Programming Models at Scale

Lijie Xu

ISCAS

June, 2014

Outline

Data processing

Dataflow

data $\Rightarrow$ compute() $\Rightarrow$ (intermediate result $\Rightarrow$ compute())* $\Rightarrow$ result

- No modification on *data*
- No need to maintain state in this one-pass dataflow
    - No modification on *intermediate result*
    - state may be kept in compute(), but it is self-contained

Which programming model is suitable for scalable data processing?

Procedural programming model

- Applications: operating system, database, Web server, etc.
- Pros
  - imperative flow => user-friendly (easy to implement algorithms)
  - stateful (have mutable variables)
  - fine-grained operations
- Cons => hard to scale out
  - stateful (mutable variables) => consistency
    - locks, synchronization, etc.

Object-oriented programming model

- Applications: Web server, Web/desktop applications, etc.
- Pros
  - imperative flow
  - encapsulated stateful interaction (object-level)
  - ease of design / maintenance / evolution
- Cons => hard to scale out

Functional programming model

Treats computation as the evaluation of mathematical functions and avoids ***state*** *and* ***mutable*** *data.*

$$y = g(f(x))$$

Object-oriented vs. Functional programming

How to model "A cat catches a bird and eats it"?

- OO => top-down => focus on "nouns"
  - see two nouns: *cat* and *bird*
  - see two verbs: *catch* and *eat*
  - *Cat cat; Bird bird; cat.cache(bird); cat.eat(bird)*
- FP => bottom-up => focus on "verbs"
  - see two verbs: *catch* and *eat*, then define functions with type system
  - *result = eat (catch(cat, bird))*

```
trait Cat
trait Bird
trait Catch
trait FullTummy

def catch(hunter: Cat, prey: Bird): Cat with Catch
def eat(consumer: Cat with Catch): Cat with FullTummy

val story = (catch _) andThen (eat _)
story(new Cat, new Bird)
```

Functional programming model

- Composition of functions (verbs) => abstract behavior
- Type system
    - stateful variables => keep the state of intermediate computing results
    - type system => keep the *"state"* of interconnected functions
- No side effects => idempotent function
    - does not modify a global variable, static variable, one of its arguments
- Lazy evaluation
    - delays the evaluation of an expression until its value is needed

Theories of functional programming model

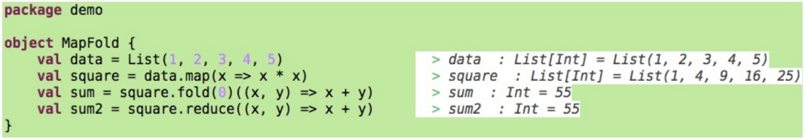
- Lambda calculus
  - e.g., $\lambda x.x^2$, $\lambda x \lambda y.x + y$
- Higher-order functions
  - takes one or more functions as an input, outputs a function
- Currying
  - a function with multiple arguments => a chain of functions (each has a single argument)

Data processing - Revisited

data $\Rightarrow$ compute() $\Rightarrow$ (intermediate result $\Rightarrow$ compute())* $\Rightarrow$ result

- No state in the dataflow
  - FP is suitable for defining the dataflow
- Algorithms in compute()
  - Imperative programming models such as Procedural and OO are suitable for the algorithm implementation

Two typical functions in functional programming language



```
package demo

object MapFold {
    val data = List(1, 2, 3, 4, 5)
    val square = data.map(x => x * x)
    val sum = square.fold(0)((x, y) => x + y)
    val sum2 = square.reduce((x, y) => x + y)
}
```

```
> data   : List[Int] = List(1, 2, 3, 4, 5)
> square  : List[Int] = List(1, 4, 9, 16, 25)
> sum  : Int = 55
> sum2   : Int = 55
```

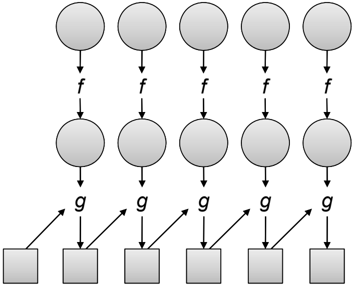


Figure 2.1: Illustration of *map* and *fold*, two higher-order functions commonly used together in functional programming: *map* takes a function f and applies it to every element in a list, while *fold* iteratively applies a function g to aggregate results.

