Neutrino: Revisiting Memory Caching for Iterative Data Analytics

Erci Xu*, Mohit Saxena†, and Lawrence Chiu†
*Ohio State University, †IBM Research Almaden
xu.1556@osu.edu,{msaxena,lchiu}@us.ibm.com

Abstract

In-memory analytics frameworks such as Apache Spark are rapidly gaining popularity as they provide order of magnitude performance speedup over disk-based systems for iterative workloads. For example, Spark uses the Resilient Distributed Dataset (RDD) abstraction to cache data in memory and iteratively compute on it in a distributed cluster.

In this paper, we make the case that existing abtractions such as RDD are *coarse-grained* and only allow *discrete cache levels* to be used for caching data. This results in inefficient memory utilization and lower than optimal performance. In addition, relying on the programmer to enforce caching decisions for an RDD makes it infeasible for the system to adapt to runtime changes. To overcome these challenges, we propose Neutrino that employs fine-grained memory caching of RDD partitions and adapts to the use of different in-memory cache levels based on runtime characteristics of the cluster. First, it extracts a data flow graph to capture the data access dependencies between RDDs across different stages of a Spark application without relying on cache enforcement decisions from the programmer. Second, it uses a dynamic-programming based algorithm to guide caching decisions across the cluster and adaptively convert or discard the RDD partitions from the different cache levels.

We have implemented a prototype of Neutrino as an extension to Spark and use four different machine-learning workloads for performance evaluation. Neutrino improves the average job execution time by up to 70% over the use of Spark's native memory cache levels.

1 Introduction

Traditional disk-based big data frameworks, for example Apache Hadoop, scale out the computation across multiple nodes in a distributed cluster. However, they fall short for the needs of the recently emerging iterative workloads such as clustering, inference and regression algorithms for machine learning [3]. These workloads require data to be cached in memory across different iterations.

Apache Spark [8, 5] is one of the most popular systems that is custom designed for serving iterative queries and use the Resilient Distributed Dataset (RDD) abstraction to cache data in memory. As a result, memory efficiency becomes critical to the overall performance [7, 4]. A Spark job is composed of Transformations (*e.g.* map, flatMap); and Actions (*e.g.* count, reduce). A RDD is typically created by loading data in an Action from HDFS wherein each HDFS block on disk represents a RDD partition in memory. The programmer can manually prescribe the system to cache the loaded RDD in a given cache level across different actions. Otherwise, the RDD is discarded from memory after the action is completed. Table 1 shows the different cache levels and their tradeoffs. These cache levels store all partitions of a RDD in serialized (compact) or deserialized (fast) memory cache levels. Alternatively, the RDD can be stored outside the Spark JVM heap (off heap) in memory and on disk.

In this paper, we find that coarse-grained cache management using RDD abstraction and discrete cache levels results in inefficient memory utilization and lower performance. First, all partitions of a RDD are stored in the same cache level across all worker nodes regardless of runtime characteristics such as the memory required for the RDD partition and free memory on a worker node. Second, the system relies on the application programmer to enforce caching decisions for a RDD using *persist* and *unpersist* interfaces exposed to the programmer. This makes it infeasible for moving a RDD or its partitions from one caching level to another.

In this paper, we address these challenges by designing Neutrino- a new distributed memory management system - implemented as an extension to Spark. Neutrino eliminates the need for the programmer to manually estimate the memory needs of serialized or deserialized cache levels; and enforce caching decisions based on cluster configuration. Instead, the programmer now simply uses a new *adaptive* cache level supported by Neutrino. Neutrino automatically extracts a *data flow graph* that tracks the access dependency of different RDDs used

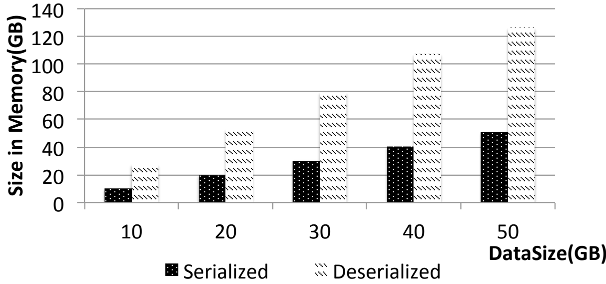

Figure 1: **Serialized and Deserialized Data Size in Spark Memory:** *Serialized data has a size in memory similar as of the dataset on disk, while size nearly triples for deserialized format.*

in different actions/stages of a Spark job.

