Partitioning

Big Data Analysis with Scala and Spark

Heather Miller

“Partitioning”?

In the last session, we were looking at an example involving `groupByKey`, before we discovered that this operation causes data to be *shuffled* over the network.

> *Grouping all values of key-value pairs with the same key requires collecting all key-value pairs with the same key on the same machine.*

We concluded the last session asking ourselves,

But how does Spark know which key to put on which machine?

Before we try to optimize that example any further, let’s first take a quick detour into what partitioning is...

Partitions

The data within an RDD is split into several *partitions*.

Properties of partitions:

- Partitions never span multiple machines, i.e., tuples in the same partition are guaranteed to be on the same machine.
- Each machine in the cluster contains one or more partitions.
- The number of partitions to use is configurable. By default, it equals the *total number of cores on all executor nodes*.

Two kinds of partitioning available in Spark:

- Hash partitioning
- Range partitioning

Customizing a partitioning is only possible on Pair RDDs.

Hash partitioning

Back to our example. Given a Pair RDD that should be grouped:

```
val purchasesPerCust =
  purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD
              .groupByKey()
```

Hash partitioning

Back to our example. Given a Pair RDD that should be grouped:

```
val purchasesPerCust =
  purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD
              .groupByKey()
```

`groupByKey` first computes per tuple `(k, v)` its partition `p`:

```
p = k.hashCode() % numPartitions
```

Then, all tuples in the same partition `p` are sent to the machine hosting `p`.

Intuition: hash partitioning attempts to spread data evenly across partitions *based on the key*.

Range partitioning

Pair RDDs may contain keys that have an *ordering* defined.

- Examples: `Int`, `Char`, `String`, ...

For such RDDs, *range partitioning* may be more efficient.

Using a range partitioner, keys are partitioned according to:

1. an *ordering* for keys
2. a set of *sorted ranges* of keys

Property: tuples with keys in the same range appear on the same machine.

Hash Partitioning: Example

Consider a Pair RDD, with keys `[8, 96, 240, 400, 401, 800]`, and a desired number of partitions of `4`.

Hash Partitioning: Example

Consider a Pair RDD, with keys `[8, 96, 240, 400, 401, 800]`, and a desired number of partitions of `4`.

Furthermore, suppose that `hashCode()` is the identity (`n.hashCode() == n`).

$p = K.\text{hashcode}() \;\%\; \text{numPartitions}$

$= K \% 4$

Hash Partitioning: Example

Consider a Pair RDD, with keys `[8, 96, 240, 400, 401, 800]`, and a desired number of partitions of `4`.

Furthermore, suppose that `hashCode()` is the identity (`n.hashCode() == n`).

In this case, hash partitioning distributes the keys as follows among the partitions:

- partition 0: `[8, 96, 240, 400, 800]`
- partition 1: `[401]`
- partition 2: `[]`
- partition 3: `[]`

$p = k \% 4$

The result is a very unbalanced distribution which hurts performance.

Range Partitioning: Example

Using *range partitioning* the distribution can be improved significantly:

- Assumptions: (a) keys non-negative, (b) `800` is biggest key in the RDD.
- Set of ranges: `[1, 200]`, `[201, 400]`, `[401, 600]`, `[601, 800]`

Range Partitioning: Example

Using *range partitioning* the distribution can be improved significantly:

- Assumptions: (a) keys non-negative, (b) `800` is biggest key in the RDD.
- Set of ranges: `[1, 200]`, `[201, 400]`, `[401, 600]`, `[601, 800]`

In this case, range partitioning distributes the keys as follows among the partitions:

- partition 0: `[8, 96]`
- partition 1: `[240, 400]`
- partition 2: `[401]`
- partition 3: `[800]`

The resulting partitioning is much more balanced.

Partitioning Data

How do we set a partitioning for our data?

Partitioning Data

How do we set a partitioning for our data?