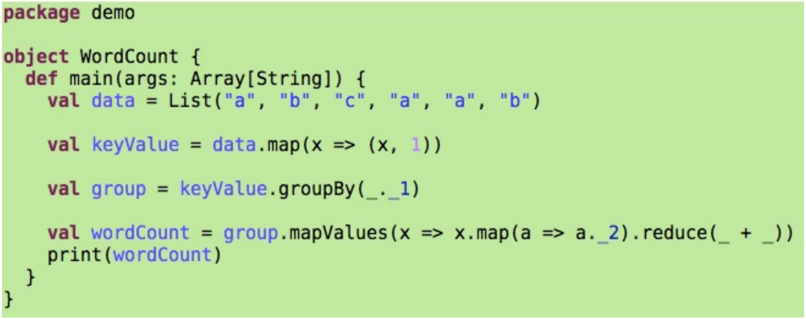
Scale out map()-fold/reduce()

- map() is easy to scale out, but fold/reduce() is hard to scale out
- However, fold()/reduce() can become scalable
    - If the result of map() can be partitioned, fold()/reduce() only needs to process partial but independent result

Programming model of MapReduce

It is amazing that only three functions can express the dataflow of a lot of data processing algorithms.

- $map(k_1, v_1) \Rightarrow (k_2, v_2)$
- $groupBy(list(k_2, v_2)) \Rightarrow (k_2, list(v_2))$
- $reduce(k_2, list(v_2)) \Rightarrow (k_3, v_3)$



```
package demo

object WordCount {
  def main(args: Array[String]) {
    val data = List("a", "b", "c", "a", "a", "b")

    val keyValue = data.map(x => (x, 1))

    val group = keyValue.groupBy(_._1)

    val wordCount = group.mapValues(x => x.map(a => a._2).reduce(_ + _))
    print(wordCount)
  }
}
```

Programming model of MapReduce

- Fixed functions with fixed parameters
- Imperative flow in the functions

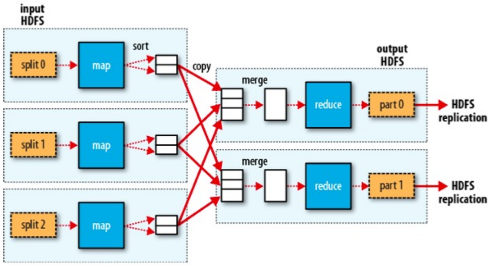
```java
public class TypicalMapper extends Mapper {

  private Object buffer;

  public void map(K key, V value) {

    Object result = process(key, value);

    buffer.cache(result);  // may exist

    generateAndOutput(newKey, newValue);
  }
}
```

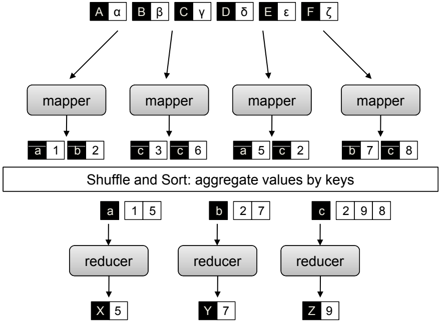
Figure 1: General mapper

```java
public class TypicalReducer extends Reducer {

   private Object cbuffer;

   public void reduce(K key, Iterable<V> values) {
     Object gbuffer;

     for (V value : values) {

       Object result= process(key, value);

       gbuffer.cache(result);  // may exist

       generateAndOutput(newKey, newValue); //can be here
     }

     cbuffer.cache(results in gbuffer); // may exist

     generateAndOutput(newKey, newValue); //can be here too
   }
 }
```

Figure 2: General reducer

MapReduce dataflow (execution engine)



