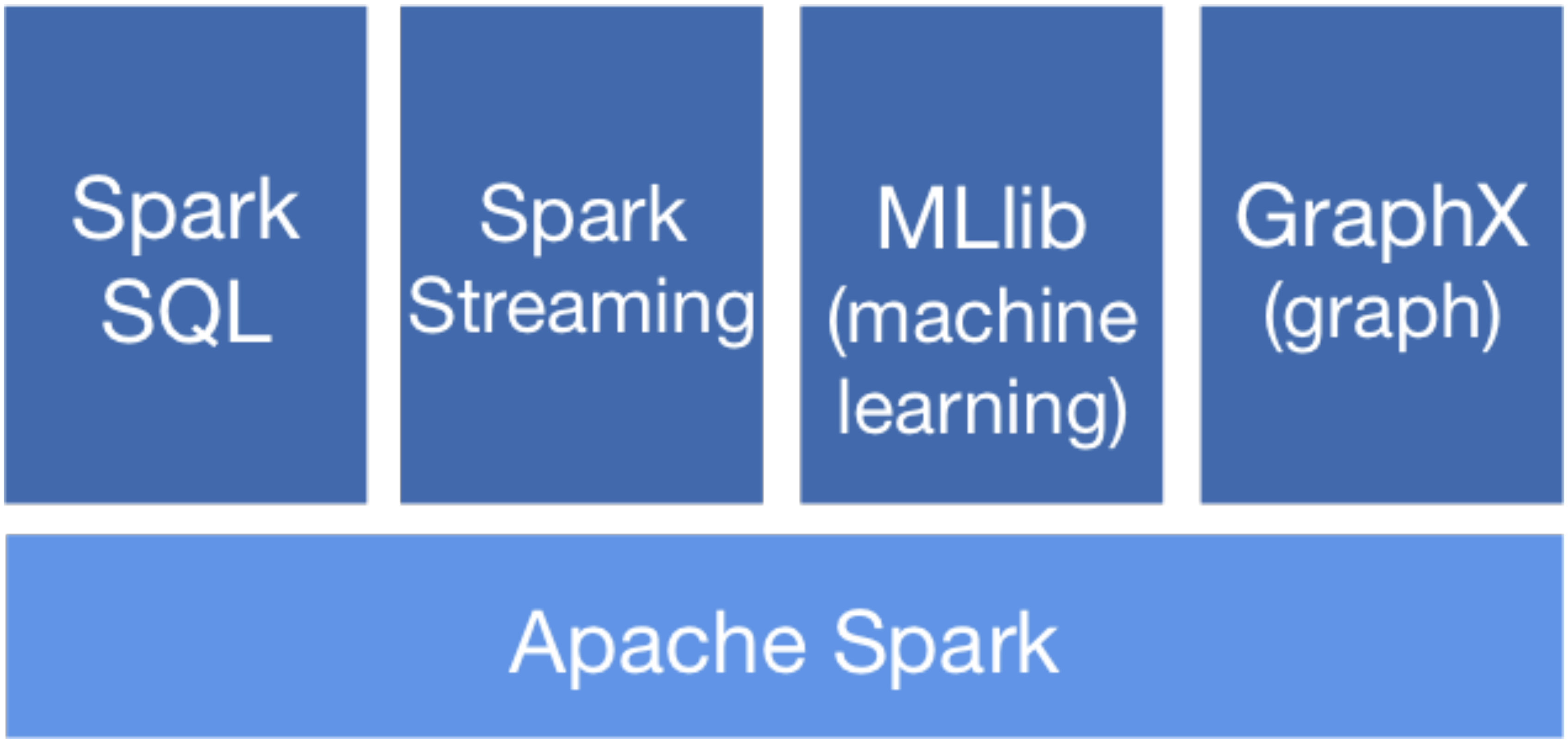
Neutrino: Revisiting Memory Caching for Iterative Data Analytics

Erci Xu*, Mohit Saxena, Lawrence Chiu
Ohio State University*
IBM Research Almaden

Background

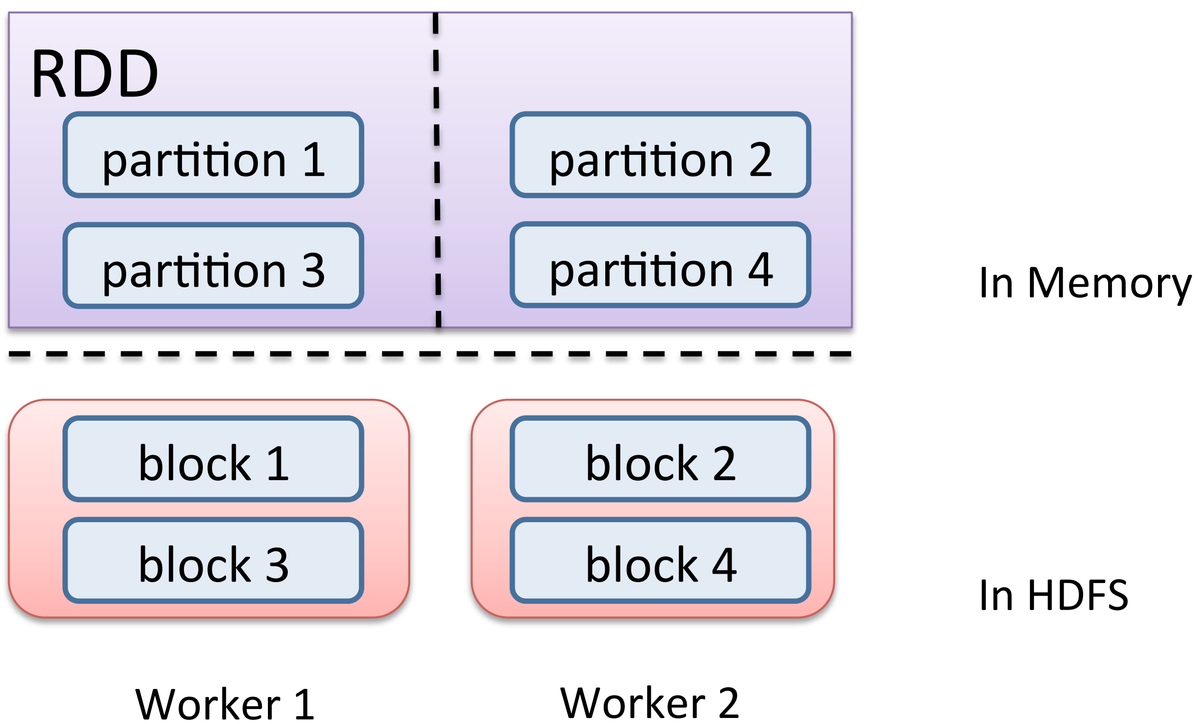
- Iterative analytics is rapidly gaining popularity
  - Data Clustering, Log Mining, Graph Processing, Machine Learning
  - Dataset is repeatedly accessed across different iterations
- In-Memory Caching best fits Iterative Analytics
  - In-Memory caching frameworks avoid frequent I/O with underlying storage systems
  - Iterative Data Analytics could have 10x – 100x speedup