Next, Neutrino uses this data flow graph and a *cost model* to compute the fine-grained cache level of each partition in a RDD. The cost model is based on a dynamic programming algorithm that tries to optimize the total job execution time by guiding adaptive caching decisions on each worker node to make the most efficient use of aggregate cluster memory. It accounts for the memory requirements for each partition in a certain cache level, free memory on each worker node, and data flow dependencies of RDDs. This is achieved by using two novel operations - *convert* and *discard* - implemented and used internally by Neutrino to move RDD partitions between different cache levels or remove them from memory respectively. As a result, a RDD could have a fraction of its partitions in deserialized or serialized cache levels, and remaining uncached on HDFS.

The remainder of the paper is structured as follows. Section 2 motivates Neutrino by demonstrating the tradeoffs for coarse-grained cache levels in Spark. Section 3 describes the design of Neutrino prototype; followed by Section 4 that compares the performance of Neutrino against Spark for four machine-learning iterative workloads.

2 Motivation

We now describe the different cache levels in Spark and their tradeoffs for memory usage and performance.

Deserialized Data in Memory. Serialization [1, 5] is an operation that converts in-memory object structures (*e.g.* string, graphs, etc.) into a stream of bytes that are written on disk or transferred over the network. Deserialization is the reverse operation. In Spark, a dataset read from HDFS is converted into a structured RDD that represents application-specific in-memory objects. For example, a text corpus of data read from HDFS can be structured into a RDD representing a vector of words for a clustering-based ML application.

Figure 1 shows the size of deserialized and serialized data in Spark memory when a text corpus of increasing size is loaded into memory. Deserialization can result in *nearly triple* the size as compared to serialized data. This is because deserialized data reconstructs the structural information (such as links in a graph, length of data types) within the memory object that is excluded in the serialized byte stream. In contrast, data in serialized format is almost identical in size as its size on disk. In general, even with the use of highly efficient serialization algorithms [1], the difference in memory footprint of data in serialized and deserialized formats can be up to an order of magnitude based on the data type [2, 5].

Level	Format	Location	Advantage
Mem_Deser	Deserialized	In-Heap	Fast
Mem_Ser	Serialized	In-Heap	Compact
Mem_Off	Serialized	Off-Heap	Less GC
Mem_Disk	Deserialized in Memory	Memory & Disk	Large

Table 1: **Popular Cache Levels in Spark**

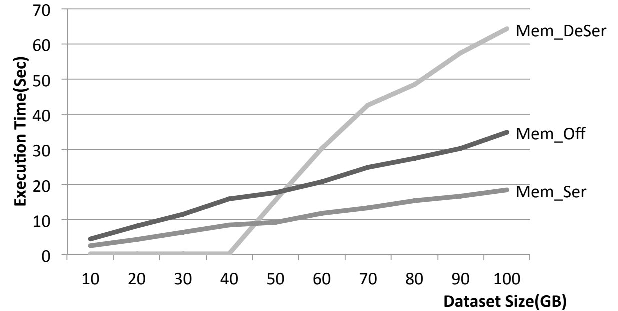

Figure 2: **Performance of Spark Cache Levels**

Spark Cache Levels. Table 1 shows the different cache levels provided by Spark. The application programmer has to select one of these levels for a RDD; otherwise the RDD is discarded from memory after a Spark Action. Each Spark cache level tradesoff between performance, memory usage, garbage collection overhead, and fault-tolerance. The most popular cache levels store the complete RDD in memory in one of the three cache levels: deserialized (*Mem_Deser*), or serialized (*Mem_Ser*) in Spark memory, or outside Spark's JVM heap in serialized format within a distributed memory filesystem (*Mem_Off*) [6].

Figure 2 shows the performance of a Spark job performing word-count on a dataset that ranges between 10 to 100 GB. Our cluster is composed of five worker nodes that provides a total of 100 GB Spark cache memory. The performance of *Mem_Deser* cache level is up to 24 times better than *Mem_Ser* and *Mem_Off* as there is no extra overhead for deserializing data during the job execution. However, it starts to drop for datasets greater than 40 GB in size because then the dataset grows beyond 100 GB of cache space in deserialized format. As a result, the RDD partitions that do not fit in memory need to be

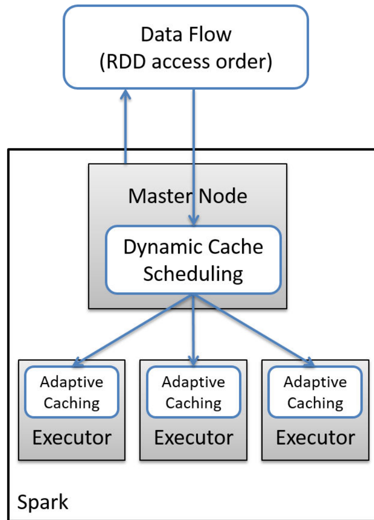

Figure 3: **Neutrino Architecture:** *Different components of the Neutrino architecture implemented as extensions to Spark.*

read from HDFS when accessed. In contrast, job execution times grow almost linearly for both *Mem_Ser* and *Mem_Off* as they store all data in memory in serialized format. *Mem_Ser* is faster than *Mem_Off* because it does not incur extra memory copies for moving data to memory outside Spark's heap. We also observe that *Mem_Ser* results in unused memory in the cluster for dataset sizes between 50-90 GB.

