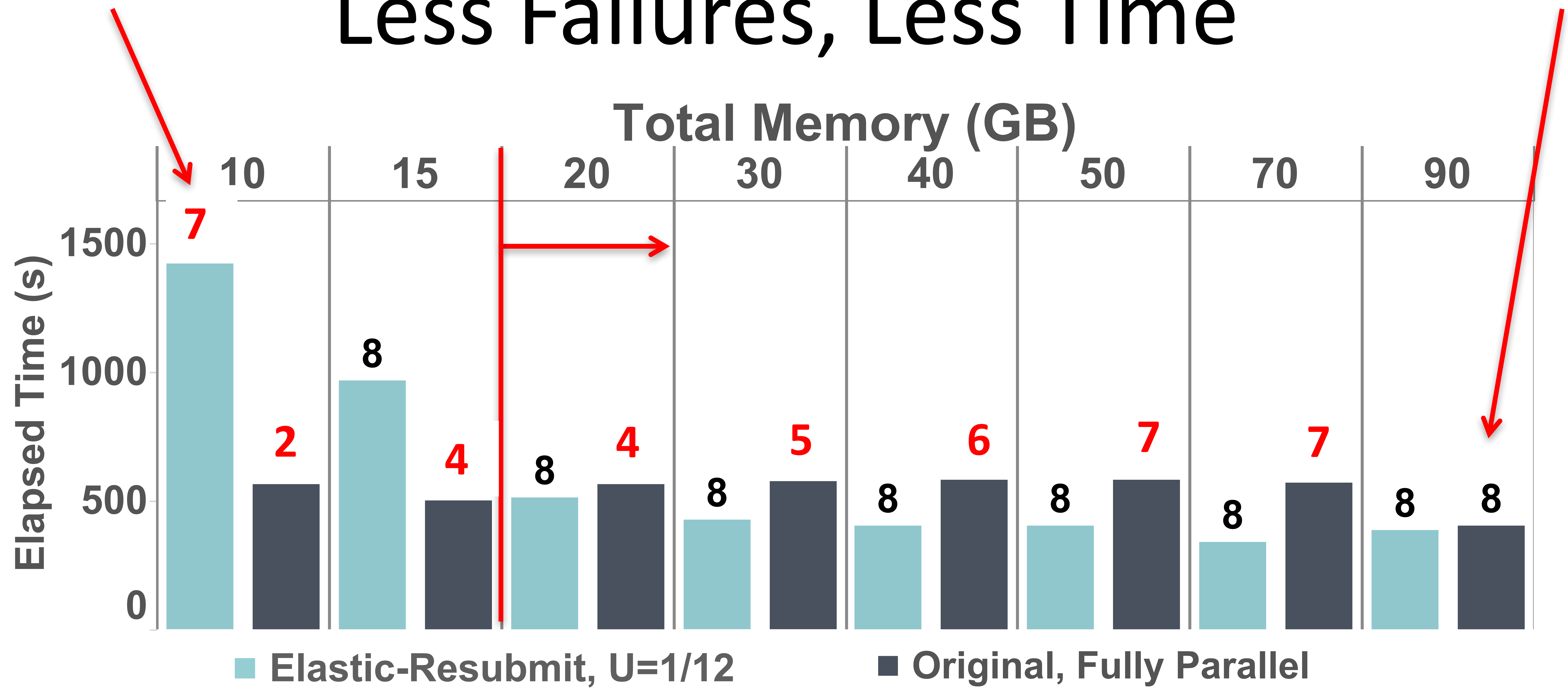
- One Amazon EC2 r3.4xlarge instance
- 4 most memory intensive TPC-H queries with scale factors 1 and 2 on Myria

- 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

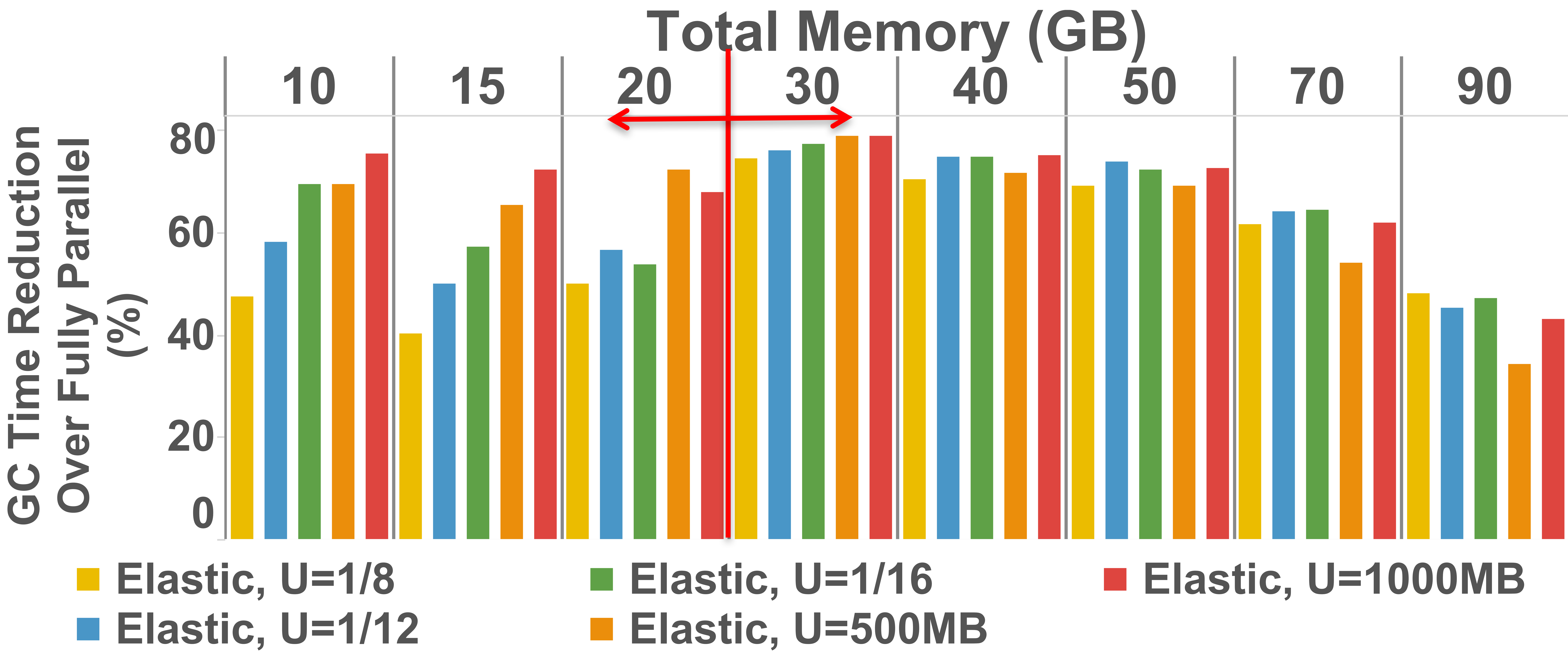


Compare to Fully Parallel: Less Failures, Less Time



- 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=500MB) 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