Partitioning for PageRank

Reza Zadeh

ICME
INSTITUTE for COMPUTATIONAL & MATHEMATICAL ENGINEERING at STANFORD UNIVERSITY

Matroid

Spark

@Reza_Zadeh | http://reza-zadeh.com
Motivation

Recall from first lecture that network bandwidth is $\sim 100\times$ as expensive as memory bandwidth.

One way Spark avoids using it is through locality-aware scheduling for RAM and disk.

Another important tool is controlling the \textit{partitioning} of RDD contents across nodes.
Spark PageRank

Given directed graph, compute node importance. Two RDDs:

» Neighbors (a sparse graph/matrix)

» Current guess (a vector)

Best representation for vector and matrix?
Example

1. Start each page at a rank of 1
2. On each iteration, have page $p$ contribute $\frac{\text{rank}_p}{|\text{neighbors}_p|}$ to its neighbors
3. Set each page’s rank to $0.15 + 0.85 \times \text{contribs}$

```scala
val links = // RDD of (url, neighbors) pairs
var ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  val contribs = links.join(ranks).flatMap {
    case (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  ranks = contribs.reduceByKey(_ + _).mapValues(.15 + .85*__)
}
```
Execution

Input File

Links (url, neighbors)

map

Ranks₀ (url, rank)

join

Contribs₀

reduceByKey

Ranks₁

join

Contribs₂

reduceByKey

Ranks₂

...

links and ranks are repeatedly joined

Each join requires a full shuffle over the network

» Hash both onto same nodes
Solution

Pre-partition the links RDD so that links for URLs with the same hash code are on the same node.

```scala
val ranks = // RDD of (url, rank) pairs
val links = sc.textFile(...).map(...)
    .partitionBy(new HashPartitioner(8))

for (i <- 1 to ITERATIONS) {
    ranks = links.join(ranks).flatMap {
        (url, (links, rank)) =>
        links.map(dest => (dest, rank/links.size))
    }.reduceByKey(_ + _)
    .mapValues(0.15 + 0.85 * _)
}
```
New Execution

Input File → Links → Ranks

map partitionBy

Ranks_0 → join flatMap reduceByKey → Ranks_1 → join flatMap reduceByKey → Ranks_2 → ...

Links not shuffled

Ranks also not shuffled
How it works

Each RDD has an optional Partitioner object.

Any shuffle operation on an RDD with a Partitioner will respect that Partitioner.

Any shuffle operation on two RDDs will take on the Partitioner of one of them, if one is set.
Examples

```
pages.join(visits).reduceByKey(...)
```

Output of join is already partitioned

```
pages.join(visits).map(...).reduceByKey(...)
```

map loses knowledge about partitioning

```
pages.join(visits).mapValues(...).reduceByKey(...)
```

mapValues retains keys unchanged
Main Conclusion

Controlled partitioning can avoid unnecessary all-to-all communication, saving computation.

Repeated joins generalizes to repeated Matrix Multiplication, opening many algorithms from Numerical Linear Algebra.
Performance

Why it helps so much: Links RDD is much bigger in bytes than ranks!
RDD partitioner

Use the `.partitioner` method on RDD

```scala
scala> val a = sc.parallelize(List((1, 1), (2, 2)))
scala> val b = sc.parallelize(List((1, 1), (2, 2)))
scala> val joined = a.join(b)

scala> a.partitioner
res0: Option[Partitioner] = None

scala> joined.partitioner
res1: Option[Partitioner] = Some(HashPartitioner@286d41c0)
```
Custom Partitioning

Can define your own subclass of Partitioner to leverage domain-specific knowledge

Example: in PageRank, hash URLs by domain name, because may links are internal

class DomainPartitioner extends Partitioner {
    def numPartitions = 20

    def getPartition(key: Any): Int =
        parseDomain(key.toString).hashCode % numPartitions

    def equals(other: Any): Boolean =
        other.isInstanceOf[DomainPartitioner]
}