These results show that the programmer can not easily select the most suitable cache level as it is difficult to statically account for different data types and free memory at runtime in the cluster. Second, a coarse-grained selection of a cache level for a RDD may be inflexible to make the most effective use of free cluster memory. We now show how Neutrino addresses these challenges using *adaptive caching* by extracting a data flow graph to capture the access dependencies and using a runtime cost model to guide fine-grained conversion of RDD partitions between different cache levels.

3 System Design

In this section, we describe the design principles for Neutrino and how we have implemented a prototype for adaptive caching for managing memory in Spark.

Figure 3 shows the three main components of Neutrino architecture: *adaptive caching* implemented within the Spark Executors running on each worker node (Section 3.1), generation of the *data flow graph* (Section 3.2), and the runtime cost model implemented as a (*dynamic cache scheduling*) algorithm at the Spark Master to trigger convert and discard operations of adaptive caching on the different executors (Section 3.3).

3.1 Adaptive Caching in Neutrino

Neutrino provides programmers a new cache level *Adaptive* that enables to provide fine-grained cache management by moving RDD partitions between cache levels at runtime.

The Spark master directs the executors on different worker nodes to apply a coarse-grained cache level to all partitions of a RDD. The Spark block manager in each executor applies the same cache level directive to all partitions read from HDFS. We extend the partition structure to add an extra attribute that defines its cache level in Neutrino. As a result, when the Spark master distributes the executors in Neutrino to work on a given partition, it also assigns a cache level when loading that partition into memory.

At runtime, adaptive caching can move a partition across different cache levels or remove it from memory completely based on the inputs from the *DP Scheduling* algorithm. We enable the executor to perform this using three operations on a partition: *cache*, *discard*, and *convert*. The *cache* operation places the partition in the given cache level similar to Spark.

Spark identifies each partition by a global identifier and can asynchronously discard a partition. As a result, the memory is actually not freed immediately after issuing the asynchronous discard. We integrate this API in the Neutrino adaptive caching as it does not block the *discard* operation. The Neutrino adaptive caching verifies the completion of the asynchronous discard by checking the free capacity of the executor in the *cache* and *convert* operations. This ensures that the space for the discarded partition is not reused before it is actually discarded. As discard is asynchronous, its overhead is always overlapped by a cache or convert operation for the next job.

In Spark, a RDD partition can not be transformed between different cache levels, instead a RDD needs to be discarded from memory and then cached again in another level. Neutrino also provides a mechanism to *convert* RDD partitions between different cache levels. We currently support conversion from deserialized to serialized cache levels and vice versa. The conversion requires extra computation to (de)-serialize a RDD partition. As a result, we perform conversion only on-demand when a RDD partition is used and cached for the executing job. A convert from deserialized to serialized cache level frees up space to be reused for loading more partitions in memory or converting another partition to the deserialized level. A convert from serialized to deserialized cache level improves performance for computing over the partition by making use of the un-utilized memory in the cluster.

3.2 Data Flow Generation

As shown in Figure 3, Neutrino first extracts the data flow graph which is the order in which the different

```
// Load the data from storage
val testing = sc.textFile("/data/mllib/knn_testing")
val training = sc.textFile("/data/mllib/knn_training")

//Enable Neutrino adaptive caching level
sc.neutrino(true)

val testVec = testing.map(s => Vectors.dense(s.split(' ').map(_.toDouble)))
val trainVec = training.map(s => Vectors.dense(s.split(' ').map(_.toDouble)))

//run KNN algorithm
val clusters = KNNjoin(trainVec, testVec, iterations, sc)
```

Figure 4: **K-Nearest Neighbor (KNN) Application**

RDDs are accessed in a Spark job. We can extract a data flow graph using different approaches, *e.g.* static analysis of the program, or by actually executing the program on a smaller dataset. We use the second approach to extract the RDD access order by executing the program on a smaller dataset.