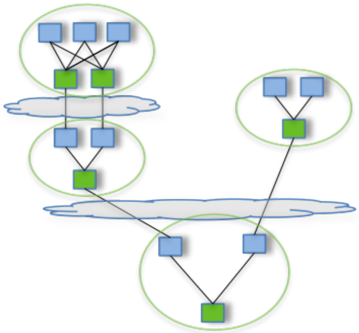


Pros of MapReduce

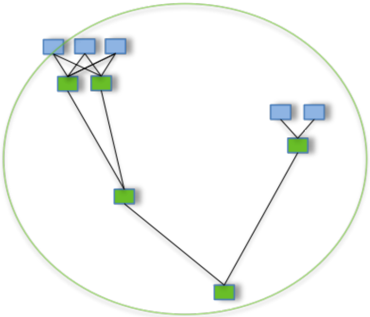
- Scalable
  - Divide and conquer => partition and aggregation
  - No state in the dataflow
- Simple programming model
  - map() and reduce() are imperative and user-defined
  - groupBy() is automatically performed by framework
- Simple fixed dataflow
  - A sequence of map/partition/shuffle/sort/reduce operations
- Fault-tolerant
  - can rerun mapper/reducer because functions are
    - idempotent
    - deterministic

Cons of MapReduce

- Simple fixed dataflow => weak expression
  - any data processing algorithms should follow map-reduce pattern
    - carefully design K, V. (e.g., too tricky to implement JOIN)
  - cannot express complex dataflow efficiently
  - lack type system => need to explicitly keep track of objects



Pig/Hive - MR



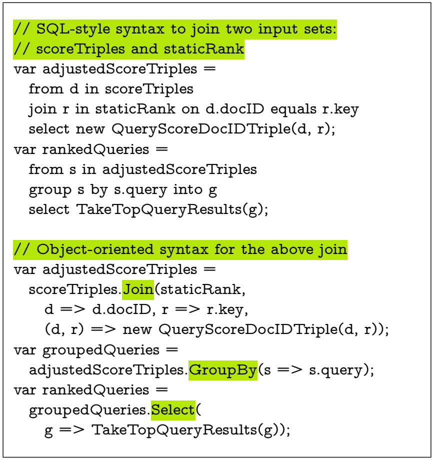
Pig/Hive - Tez

Cons of SQL-like languages such as Hive, Pig, etc.

- Has a very restrictive and simple custom type system
    - e.g., hard to express <K, List<V>>
- “query-oriented”
    - hard to express common patterns such as *iteration*
- A limited hybridization of declarative and imperative programs
    - UDF

Programming model of DraydLINQ

- General-purpose language
  - like writing in a single computer
  - imperative + declarative
- Arbitrary side-effect-free transformations
- Expressive data model
  - strongly-typed collections of partitioned .NET objects

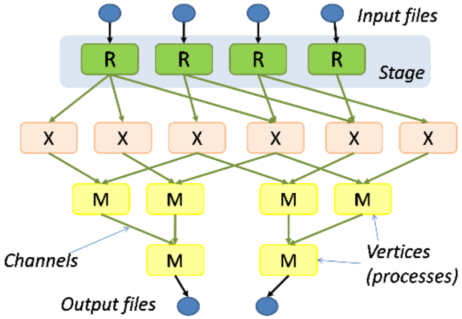


```
// SQL-style syntax to join two input sets:
// scoreTriples and staticRank
var adjustedScoreTriples =
  from d in scoreTriples
  join r in staticRank on d.docID equals r.key
  select new QueryScoreDocIDTriple(d, r);
var rankedQueries =
  from s in adjustedScoreTriples
  group s by s.query into g
  select TakeTopQueryResults(g);

// Object-oriented syntax for the above join
var adjustedScoreTriples =
  scoreTriples.Join(staticRank,
    d => d.docID, r => r.key,
    (d, r) => new QueryScoreDocIDTriple(d, r));
var groupedQueries =
  adjustedScoreTriples.GroupBy(s => s.query);
var rankedQueries =
  groupedQueries.Select(
    g => TakeTopQueryResults(g));
```

