Data-Parallel Programming with Apache Spark

Distribute data over multiple machine to process it in parallel. Concerns:

- *Partial failure*: what if one or more nodes fail?
- *Latency*: certain operations have higher latency due to network communication

Distributed Data-Parallel

Data are partitioned between machines, network in between, operated upon in parallel.

Apache Spark

Spark implements **Resilient Distributed Datasets (RDDs)**. They look like ***imutable*** sequential or parallel Scala collections. We can use `map, flatMap, filter, reduce, fold, aggregate`

Transformation and Actions

- **Transformations**: return new RDDs as result. They are **lazy**, their result is not immediately computed (wait for an action)
- **Actions**: compute a result based on an RDD and either return it or save it to external storage

Lazy -> RDDs are computed **the first time they are used in an action**.

Topology

A Spark application uses a set of processes on a cluster. All these processes are coordinated by the **driver program**. The processes on the cluster that run computations and store data are called **executors**.

The driver program creates a `SparkContext` which connects to a cluster manager. Spark acquires executors, the driver sends the application code to the executors, the context sends **task** to run.

Transformations

`sample`, `union` (duplicates remain), `intersection` (duplicates remain), `distinct`, `coalesce` (reduce number of partitions to argument), `repartition` (reshuffle data randomly)

Actions

`collect` (return array to driver program), `count`, `foreach`, `saveAsTextFile`

Pair RDDs

RDDs of couples (Key, Value). provide:

- `def groupByKey(): RDD[(K, Iterable[V])]` ATTENTION SHUFFLING
- `def reduceByKey(f: (V, V) => V): RDD[(K, V)]` better than groupbykey as it reduces before sending into network -> less latency and shuffling
- `def mapValues[U](f: V => U): RDD[(K, U)]`
- `def countByKey(): Map[K, Long]`

Create PairRDD usually using `map`

Joins

- `def join[W](other: RDD[(K,V)]): RDD[(K, (V, W))]` only the keys present in both RDDs are present in the result
- `def leftOuterJoin[W](other: RDD[(K, V)]): RDD[(K, (V, Option[W]))]` if present left will be in result, matched with None if not in right.
- `def rightOuterJoin[W](other: RDD[(K, V)]): RDD[(K, (Option[V], W))]` if present right will be in result, matched with None if not in left.

Remember parallelizable operation

Associativity!!!

- `def fold(z: A)(f: (A, A) => A): A` (not left or right)
- `def aggregate[B](z: => B)(seqop: (B, A) => B, combop: (B, B) => B): B`
- `reduce` (not left and right)

As we cannot use foldLeft or foldRight, when we need these functions, we have to use aggregate.

Shuffling (shit happens)

`groupByKey` results in one key-value pair per key. This single key-value pair cannot span across multiple worker nodes -> Data are shuffled across workers (sent over network) which slows downs the application as network communication introduces huge latencies.

Prefer `reduceByKey` when possible.

Spark, by default, uses *hash partitioning* to determine which key-value pair should be sent to which machine.

Partitioning

Data within an RDD is split into several ***partitions*** such that

- partitions never span multiple machines (tuples in same partition stays on same machine)
- Each machine contains at least one partition
- The number of partitions is configurable, by defaults it equals the total number of cores on all executor nodes.

In Spark two kinds of partitioning: **Hash partitioning** and **Range partitioning**.

Two ways to create RDDs with specific partitionings: 1. Call `partitionBy` on an RDD, providing an explicit `Partitioner`:

`def partitionBy(partitioner: Partitioner): RDD[(K, V)]` 2. Using transformations that return RDDs with specific partitioners.