A Spark application is split into different jobs and the data flow graph captures the access order of RDDs in different jobs' execution. We retrieve the access order by instrumenting the *getOrCompute* API in the master node. This API is used whenever a RDD is loaded from HDFS or computed using another RDD. The access order is retrieved as a table of key-value pairs where each key represents a job id and the value is the list of RDDs that are used in this job. The order of RDD access across jobs is dependent on how Transformations and Actions are applied in a Spark program. For example, in the K-Nearest Neighbor (KNN) Application (see Figure 4), the *testing* and *training* RDDs are loaded from HDFS datasets, and *testVec* and *trainVec* vector RDDs are created from them in the map Transformations. KNNjoin represents a series of Actions applied iteratively on these testVec and trainVec RDDs. Each iteration represents an action or stage. The dataflow graph generated by Neutrino represents all four RDDs used in the first stage, and following iterative stages only include the vectorized RDDs. The dataflow graph captures the RDD access order, hence it can be generated on a relatively small dataset and used for different iterations or job executions.

3.3 Dynamic Cache Scheduling at Runtime

Neutrino extends the Spark master to maintain runtime information and guide adaptive caching decisions at the executors. We maintain a *partitionStatus* map at the master node that records the status of each partition. It includes the blockId of the partition as the key, and a value that is a combination of the serialized/deserialized sizes of the partition, location of the partition, current cache level, and the pending action on the partition. Each record in partitionStatus map is 48 bytes, and it takes no more than 512 MB of memory space for a very large petabyte dataset.

Figure 5 shows how the Neutrino master generates

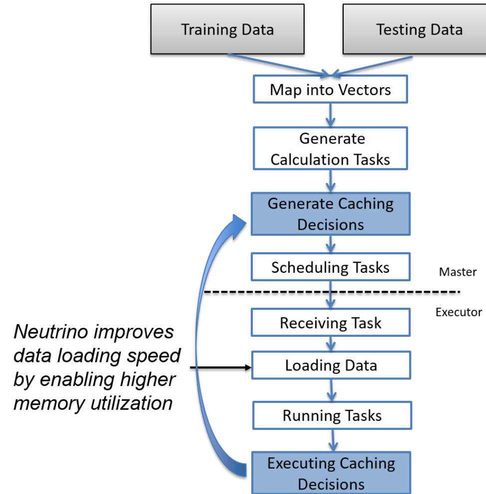

Figure 5: **Neutrino KNN Execution Flow**

```
//Dynamic Cache Scheduling in Neutrino
dy_sched(rdd_seq[i],pMap) {
    if (impossible cases)   return {INT_MAX, Null}
    if (execution finished) return {0, Null}
    if (opt[i,pMap] exists) return  opt[i,pMap]
    MinTime = INT_MAX
    for(op in available_operations){
        tempMap         = pMap
        op_Time         = Accessing(pMap) + Executing(op, tempMap)
        next_stage_time = dy_sched(rdd_seq[i+1],tempMap)
        cur_stage_time  = next_stage_time->time + op_Time
        if (cur_stage_time < MinTime) {
                opt[i, pMap] = {cur_stage_time, op}
                MinTime = cur_stage_time->time
            }
    }
    return opt[i,pMap];
}
```

Figure 6: **Dynamic Cache Scheduling Algorithm**

the caching decisions for a Spark K-Nearest Neighbor application (see Figure 4 with two RDDs created from the training and testing datasets. The two datasets are mapped into vectors and the compute tasks are created based on the location of the dataset partitions. Before the tasks are scheduled on the different worker nodes, Neutrino master uses the *Dynamic Cache scheduling* algorithm to generate the decisions for adaptive caching to trigger the cache, convert or discard operations for selecting the cache level of each RDD partition. For example, without the Dynamic Cache Scheduling generating caching decisions in Figure 5, the vectorized datasets as RDDs will be cached in a cache level, which might overflow the memory if deserialized or leave memory underutilized if serialized.

Figure 6 shows the Dynamic Cache Scheduling algorithm. It computes the minimum time to execute a Spark application for a given data flow graph (rdd_seq) and partition status map (pMap). rdd_seq[i] is the list of RDDs