Spark for In-Memory Iterative Analytics



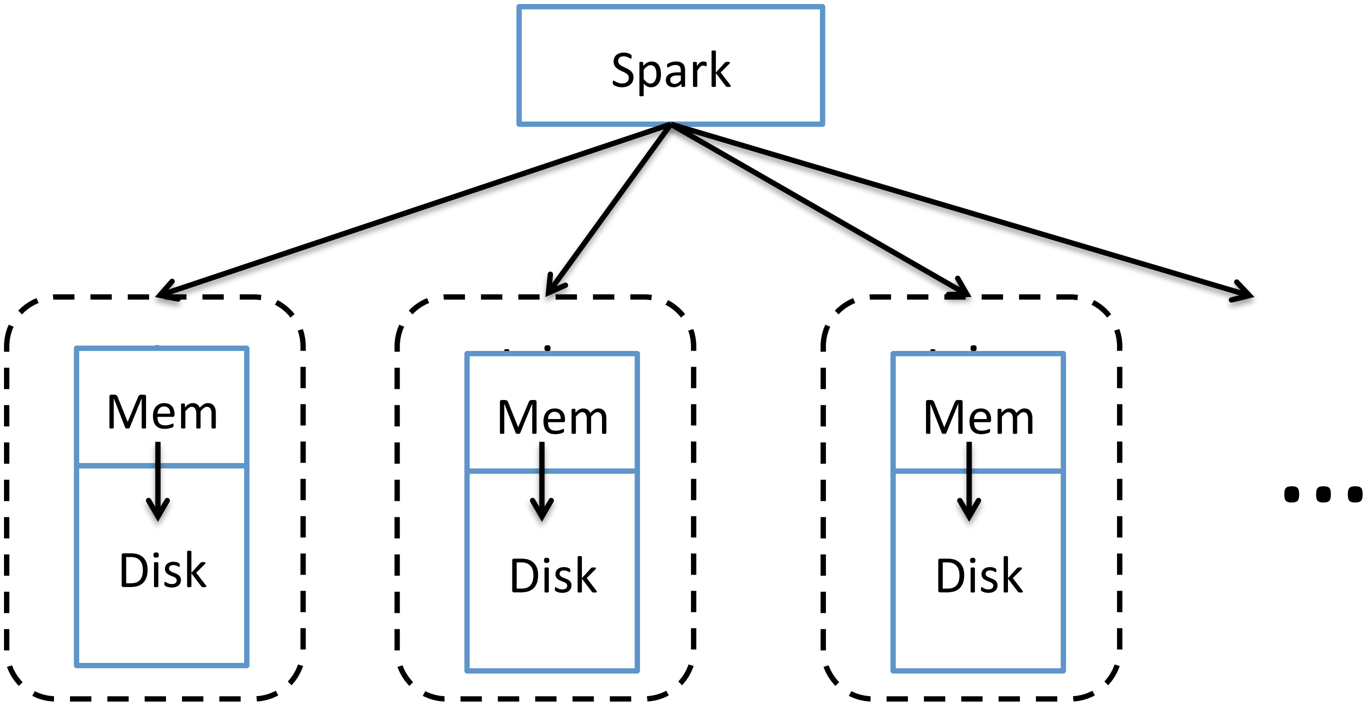
Ref: wikibon.org

Spark RDD



RDD: Resilient Distributed Datasets

Memory Caching in Spark



RDD Cache Options

- Deserialized or Serialized
- On Heap or Off Heap
- In Memory or Disk

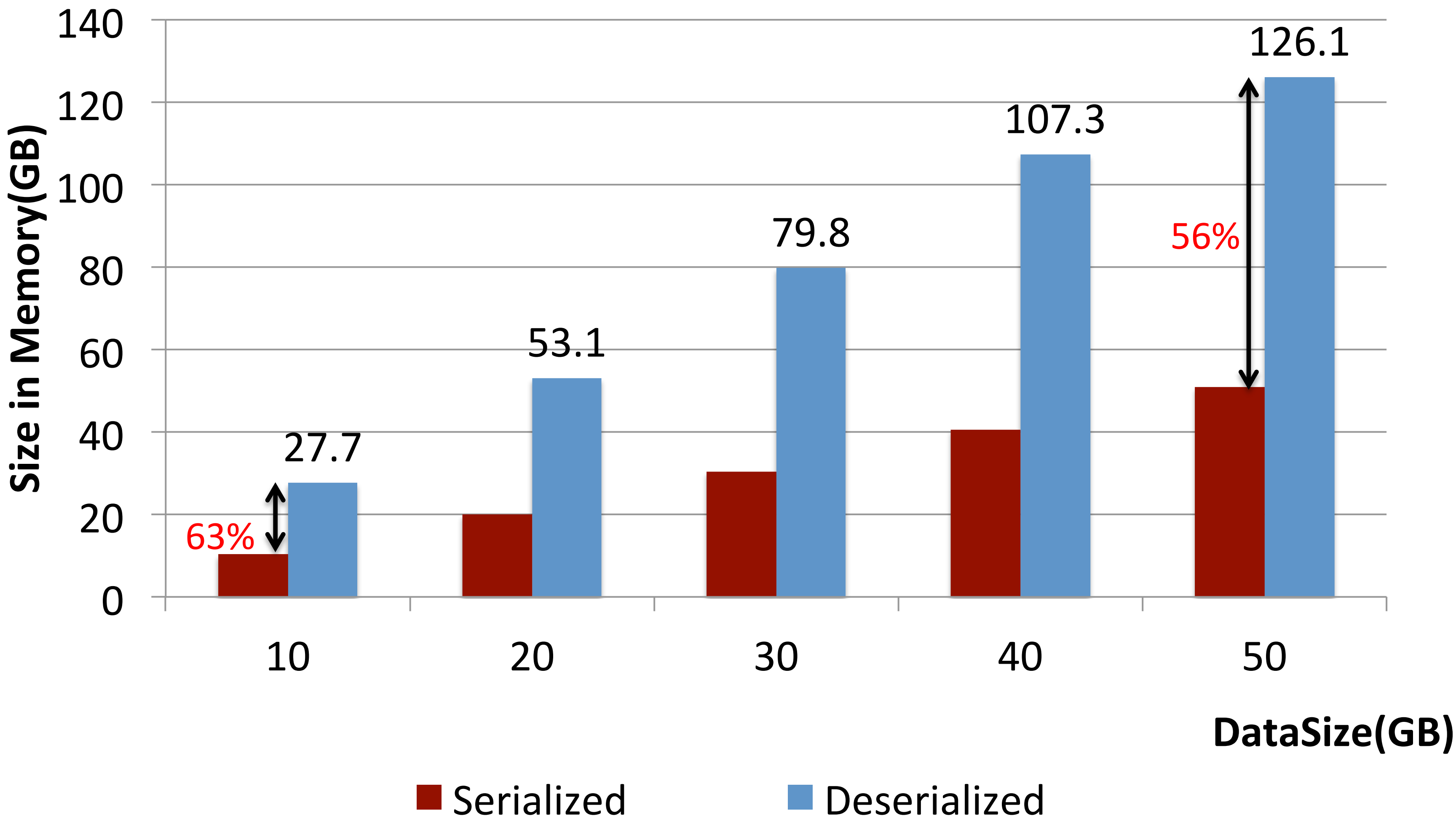
Problems: In-memory Caching for Iterative Data Analytics