We can retrieve the partitioner of an rdd with `.partitioner` which returns `Option[Partitioner]`

EX

```
val pairs = purchasedRdd.map(p => (p.customerId, p.price))
val tunedPartitioner = new RangePartitioner(8, ranges)
val partitioned = pairs.partitionBy(tunedPartitioner).persist()
```

The result of `partitionBy` should be persisted. Otherwise, the partitioning is repeatedly applied each time the partitioned RDD is used

Hash partitioning

`groupByKey` first computes, for each tuple, `(k, v)` its partition: `p = k.hashCode() % numPartitions` and then sends all the tuples in the same partition `p` to the same machine hosting `p`

Create hash partitioner: `val partitioner: HashPartitioner = new HashPartitioner(numOfPartitions)`

Range Partitioning

Pair RDDs may contain keys that have an *ordering* defined (Int, Char, String, ...). For these RDDs, *range partitioning* may be more efficient. **Tuples with keys in the same range appear on the same machine**. Keys are partitioned according to:

1. An *ordering* for keys
2. A set of *sorted ranges* for keys.

Create range partitioner: `val partitioner: RangePartitioner = new RangePartitioner(numOfPartitions, rddTOpartition)`

Partitioning using Transformations

Pair RDD resulting from a **transformation on a partitioned Pair RDD** are typically configured to use the hash partitioner that was used to construct the orginal RDD.

Some operations **automatically** result in an RDD with a known partitioner. For example, using `sortByKey` results in partition using a `RangePartitioner`, while using `groupByKey` results in using a `HashPartitioner`.

Operations on PairRDD that **hold to (and propagate) a partiioner**

cogroup	foldByKey	groupWith
combineByKey	join	partitionBy
leftOuterJoin	rightOuterJoin	join
sort	mapValues ()	flatMapValues ()
groupByKey	reduceByKey	filter()
sort		

(): if parent has partitioner. **All other operations will produce a result without a partitioner** (for example, when using map, it does not make sense to keep the partitioner if we change the keys) Therefore `mapValues` it's not only a shortcut, but also enables us to do map transformations without changing the keys, preserving the partitioner. **Using range partitioners we can optimize reduceByKey so that it does not involve any shuffling over the network at all** (9x speed up)

A shuffle *can* occur when the resulting RDD depends on other elements from the same RDD or another RDD.

When a shuffle occurs, the return type is `ShuffledRDD[x]`

Operations that might cause a shuffle:

cogroup	groupWith	join
leftOuterJoin	rightOuterJoin	groupByKey
reduceByKey	combineByKey	distinct
intersection	repartition	coalesce

Running reduceByKey on a prepartitioned RDD will cause the values to be computed locally, requiring only the final reuslt to be sent over the network. Similarly `join` called on two RDDs prepartitioned with the same partitioner and cached on the same machine will cause the join to be computed locally, with no shuffling.

Closures

are passed to most transformations and to some actions (reduce, foreach). ISSUES:

- **Serialization** at runtime, when closures are not serializable
- **Closures that are too large**.

```
class App{
  val rdd: RDD[X] = ...
```

```
  val localObject = Map[xxx]
  def compute(): Array[X] = {
    val filtered = rdd.filter(o => localObject.exists(..))
  }
  ...
}
```

-> `java.io.NotSerializableException` : the closure (compute) is not serializable. *A closure is serializable if all captured variables are serializable.* Here the captured variables are `localObject` **AND `this`** which is not serializable as **App does not extend `Serializable`**. Solution: copy `localObject` into a local variable : `val local = localObject; val filtered = rdd.filter(o => local.exists....)`

Shared Variables

When a function is passed to a Spark operation, its variables are all copied on each machine and the updates made to these objects are not propagated back to the driver program.

Spark provides two types of shared variables:

1. Broadcast variables

in the previous example, localObject can be huge and several operation may require it. In this case **broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks**.

```
val broadcastVariable = sc.broadcast(variable)
val filtered = rdd.filter(o => broadcastVariable.value ......)
```

2. Accumulators

Can be only modified through **associative operations** and provide a simple syntax for aggregating values from workers back to the driver program. By default, **spark only supports numeric accumulators**.

```
val counter = sc.accumulator(0)
for (r <- rdd, if (...)) counter += 1
```

FAULT TOLERANCE: each task is applied to each accumulator only once. An accumulator update within a transformation can occur more than once (when RDD recomputed) and should only be used for debugging in transformations.