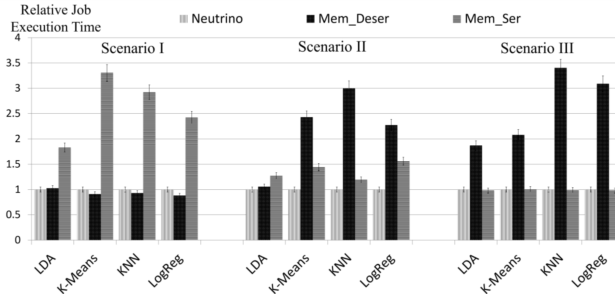

Figure 7: **System Comparison: Neutrino vs. Spark**

used in the ith stage/action of the data flow graph. pMap represents the cache level and location of each partition in the system. The dy_sched algorithm uses dynamic programming to compute the minimum execution time at the ith stage by exploring all possible combinations of *available_operations* (convert, cache, discard) applied to the different partitions at stages *i* and later. It does not explore infeasible combinations such as caching with insufficient memory or convert non-cached partitions. The functions Accessing and Executing compute the time to read data from the given cache level (partition map) in the ith stage, and the time to apply the selected operations at the end of ith stage respectively (also shown as Executing Cache Decisions in Figure 5). To limit the number of possible combinations to explore in each stage and use memory fairly across different executors in the cluster, we apply the cache operations at the granularity of a configurable fraction of a RDD. For example, we can apply a cache operation such as convert 25% of the partitions of a RDD from deserialized to serialized cache level to save memory space uniformly across all worker nodes.

4 Evaluation

We compare Neutrino against Spark's native caching levels: *Mem_Ser* and *Mem_Deser*. The experiments were performed on a cluster of six worker nodes, each equipped with Intel i5-2400 3.1 GHz CPU, one 500 GB disk, 8 GB DRAM out of which Spark uses 6 GB memory. We use HDFS v2.6.3 with a block size of 128 MB and Spark v1.5.0 with four MLlib workloads: K-Means, KNN, Latent Dirichlet Allocation (LDA), and Logistic Regression. These workloads cover the three different popular categories of ML workloads including inference, nearest neighbor, and regression; and iteratively query one or multiple RDDs.

Figure 7 shows the average job execution time for Spark *Mem_Ser* and *Mem_Deser* cache levels relative to Neutrino. We evaluate three scenarios with different dataset size: (1) deserialized dataset size $<$ cluster memory, (2) deserialized dataset size = cluster memory, and (3) serialized dataset size = cluster memory.

Neutrino outperforms Spark cache levels in most scenarios for all three workloads. In Scenario 1, Neutrino is about 45-60% faster than Spark *Mem_Ser* because it deserializes all partitions and makes more efficient use of unused cluster memory than *Mem_Ser*. Neutrino is up to 7% slower than Spark *Mem_Deser* because of the extra computation overhead for dynamic cache scheduling and additional cache or convert operations. The discard operation is asynchronous and its overhead gets overlapped with the job computation time. In Scenario 2, Neutrino is about 16-35% and 5-66% faster than Spark *Mem_Ser* and *Mem_Deser* cache levels respectively. This is because Spark *Mem_Deser* starts missing the memory cache and has to recompute partitions from HDFS. In contrast, Neutrino always hits in the memory and keeps RDD partitions in both formats. Spark *Mem_Ser* has an overhead for deserializing data during job execution. In Scenario 3, Neutrino serializes all RDD partitions in memory and is almost identical in performance to Spark *Mem_Ser*. The performance gap between Neutrino and Spark *Mem_Deser* increases to 46-70% because of more frequent cache misses in Spark *Mem_Deser*.

5 Conclusions and Future Directions

In this paper, we take a fresh look on a very pressing area of memory caching for iterative data analytics. We make a case for fine-grained caching with a new dynamic cache scheduling algorithm. The results look promising and we plan to build upon current work. In future, we will continue to work in two main directions: (1) improve data flow generation process to track dependencies for iterative workloads that can result in non-deterministic execution flows, and (2) improve the pruning technique to reduce the exploration space and provide stronger fairness guarantees for the dynamic cache scheduling algorithm.

References

[1] Kyro: Java serialization and cloning. `https://github.com/EsotericSoftware/kryo`, 2016.

[2] Project Tungsten: Bringing spark closer to bare metal. `https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html`, 2016.

[3] Spark MLlib. `http://spark.apache.org/docs/latest/mllib-guide.html`, 2016.

[4] Ananthanarayanan, G., Ghodsi, A., Wang, A., Borthakur, D., Kandula, S., Shenker, S., and Stoica, I. Pacman: Coordinated memory caching for parallel jobs. In *NSDI* (2012).

[5] Armbrust, M., Das, T., Davidson, A., Ghodsi, A., Or, A., Rosen, J., Stoica, I., Wendell, P., Xin, R., and Zaharia, M. Scaling spark in the real world: Performance and usability. *PVLDB* (2015).

[6] Li, H., Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SOCC* (2014).

[7] Pu, Q., Li, H., Zaharia, M., Ghodsi, A., and Stoica, I. Fairride: Near-optimal, fair cache sharing. In *NSDI* (2016).

[8] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (2012).