**Reading Assignment 3 – Aparna Pavithran (axp161730)**

**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.**

**Answer: -**

The above scenario makes distributed shared memory inefficient for an important class of emerging applications. Those are reuse intermediate results across multiple computations and interactive data mining. Data reuse is common in many iterative machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. In interactive data mining, where 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?**

**Answer: -**

RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

RDDs provide an interface based on coarse-grained transformations, which apply the same operation to many data items. RDDs 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 re-compute 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, DryadLINQ, SQL, Pregel and HaLoop, as well as 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, we believe, the most credible evidence of the power of the RDD abstraction.

RDDs do not need to be materialized at all times. Instead, an RDD has enough information about how it was derived from other datasets (its lineage) to compute its partitions from data in stable storage.

**2. RDDs**

**2.1 RDD Abstraction**

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

**Answer: -**

RDDs can only be created through deterministic operations on either (1) data in stable storage or (2) other RDDs. These operations are called as transformations to differentiate them from other operations on RDDs. Examples of transformations include map, filter, and join.

**b. Which properties of RDDs can users control?**

**Answer: -**

Users can control two other aspects of RDDs: persistence and partitioning. Users can indicate which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage). 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.

**2.2 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?**

**Answer: -**

Spark exposes RDDs through a language-integrated API similar to DryadLINQ and FlumeJava, where each dataset is represented as an object and transformations are invoked using methods on these objects. 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. In addition, programmers can call a persist method to indicate which RDDs they want to reuse in future operations. Spark keeps persistent RDDs in memory by default, but it can spill them to disk if there is not enough RAM. Users can also request other persistence strategies, such as storing the RDD only on disk or replicating it across machines, through flags to persist. Finally, users can set a persistence priority on each RDD to specify which in-memory data should spill to disk first.

**b. Look at the example 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?**

**Answer: -**

persist() :- it asks RDD to persist in memory so that it can be shared across queries.

**2.3 Advantages of the RDD model**

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

**Answer: -**

In DSM systems, applications read and write to arbitrary locations in a global address space. Note that under this definition, we include not only traditional shared memory systems, but also other systems where applications make fine grained writes to shared state, including Piccolo, which provides a shared DHT, and distributed databases. DSM is a very general abstraction, but this generality makes it harder to implement in an efficient and fault tolerant manner on commodity clusters. The main difference between RDDs and DSM is that RDDs can only be created (“written”) through coarse grained transformations, 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. In particular, RDDs do not need to incur the overhead of check pointing, as they can be recovered using lineage. Furthermore, only the lost partitions of an RDD need to be recomputed upon failure, and they can be recomputed in parallel on different nodes, without having to roll back the whole program.

A second benefit of RDDs is that their immutable nature lets a system mitigate slow nodes (stragglers) 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. Finally, 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.

Comparison of RDDs with distributed shared memory.

|  |  |  |
| --- | --- | --- |
| Aspect | RDDs | DSM |
| Reads | Coarse- or fine-grained | Fine-grained |
| Writes | Coarse-grained | Fine-grained |
| Consistency | Trivial (immutable) | Up to app / runtime |
| Fault recovery | Fine-grained and low overhead using lineage | Requires checkpoints and program rollback |
| Straggler mitigation | Possible using backup tasks | Difficult |
| Work placement | Automatic based on data locality | Up to app (runtimes aim for transparency) |
| Behavior if not enough RAM | Similar to existing data flow systems | Poor performance (swapping?) |

**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.**

**Answer: -**

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.

Interface used to represent RDDs in Spark.

|  |  |
| --- | --- |
| Operation | Meaning |
| partitions() | Return a list of Partition objects |
| preferredLocations(p) | List nodes where partition p can be accessed faster due to data locality |
| dependencies() | Return a list of dependencies |
| iterator(p, parentIters) | Compute the elements of partition p given iterators for its parent partitions |
| partitioner() | Return metadata specifying whether the RDD is hash/range partitioned |

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

**Answer: -**

Narrow dependencies, where each partition of the parent RDD is used by at most one partition of the child RDD, wide dependencies, where multiple child partitions may depend on it. For example, map leads to a narrow dependency, while join leads to wide dependencies (unless the parents are hash-partitioned).

This distinction is useful for two reasons. First, narrow dependencies allow for pipelined execution on one cluster node, which can compute all the parent partitions. For example, one can apply a map followed by a filter on an element-by-element basis. In contrast, wide dependencies require data from all parent partitions to be available and to be shuffled across the nodes using a MapReduce like operation. Second, recovery after a node failure is more efficient with a narrow dependency, as only the lost parent partitions need to be recomputed, and they can be recomputed in parallel on different nodes. In contrast, in a lineage graph with wide dependencies, a single failed node might cause the loss of some partition from all the ancestors of an RDD, requiring a complete re-execution.

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

**Answer: -**

In wide dependencies multiple child partitions may depend on it. Wide dependency makes recovery from failure easier.

Narrow dependencies allow for pipelined execution on one cluster node, which can compute all the parent partitions. For example, one can apply a map followed by a filter on an element-by-element basis. In contrast, wide dependencies require data from all parent partitions to be available and to be shuffled across the nodes using a MapReduce like operation.

Recovery after a node failure is more efficient with a narrow dependency, as only the lost parent partitions need to be recomputed, and they can be recomputed in parallel on different nodes. In contrast, in a lineage graph with wide dependencies, a single failed node might cause the loss of some partition from all the ancestors of an RDD, requiring a complete re-execution.

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

**- HDFS files**

**- Instance of MappedRDD object**

**- An RDD created by union method**

**- An RDD created by sample method**

**- An RDD created by join method**

**Answer: -**

**- HDFS files :-** partitions returns one partition for each block of the file with the block’s offset stored in each Partition object.

**- 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 is 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 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?**

**Answer: -**

Whenever a user runs an action (e.g., count or save) on an RDD, the scheduler examines that RDD’s lineage graph to build a DAG of stages to execute. Each stage contains as many pipelined transformations with narrow dependencies as possible.

**b. What do the boundary of the stages represent?**

**Answer: -**

The boundaries of the stages are the shuffle operations required for wide dependencies, or any already computed partitions that can short circuit the computation of a parent RDD.

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

**Answer: -**

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 locations (e.g., an HDFS file), 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**

**Answer: -**

Spark provides three options for storage of persistent RDDs: in-memory storage as desterilized 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 second option lets users choose a more memory-efficient representation than Java object graphs when space is limited, at the cost of lower performance. 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 --**