Questions from the paper "RDD: A Fault-Tolerant Abstraction for In-Memory Cluster Computing "

Paper can be downloaded from:
https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

1. Introduction

a. Existing cluster computing frameworks lack abstractions for leveraging distributed shared memory. This makes them inefficient for two classes of applications. Name these two classes and explain them briefly.

Iterative algorithms and iterative data mining. Those applications reuse intermediate results across multiple computations. In interactive data mining, a user runs multiple adhoc queries on the same subset of the data.

b. What are some of the design considerations for RDDs? How is the problem of efficiently providing fault tolerance solved in case of RDDs?

If defining a programming interface that can provide fault tolerance efficiently, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. This approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster netword, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead. Also, the design of RDD also including the parallel applications.

In contrast to these systems, RDDs provide an interface based on coarse-grained transformations that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its lineage) rather than the actual data. If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute just that partition. Thus, lost data can be recovered, often quite quickly, without requiring costly replication.

Although an interface based on coarse-grained transformations may at first seem limited, RDDs are a good fit for many parallel applications, because these applications naturally apply the same operation to multiple data items. Indeed, we show that RDDs can efficiently express many cluster programming models that have so far been proposed as separate systems, including MapReduce, SQL and new applications that these systems do not capture, like interactive data mining. The ability of RDDs to accommodate computing needs that were previously met only by introducing new frameworks is the most credible evidence of the power of the RDD abstraction.

2. RDDs

2.1 RDD Abstraction

a. What are the two deterministic ways in which RDDs can be created?

(1) Data in stable storage
(2) Other RDDs

b. Which properties of RDDs can users control?

Persistence and partitioning. Users can indicate which RDDs they will reuse and choose a storage strategy for them. They can also ask that an RDD's elements be partitioned across machines based on a key in each record. This is useful for placement optimizations, such as ensuring that two datasets that will be joined together are hash-partitioned in the same way.

Spark Programming Interface

a. Understand actions in Spark. What do they return? Why do you think Spark computes RDDs lazily the first time they are used in an action?

Programmers start by defining one or more RDDs through transformations on data in stable storage (e.g., map and filter). They can then use these RDDs in actions, which are operations that return a value to the application or export data to a storage system. Examples of actions include count (which returns the number of elements in the dataset), collect (which returns the elements themselves), and save (which outputs the dataset to a storage system). Like DryadLINQ, Spark computes RDDs lazily the first time they are used in an action, so that it can pipeline transformations.

b. Look at the example 2.2.1 and understand the code. What method can you call on an RDD so that it stays in memory even after an action command has been issued on it?

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

After the first action involving errors runs, Spark will store the partitions of errors in memory, greatly speeding up subsequent computations on it.

2.2 Advantages of the RDD model

a. What are the advantages of RDD model over traditional DSM model

RDDs can only be created through coarse-grained tranformations, while DSM allows reads and writes to each memory location. This restricts RDDs to applications that perform bulk writes, but allows for more efficient fault tolerance.

RDDs' immutable nature lets a system mitigate slow nodes by running backup copies of slow tasks as in MapReduce. Backup tasks would be hard to implement with DSM, as the two copies of a task would access the same memory locations and interfere with each other's updates.

RDDs provide two other benefits over DSM. First in bulk operations on RDDs, a runtime can schedule tasks based on data locality to improve performance. Second, RDDs degrade gracefully when there is not enough memory to store them, as long as they are only being used in scan-based operations. Partitions that do not fit in RAM can be stored on disk and will provide similar performance to current data-parallel systems.

3. Spark Programming Interface

-- No written answers required for this section, but be sure to understand the concepts and examples carefully –
-- Understand the transformations and actions presented in Table 2 carefully –
-- You can also consult the Spark programming guide:
http://spark.apache.org/docs/latest/programming-guide.html

4. Representing RDDs

a. What are the 5 pieces of information that the RDD common interface exposes? Explain them briefly.

In a nutshell, we propose representing each RDD through a common interface that exposes five pieces of information: a set of partitions, which are atomic pieces of the dataset; a set of dependencies on parent RDDs; a function for computing the dataset based on its parents; and metadata about its partitioning scheme and data placement. For example, an RDD representing an HDFS file has a partition for each block of the file and knows which machines each block is on. Meanwhile, the result of a map on this RDD has the same partitions, but applies the map function to the parent's data when computing its elements.

What is the difference between *narrow* and *wide* dependencies for RDDs?

narrow dependencies: each partition of the parent RDD is used by at most one partition of the child RDD
wide dependencies: multiple child partitions may depend on it.

b. Which type of dependency needs data from multiple nodes? Which type of dependency makes recovery from failure easier?

Wilde dependencies need data from multiple nodes. Narrow dependencies make recovery from failure easier.

c. What would the partitions() operation would return for following cases:

- **HDFS files**
  The input RDDs in our samples have been files in HDFS. For these RDDs, partitions returns one partition for each block of the file (with the block's offset stored in each Partition object), preferredLocations gives the nodes the block is on, and iterator reads the block

- **Instance of MappedRDD object**
  Calling map on any RDD returns a MappedRDD object. This object has the same partitions and preferred locations as its parent, but applies the function passed to map to the parent's records in its iterator method.
- **An RDD created by union method**
  Calling union on two RDDs returns an RDD whose partitions are the union of those of the parents. Each child partition is computed through a narrow dependency on the corresponding parent
- **An RDD created by sample method**
  Sampling is similar to mapping, except that the RDD stores a random number generator seed for each partition to deterministically sample parent records.
- **An RDD created by join method**
  Joining two RDDs may lead to either two narrow dependencies (if they are both hash/range partitioned with the same partitioner), two wide dependencies, or a mix (if one parent has a partitioner and one does not). In either case, the output RDD has a partitioner (either one inherited from the parents or a default hash partitioner).

5. Implementation

5.1 Job Scheduling

a. The scheduler tries to build stages by combining transformation that have what type of dependencies?

Each stage contains as many pipelined transformations with narrow dependencies as possible.

b. What do the boundary of the stages represent?

The boundaries of the stages are the shuffle operations required for wide dependencies, or any already computed partitions that can shortcircuit the computation of a parent RDD. The scheduler then launches tasks to compute missing partitions from each stage until it has computed the target RDD.

c. How does Spark use the concept of data locality?

Our scheduler assigns tasks to machines based on data locality using delay scheduling.If a task needs to process a partition that is available in memory on a node, we send it to that node. Otherwise, if a task processes a partition for which the containing RDD provides preferred location, we send it to those.

5.3 Memory Management

a. What 3 options does Java provide for storage of persistent RDDs? Which one provides fastest performance and which one is best suited for large RDDs

Spark provides three options for storage of persistent RDDs: in-memory storage as deserialized Java objects, in-memory storage as serialized data, and on-disk storage. The first option provides the fastest performance, because the Java VM can access each RDD element natively. The third

option is useful for RDDs that are too large to keep in RAM but costly to recomputed on each use.

6. Evaluation

-- Read the sections and performance improvement metrics –
-- No written answers needed –

7.Discussion

-- Read the sections and understand how various programming models can be expressed in Spark --
-- No written answered needed --