Figure 3: *A program fragment illustrating two ways of expressing the same operation. The first uses LINQ's declarative syntax, and the second uses object-oriented interfaces. Statements such as* `r => r.key` *use C#'s syntax for anonymous lambda expressions.*

DryadLINQ => distributed execution plan executed by Dryad

- More general and flexible dataflow
  - dataflow graph
  - multiple inputs and multiple outputs, of different types
- Fault-tolerance
  - re-execution (deterministic vertex, acyclic graph, immutable inputs)



DraydLINQ - Revisited

- Pros
  - Let users focus on data processing procedure and algorithms
    - do not need to follow map-reduce pattern
    - *forget* the distributed issues (like in a single computer)
    - use type system to describe the input/output/intermediate data
    - function chain is dataflow chain
  - Automatic translation and optimization
    - DAG generation (modeling data dependency)
    - Stage partition (task generation)
    - Data model: collection of disjoint partitions
    - Pipelining
    - Dynamic hash and range partition
- Cons
  - Coarse-grained parallelization
  - All functions invoked should be side-effect free

Programming model of Spark

data ⇒ (transformation() ⇒ RDD[partition[records]])* ⇒ action() ⇒ result

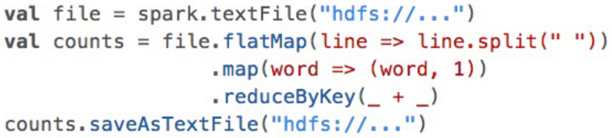
- Similar with DryadLINQ
- More emphasis on data model called *RDD*
    - read-only, partitioned collection of records
    - *"parallel collections" (similar with List<Array<T>>)*

More about RDD

- RDD
  - General-purpose
    - leverage type system
    - coarse-grained bulk transformations (e.g., map, filter and join)
  - Efficient
    - only one copy, mainly in-memory, pipelining, sharable
  - Fault-tolerant
    - lineage-based recovery => re-computation
- transformation()
  - compute()
- action()
  - trigger the execution of compute() chain

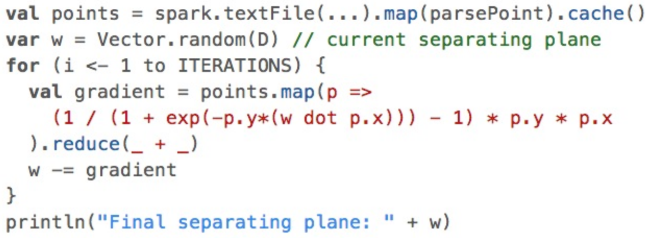
Spark examples => data + transformation() + action()

- WordCount



```
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

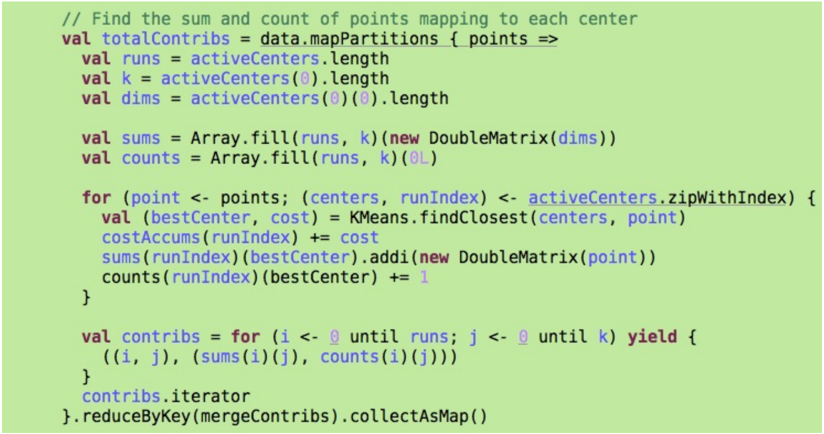
- Logistic regression