1. Discrete Cache Levels

2. Manual Programmer Management

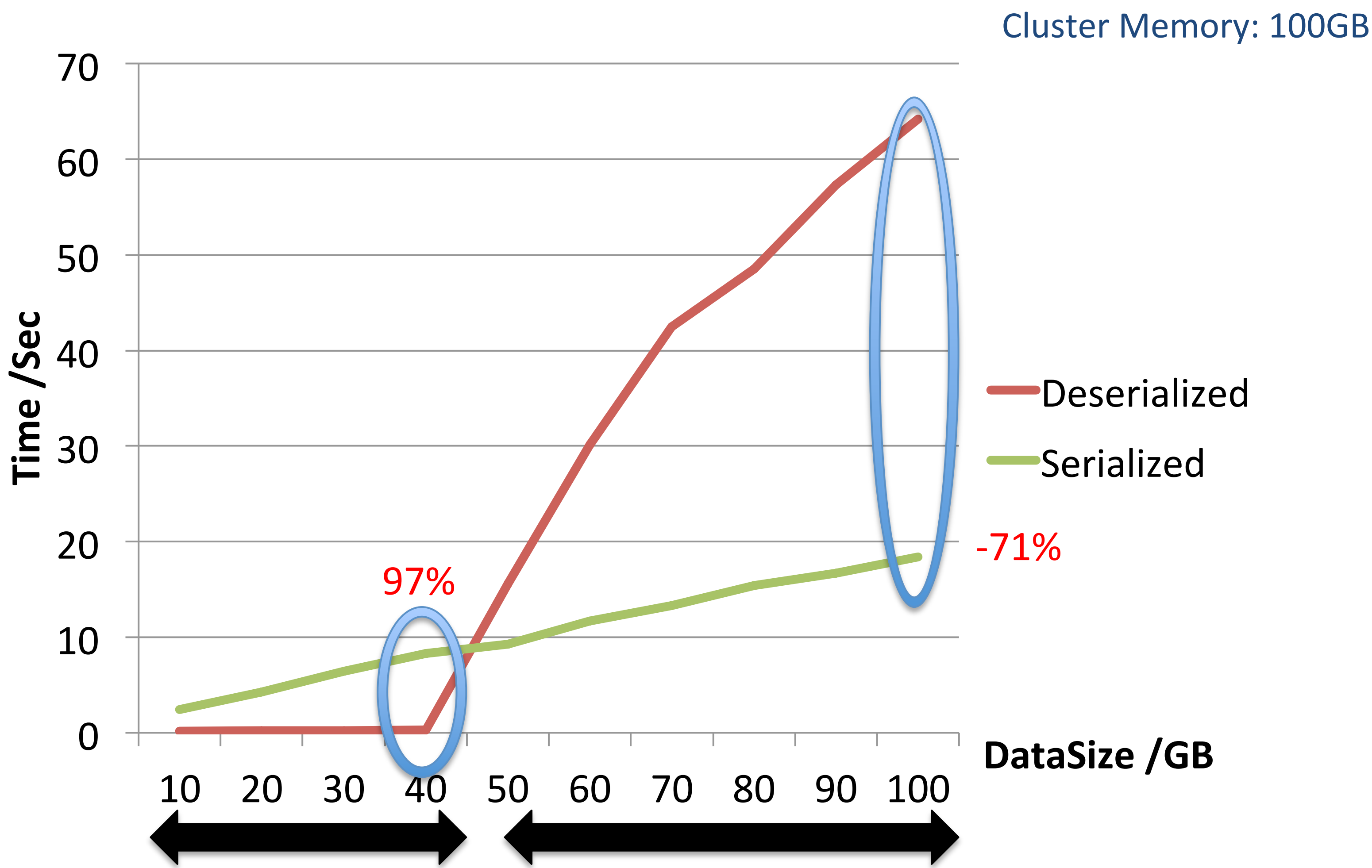
3. Not Adaptive to Runtime Changes

Problem 1: Discrete Cache Levels



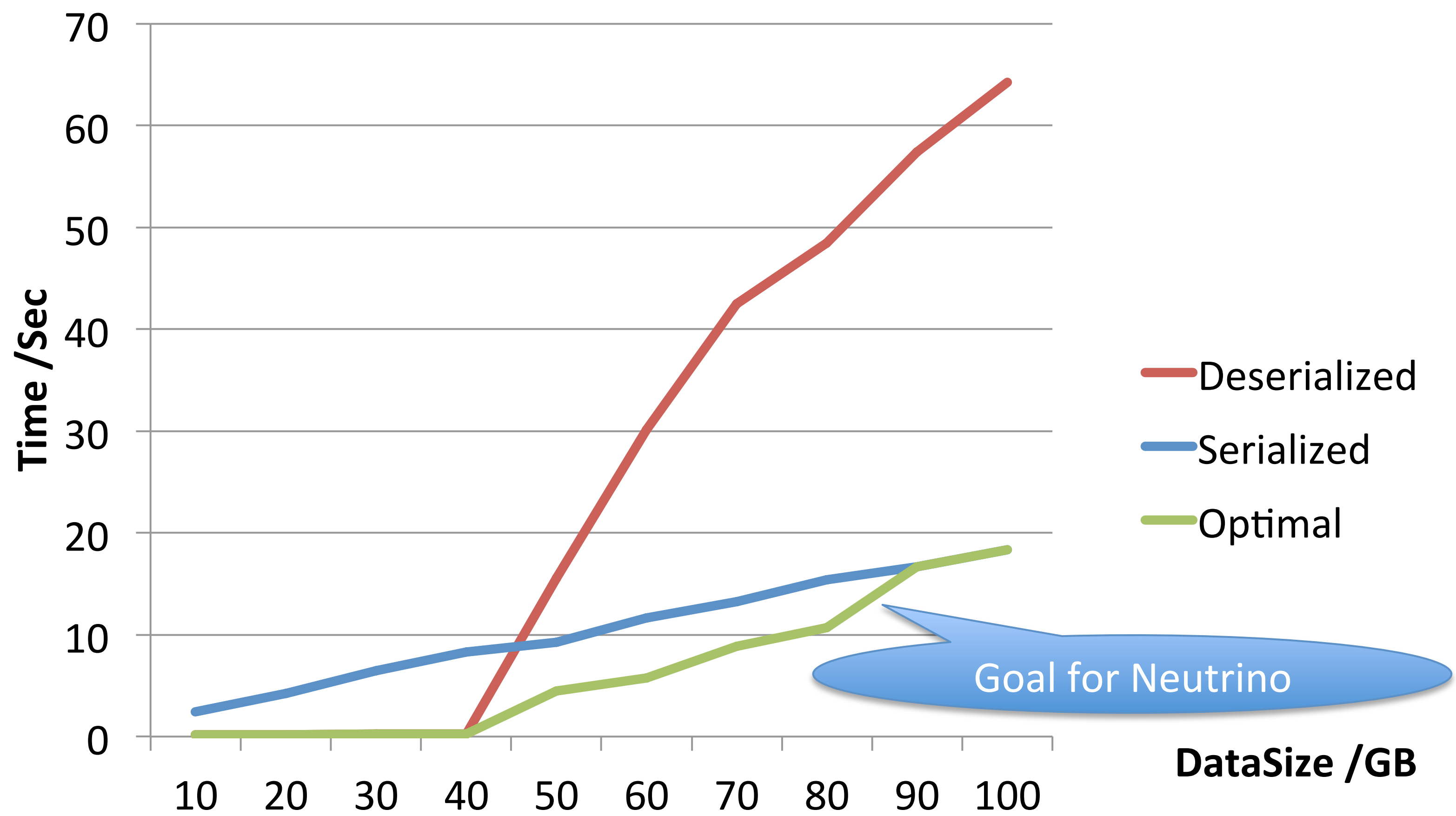
Serialized Cache saves 56% to 63% of the space but relatively slower

Problem 1: Discrete Cache Levels



- Deserialized Cache is an order of magnitude faster but become very slow once spilled to disk

Problem 1: Discrete Cache Levels



Problems: In-memory Caching for Iterative Data Analytics

1. Discrete Cache Levels
2. Manual Programmer Management
3. Not Adaptive to Runtime Changes

Problem 2: Manual Management

```
rdd_1 = sc.textfile(HDFS://file1)
rdd_2 = sc.textfile(HDFS://file2)
```

dataset size?

```
rdd_1.persist(Cache_Level)
rdd_2.persist(Cache_Level)
```

de/serialized?

```
rdd_1.tranformations().action()
rdd_2.tranformations().action()
```

access order ?