There are two ways to create RDDs with specific partitionings:

1. Call `partitionBy` on an RDD, providing an explicit `Partitioner`.
2. Using transformations that return RDDs with specific partitioners.

Partitioning Data: `partitionBy`

Invoking `partitionBy` creates an RDD with a specified partitioner.

Partitioning Data: `partitionBy`

Invoking `partitionBy` creates an RDD with a specified partitioner.

Example:

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))
```

Partitioning Data: `partitionBy`

Invoking `partitionBy` creates an RDD with a specified partitioner.

Example:

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))

val tunedPartitioner = new RangePartitioner(8, pairs)
val partitioned = pairs.partitionBy(tunedPartitioner).persist()
```

Partitioning Data: `partitionBy`

Invoking `partitionBy` creates an RDD with a specified partitioner.

Example:

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))

val tunedPartitioner = new RangePartitioner(8, pairs)
val partitioned = pairs.partitionBy(tunedPartitioner).persist()
```

Creating a `RangePartitioner` requires:

1. Specifying the desired number of partitions.
2. Providing a Pair RDD with *ordered keys*. This RDD is *sampled* to create a suitable set of *sorted ranges*.

Partitioning Data: `partitionBy`

Invoking `partitionBy` creates an RDD with a specified partitioner.

Example:

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))

val tunedPartitioner = new RangePartitioner(8, pairs)
val partitioned = pairs.partitionBy(tunedPartitioner).persist()
```

Creating a `RangePartitioner` requires:

1. Specifying the desired number of partitions.
2. Providing a Pair RDD with *ordered keys*. This RDD is *sampled* to create a suitable set of *sorted ranges*.

Important: the result of `partitionBy` should be persisted. Otherwise, the partitioning is repeatedly applied (involving shuffling!) each time the partitioned RDD is used.

Partitioning Data Using Transformations

Partitioner from parent RDD:

Pair RDDs that are the result of a transformation on a *partitioned* Pair RDD typically is configured to use the hash partitioner that was used to construct it.

Automatically-set partitioners:

Some operations on RDDs automatically result in an RDD with a known partitioner – for when it makes sense.

For example, by default, when using `sortByKey`, a `RangePartitioner` is used. Further, the default partitioner when using `groupByKey`, is a `HashPartitioner`, as we saw earlier.

Partitioning Data Using Transformations

Operations on Pair RDDs that hold to (and propagate) a partitioner:

- `cogroup`
- `groupWith`
- `join`
- `leftOuterJoin`
- `rightOuterJoin`
- `groupByKey`
- `reduceByKey`
- `foldByKey`
- `combineByKey`
- `partitionBy`
- `sort`
- `mapValues` (if parent has a partitioner)
- `flatMapValues` (if parent has a partitioner)
- `filter` (if parent has a partitioner)

All other operations will produce a result without a partitioner.

Partitioning Data Using Transformations

...All other operations will produce a result without a partitioner.

Why?

Partitioning Data Using Transformations

...All other operations will produce a result without a partitioner.

Why?

Consider the `map` transformation. Given that we have a hash partitioned Pair RDD, why would it make sense for `map` to lose the partitioner in its result RDD?

Partitioning Data Using Transformations

...All other operations will produce a result without a partitioner.

Why?

Consider the `map` transformation. Given that we have a hash partitioned Pair RDD, why would it make sense for `map` to lose the partitioner in its result RDD?

Because it's possible for `map` to change the key . *E.g.*,:

Partitioning Data Using Transformations

...All other operations will produce a result without a partitioner.

Why?

Consider the `map` transformation. Given that we have a hash partitioned Pair RDD, why would it make sense for `map` to lose the partitioner in its result RDD?

Because it's possible for `map` to change the key . *E.g.*,:

```
rdd.map((k: String, v: Int) => ("doh!", v))
```

In this case, if the `map` transformation preserved the partitioner in the result RDD, it no longer make sense, as now the keys are all different.

Hence `mapValues`. It enables us to still do `map` transformations without changing the keys, thereby preserving the partitioner.