```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.random(D) // current separating plane
for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}
println("Final separating plane: " + w)
```

Spark examples => Kmeans

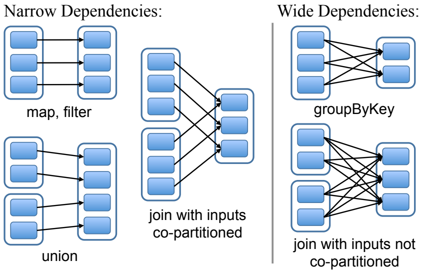
- function chain
  - data.mapPartitions().reduceByKey().collectAsMap()
- imperative flow in mapPartitions()



```
// Find the sum and count of points mapping to each center
val totalContribs = data.mapPartitions { points =>
  val runs = activeCenters.length
  val k = activeCenters(0).length
  val dims = activeCenters(0)(0).length

  val sums = Array.fill(runs, k)(new DoubleMatrix(dims))
  val counts = Array.fill(runs, k)(0L)

  for (point <- points; (centers, runIndex) <- activeCenters.zipWithIndex) {
    val (bestCenter, cost) = KMeans.findClosest(centers, point)
    costAccums(runIndex) += cost
    sums(runIndex)(bestCenter).addi(new DoubleMatrix(point))
    counts(runIndex)(bestCenter) += 1
  }

  val contribs = for (i <- 0 until runs; j <- 0 until k) yield {
    ((i, j), (sums(i)(j), counts(i)(j)))
  }
  contribs.iterator
}.reduceByKey(mergeContribs).collectAsMap()
```

Spark dataflow

- DAG => multiple types of data dependency
- Stage partition => task generation
- Piplining => records are computed when needed



Narrow Dependencies:
Wide Dependencies:
map, filter
groupByKey
join with inputs co-partitioned
union
join with inputs not co-partitioned

Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

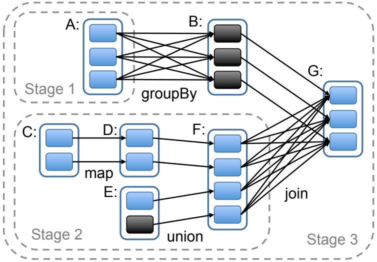


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

Spark runtime

- Driver => main()
- Worker => compute()

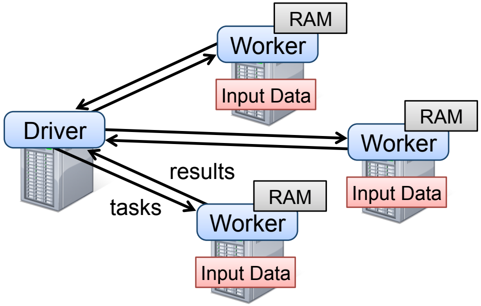


Figure 2: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

Pros of Spark

- has the pros of DryadLINQ
- leverage distributed memory
- reuse data and intermediate computing results
- suitable for *iterative* and *interactive* computing

Cons of Spark

- Has the cons of DryadLINQ
- need to maintain balance between memory and disk
- Compute() on each partition is independent
  - worker only communicates with driver
  - If compute() needs the data or computing result in other partitions, need to perform another transformation() or perform collect-broadcast using driver.

Summary

- No need to keep state in the dataflow
- Use FP to define dataflow
- Use imperative flow in compute()

References

Most figures and descriptions come from

[1] Dean J, Ghemawat S. *“MapReduce: simplified data processing on large clusters.”* ODSI, 2004.
[2] Yu, Yuan, et al. *"DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language."* OSDI, 2008.
[3] Isard, Michael, et al. *"Dryad: distributed data-parallel programs from sequential building blocks."* EuroSys, 2007.
[4] Zaharia M, Chowdhury M, Das T, et al. *“Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing.”* NSDI, 2012.
[5] Lin J, Dyer C. *“Data-intensive text processing with MapReduce.”* Synthesis Lectures on Human Language Technologies, 2010, 3(1): 1-177.
[6] Suereth J D. *“Scala in depth.”* Manning Publications, 2012.
[7] White T. *“Hadoop: The definitive guide."* O’Reilly Media, Inc.", 2012.