Problems: In-memory Caching for Iterative Data Analytics

1. Discrete Cache Levels

2. Manual Programmer Management

3. Not Adaptive to Runtime Changes

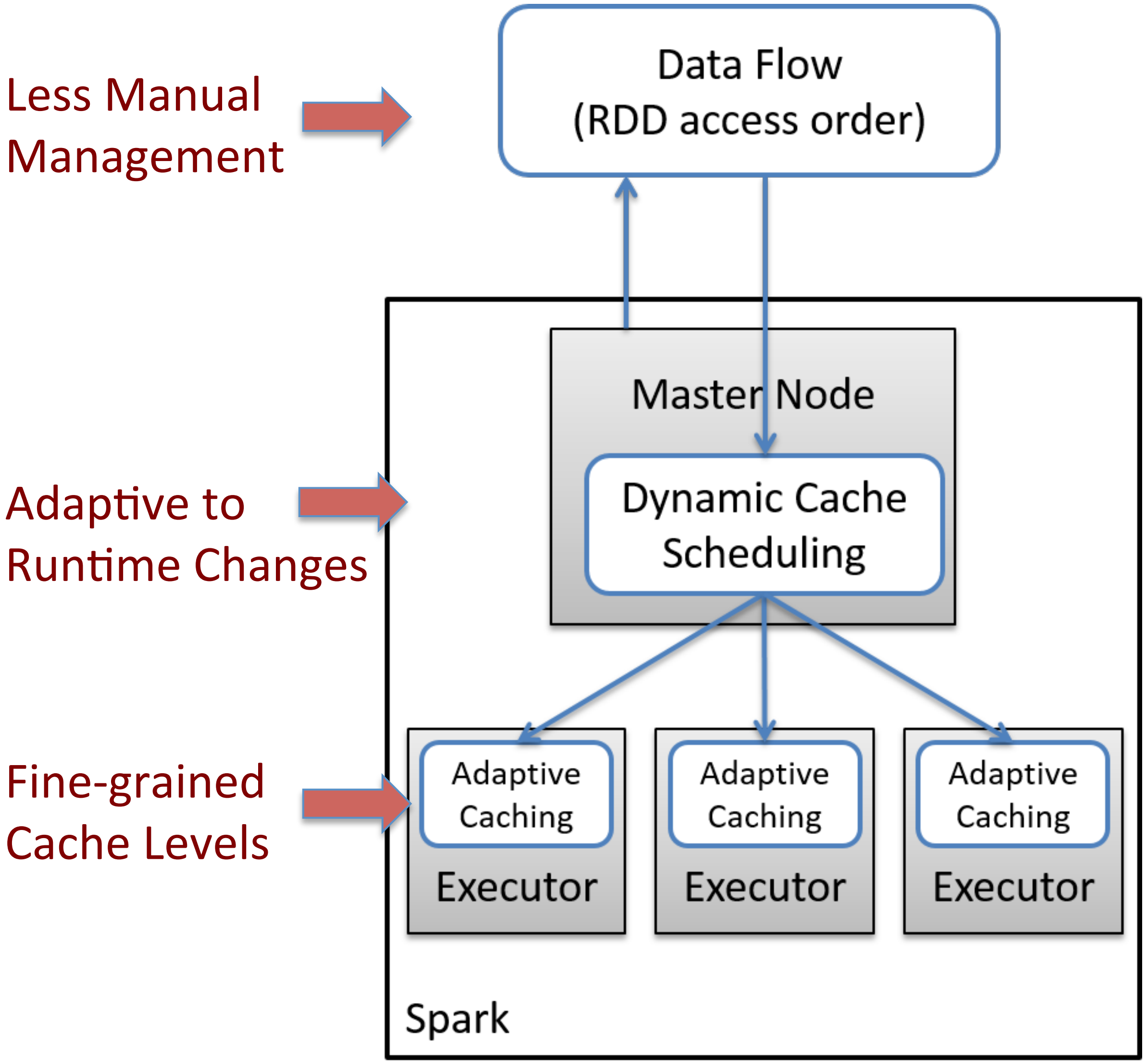
Not Adaptive to Runtime Changes

Cache levels are statically assigned to RDD and such programmer decisions may not adapt to:

