Neutrino: Revisiting Memory Caching for Iterative Data Analytics

Erci Xu*, Mohit Saxena, Lawrence Chiu
Ohio State University*
IBM Research Almaden
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 $10x \text{ – } 100x$ speedup
Spark for In-Memory Iterative Analytics

Apache Spark

Spark SQL
Spark Streaming
MLlib (machine learning)
GraphX (graph)
Spark RDD

RDD: Resilient Distributed Datasets

<table>
<thead>
<tr>
<th>Worker 1</th>
<th>Worker 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>block 1</td>
<td>block 2</td>
</tr>
<tr>
<td>block 3</td>
<td>block 4</td>
</tr>
<tr>
<td>partition 1</td>
<td>partition 2</td>
</tr>
<tr>
<td>partition 3</td>
<td>partition 4</td>
</tr>
</tbody>
</table>

In Memory

In HDFS
**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


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

![Graph showing the relationship between time and data size for different serialization methods. The graph includes lines for Deserialized, Serialized, and Optimal, with a goal for Neutrino indicated by an arrow.]
Problems: In-memory Caching for Iterative Data Analytics

1. Discrete Cache Levels


3. Not Adaptive to Runtime Changes
Problem 2: Manual Management

\[
\begin{align*}
\text{rdd}_1 &= \text{sc.textfile}(\text{HDFS://file1}) \\
\text{rdd}_2 &= \text{sc.textfile}(\text{HDFS://file2}) \\
\text{rdd}_1.\text{persist}(&\text{Cache-Level}) \\
\text{rdd}_2.\text{persist}(&\text{Cache-Level}) \\
\text{rdd}_1.\text{transformation}().\text{action}() \\
\text{rdd}_2.\text{transformation}().\text{action}() \\
\end{align*}
\]
Problems: In-memory Caching for Iterative Data Analytics

1. Discrete Cache Levels


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

- Less Manual Management
- Adaptive to Runtime Changes
- Fine-grained Cache Levels

Data Flow (RDD access order)

Master Node

Dynamic Cache Scheduling

Adaptive Caching
Executor

Adaptive Caching
Executor

Adaptive Caching
Executor

Spark
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

Spark Application

Job 1
- Train
- Split
- KNNjoin
- Executing Task
- Scheduling Tasks
- Map

Job 2
- Train
- Split
- KNNjoin
- Executing Task
- Scheduling Tasks
- Map

Job 3
- Train
- Split
- KNNjoin
- Executing Task
- Scheduling Tasks
- Map

Test_1
- Split
- KNNjoin
- Executing Task
- Scheduling Tasks
- Map

Test_2
- Split
- KNNjoin
- Executing Task
- Scheduling Tasks
- Map

Test_3
- Split
- KNNjoin
- Executing Task
- Scheduling Tasks
- Map

Master
Executor
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

- **Split**
- **Map**
- **KNNjoin**
- **DP Scheduling**
- **Scheduling Tasks**
- **Executing Task**
- **Executing Caching Decisions**

**RDD Access Order**

**Partition Status**

**Master**

**Executor**

Update
Dynamic Cache Scheduling: Caching Decisions

### Partition Status Table

<table>
<thead>
<tr>
<th>RDD#</th>
<th>Part#</th>
<th>Node</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>worker1</td>
<td>uncached</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>worker1</td>
<td>uncached</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>worker1</td>
<td>uncached</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>worker1</td>
<td>uncached</td>
</tr>
</tbody>
</table>

### RDD Access Order

<table>
<thead>
<tr>
<th>Job#</th>
<th>Part#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RDD0, RDD1</td>
</tr>
<tr>
<td>2</td>
<td>RDD0, RDD2</td>
</tr>
<tr>
<td>3</td>
<td>RDD0, RDD3</td>
</tr>
</tbody>
</table>

#### Path 1

- **Decision 1:** RDD_0_Part_1 → Deser_Cache
  
  RDD_1_Part_1 → Deser_Cache

#### Path 2

- **Decision 1:** RDD_0_Part_1 → Deser_Cache
  
  RDD_1_Part_1 → Do_Nothing

- **Decision 2:** RDD_0_Part_1 → Do_Nothing
  
  RDD_2_Part_1 → Do_Nothing

- **Return Path 2:** best caching decisions
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 deserializes all partitions and makes efficient use of unused memory.

Neutrino has extra computation overhead for dynamic scheduling and additional operations.

Relative Job Execution Time

- LDA: Neutrino increases execution time by 45%.
- K-Means: Neutrino increases execution time by 60%.
- KNN: Neutrino increases execution time by 25%.
- LogReg: Neutrino increases execution time by 15%.
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

Relative Job Execution Time

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