1. Changing memory utilization on each worker node

2. Different memory requirement for a RDD partition in deserialized/serialized cache levels

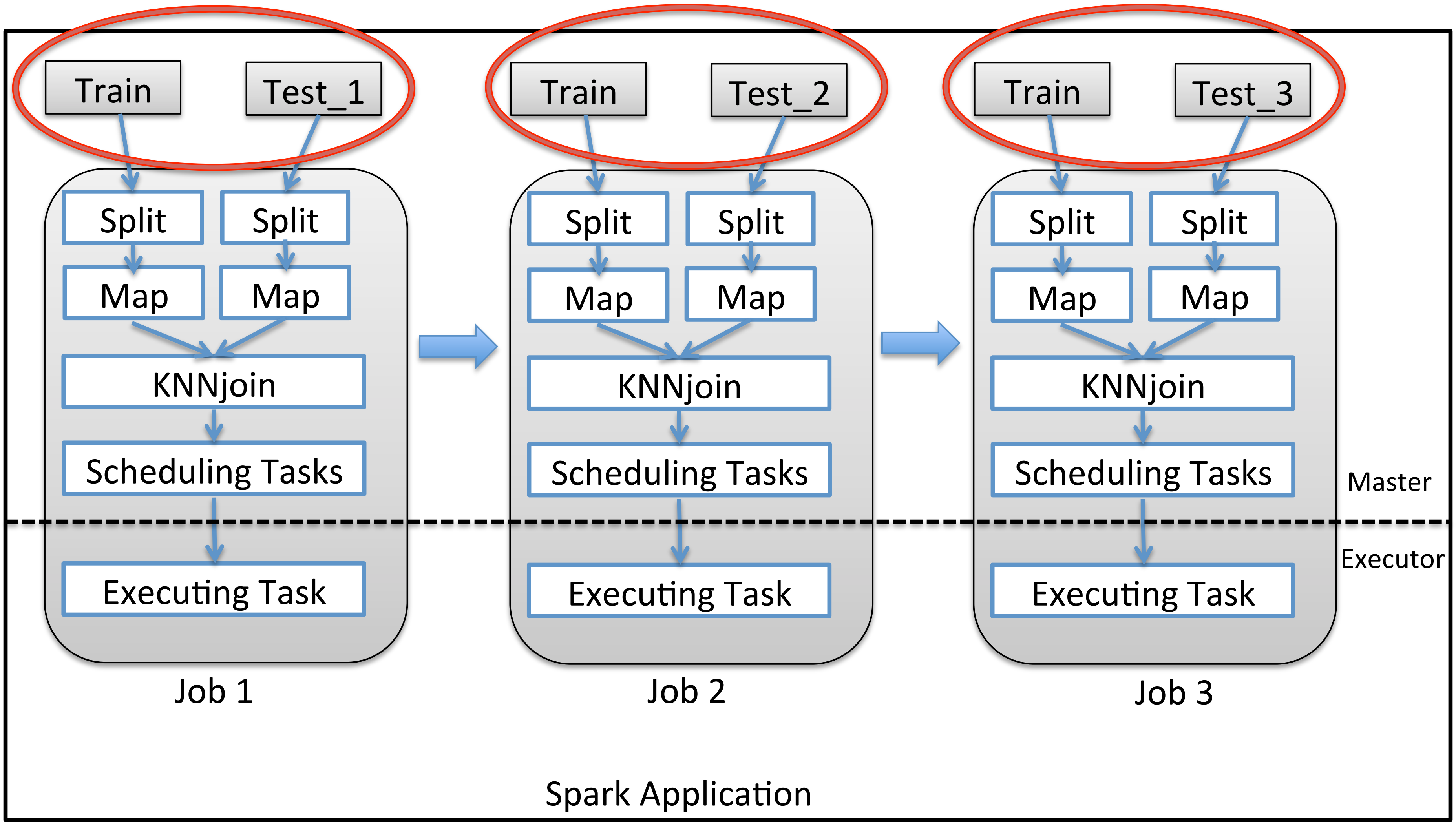
Our Solution: Neutrino



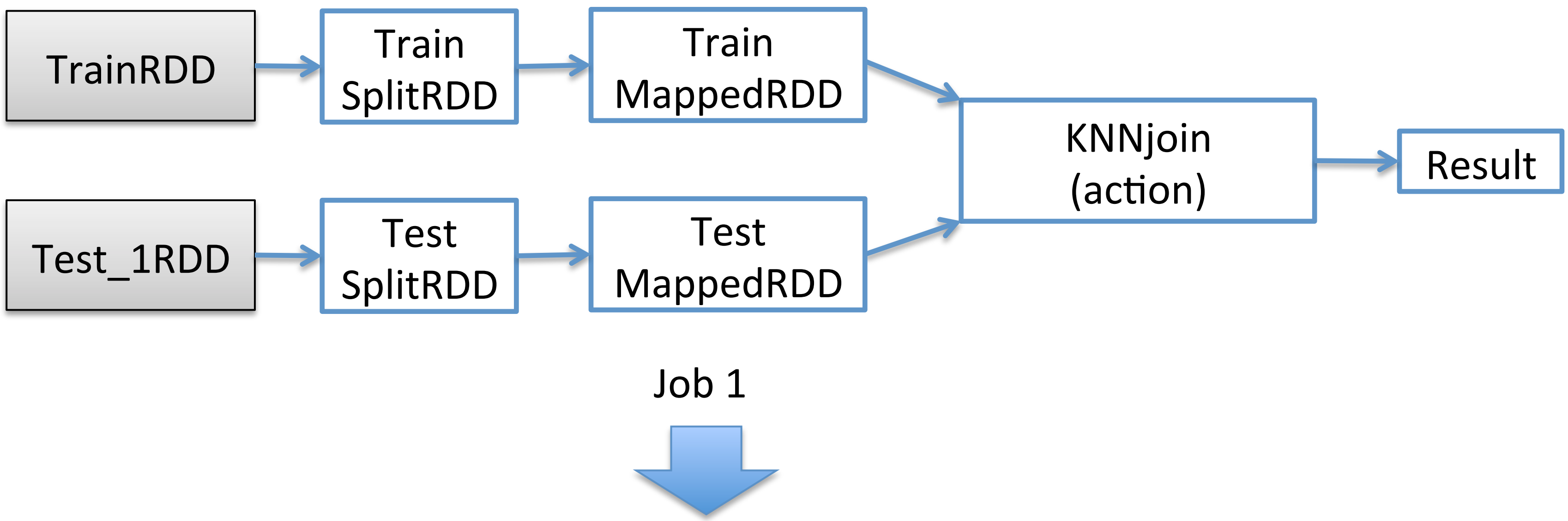
1. Data Flow Generation

- Goal: To understand RDD access order between jobs
- Solution: Preliminary run on small workloads to extract RDD access order
- Example: K Nearest Neighbors Classification
  - Classical ML classification algorithm
  - 1 *train* dataset, 3 *test* dataset

KNN Example: Job Execution



KNN Data Flow Graph



RDD_seq[1]={TrainRDD, Test_1RDD} Job 1

RDD_seq[2]={TrainRDD, Test_2RDD} Job 2

RDD_seq[3]={TrainRDD, Test_3RDD} Job 3

Goal: Understand the RDD access order between jobs.

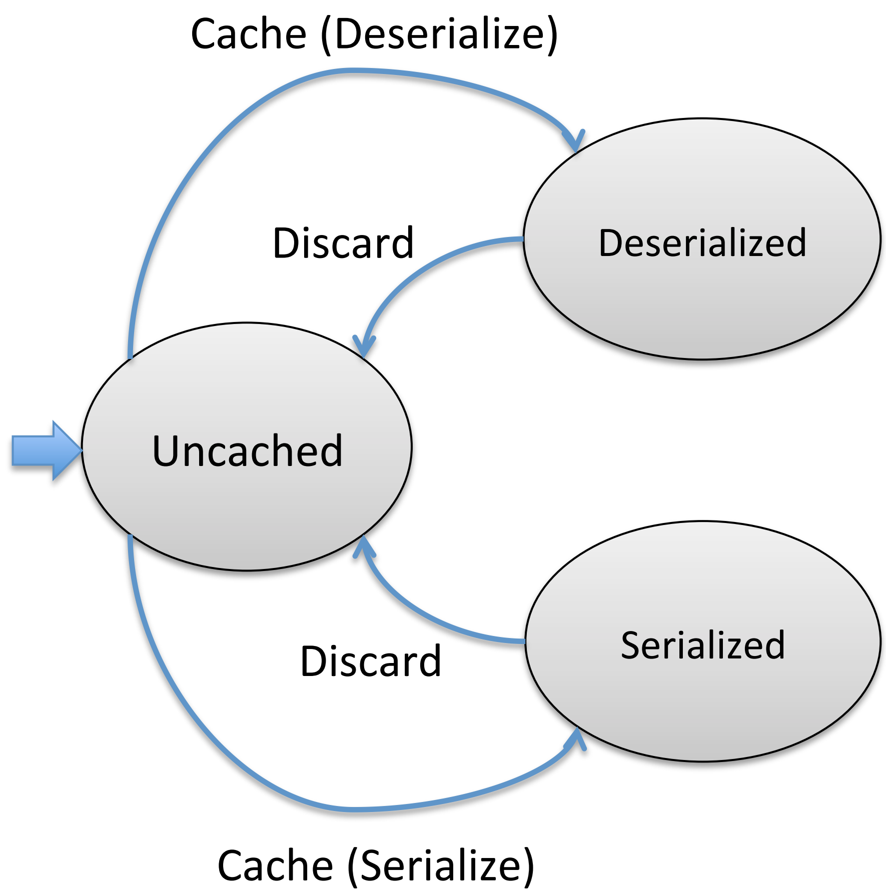
2. Adaptive Caching

- Goal: Fine-grained cache management at RDD partition level

- Solution: New cache level: *Adaptive*. It can move RDD partitions between cache levels at runtime

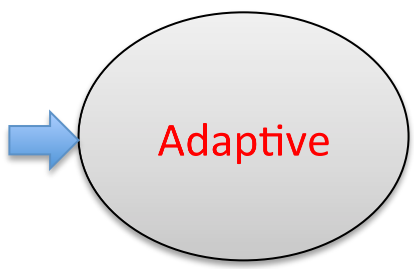
- Partition-level Operations: cache, discard and convert

Cache Operations in Spark

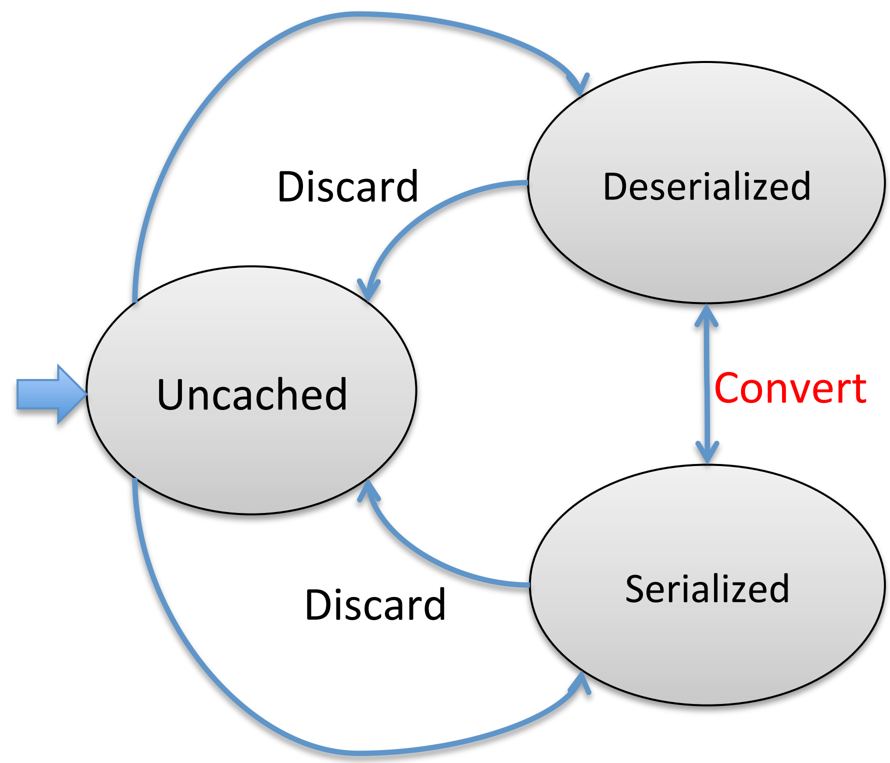


Caching Granularity: RDD

Adaptive Cache Operations in Neutrino



RDD



Caching Granularity: Partition

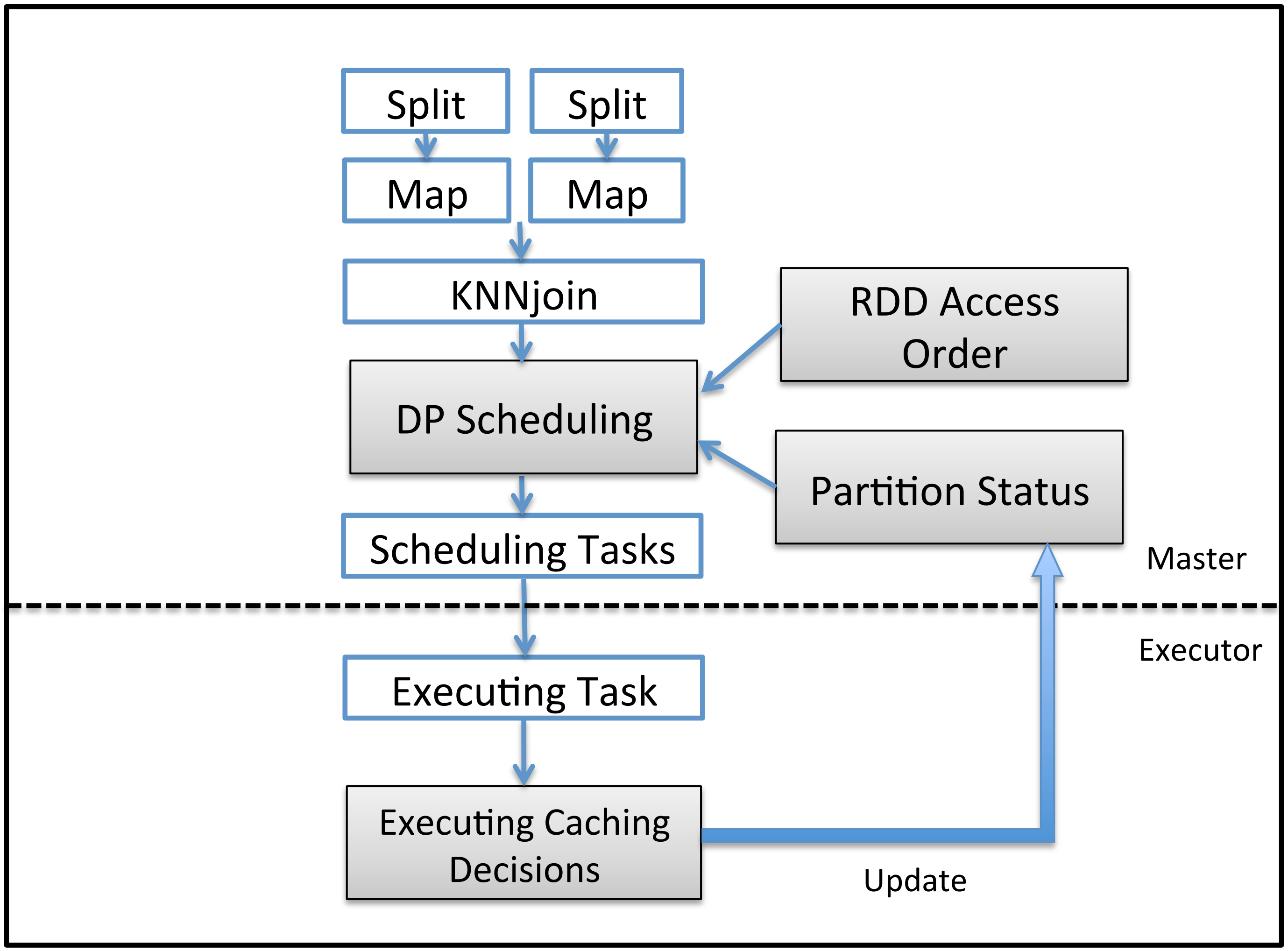
3. Dynamic Cache Scheduling

- Goal: Adapt to runtime changes for achieving optimal performance

- Solutions: Explore cache decisions on all partitions by dynamic programming each time before scheduling

- Dynamic Programming Model
  - Inputs: RDD access order, partition status
  - Output: Cache decision for each partition in the next job
  - Cost Model: Overall execution time

Execution of Dynamic Cache Scheduling



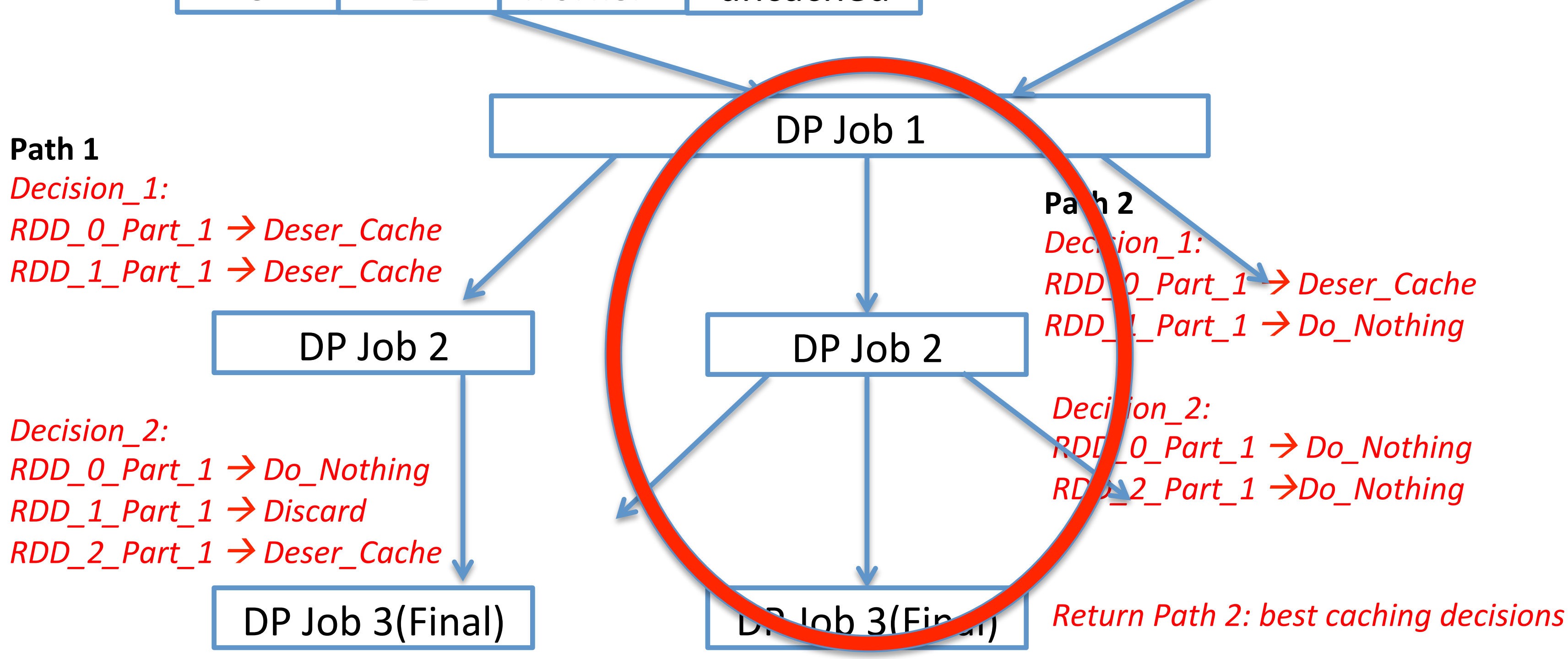
Dynamic Cache Scheduling: Caching Decisions

Partition Status Table

RDD#	Part#	Node	Status
0	1	worker1	uncached
1	1	worker1	uncached
2	1	worker1	uncached
3	1	worker1	uncached

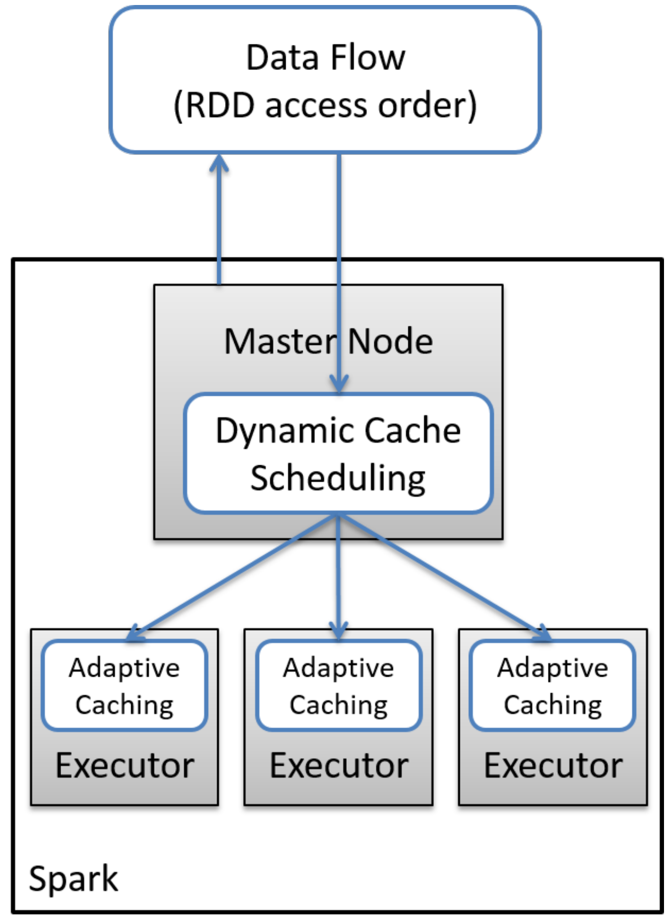
RDD Access Order

Job#	Part#
1	RDD0, RDD1
2	RDD0, RDD2
3	RDD0, RDD3



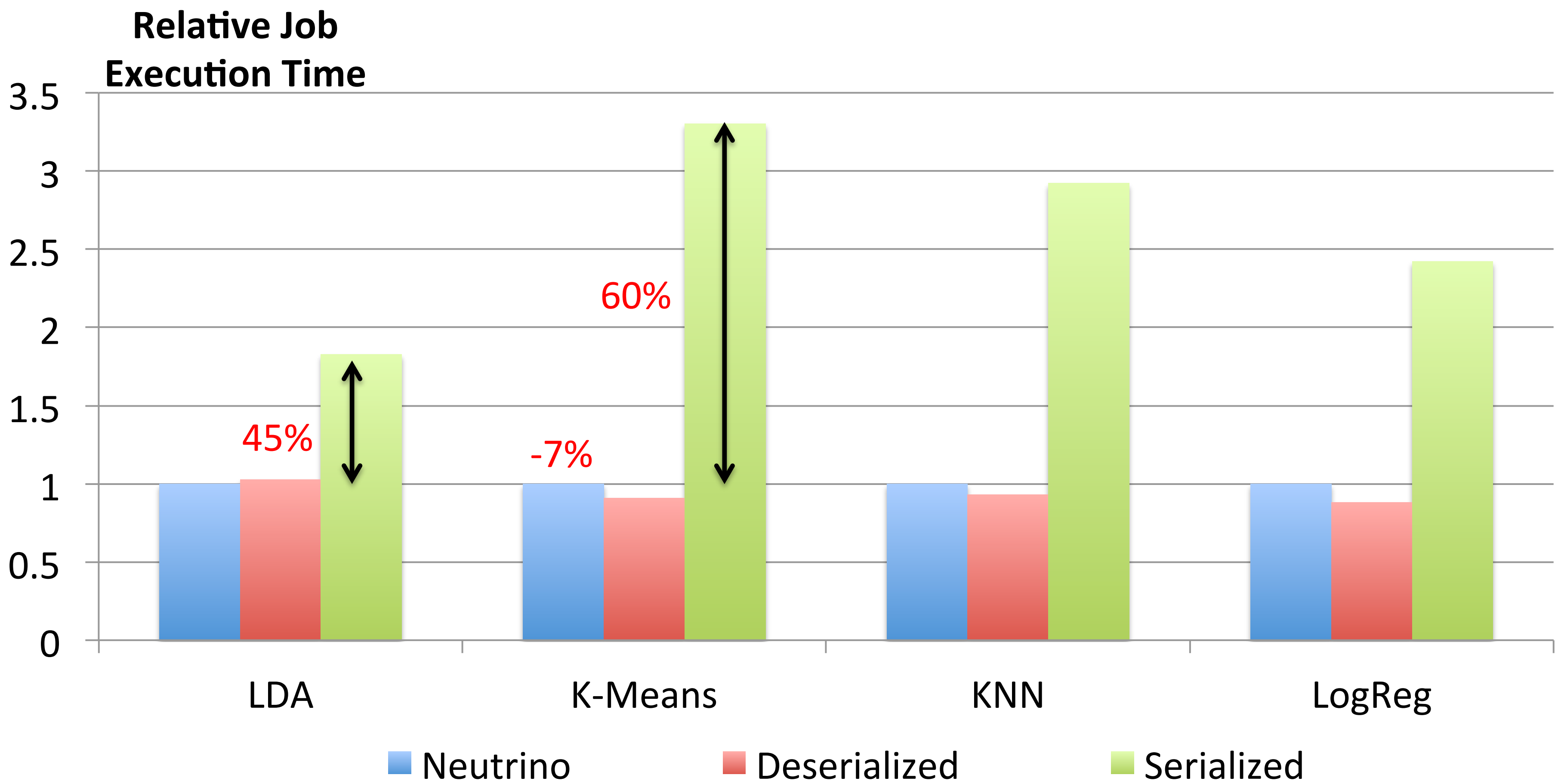
Evaluation

- Neutrino Implementation
  - Extension to Apache Spark
- Methodology
  - 6 nodes of 4 cores, 8GB memory each
  - Iterative machine learning workloads:
    - Classification: KNN, Logistic Regression
    - Clustering: K-Means
    - Inference: LDA
  - Systems Compared:
    - Neutrino with Adaptive Caching
    - Spark with Serialized and Deserialized Caching



Scenario 1: Abundant Memory

Deserialized data size < Cluster Memory

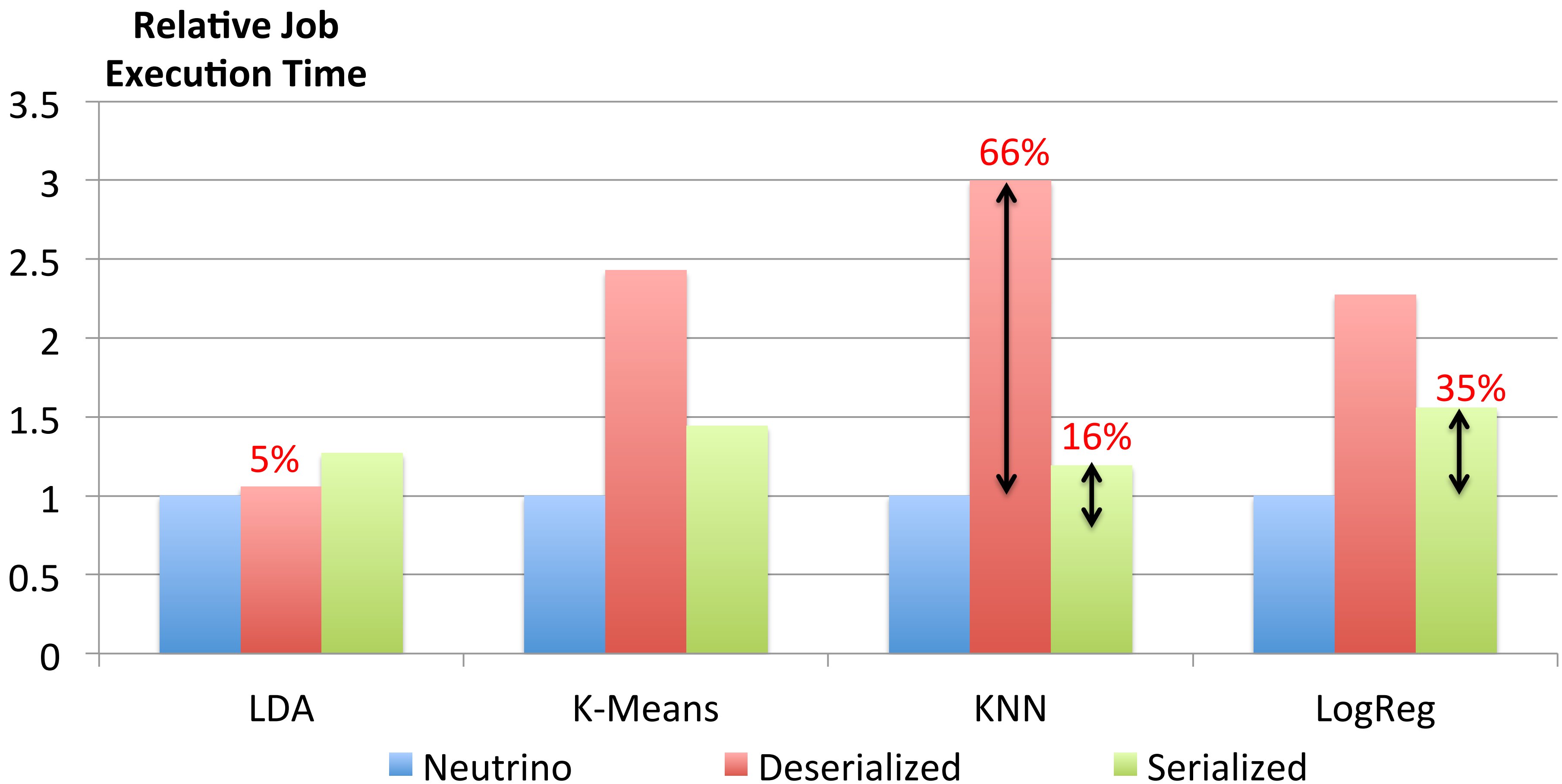


Neutrino has extra computation overhead for dynamic scheduling and additional operations

Neutrino deserialize all partitions and make efficient use of unused memory

Scenario 2: Sufficient Memory

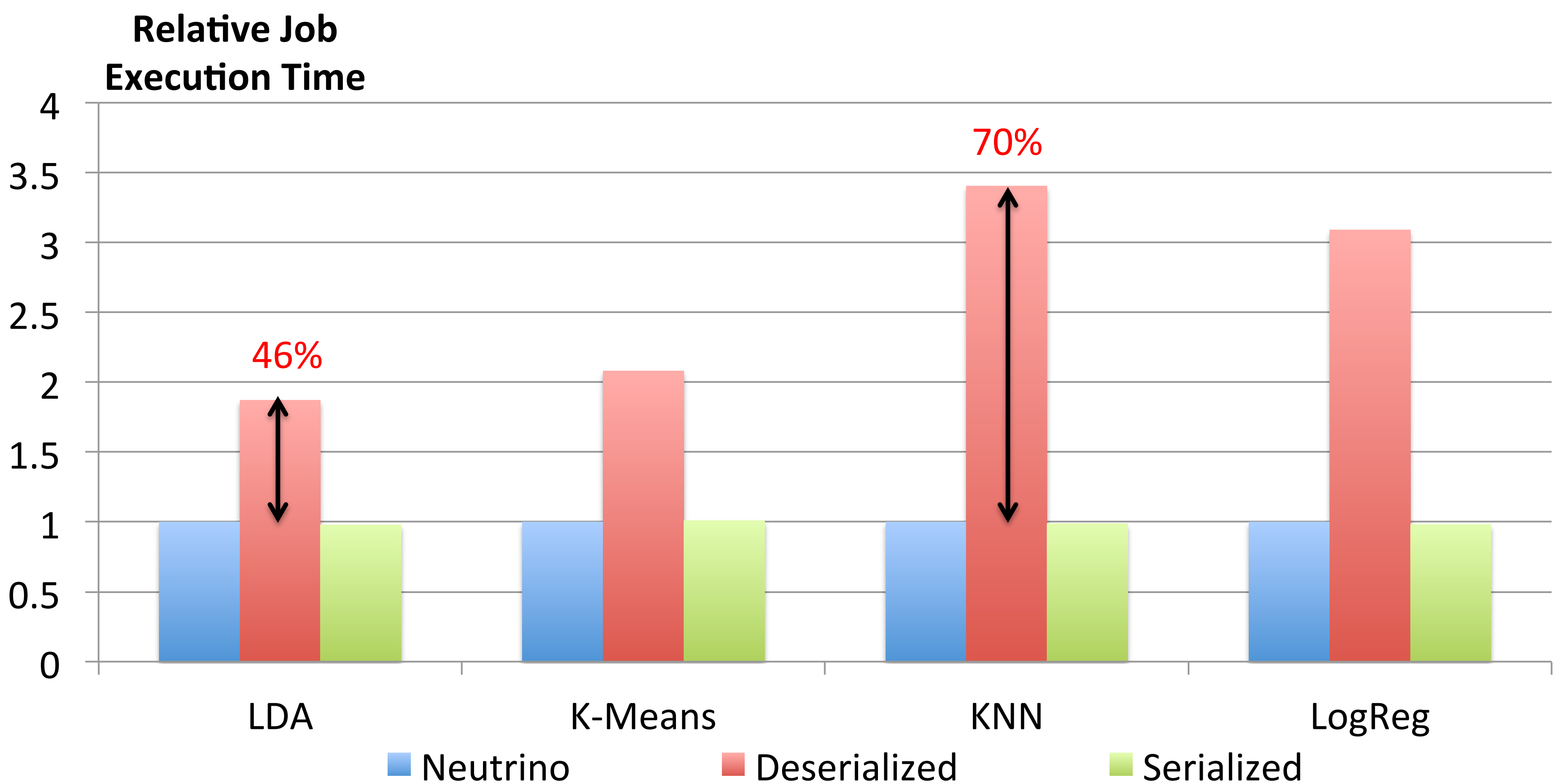
Deserialized data size > Cluster Memory



Deserialized level starts to hit disk and hence require re-computation from HDFS
Serialized level has extra overhead on deserialization.
Neutrino cache partially in deserialized level and partially in serialized level

Scenario 3: Just Enough Memory

Serialized data size = Cluster Memory



With more frequent cache misses occurred for Deserialized level

Conclusions

- Discrete Cache Levels for In-Memory Caching
  - Inefficient memory usage → not optimal performance
- Neutrino:
  - Partition level adaptive caching
  - Dataflow graph generation
  - Dynamic cache scheduling
- Neutrino improves average job execution time by up to 70% over native Spark caching

Thanks
Q&A

Neutrino: Revisiting Memory Caching for Iterative Data Analytics

Erci Xu*, Mohit Saxena, Lawrence Chiu
Ohio State University*
IBM Research Almaden