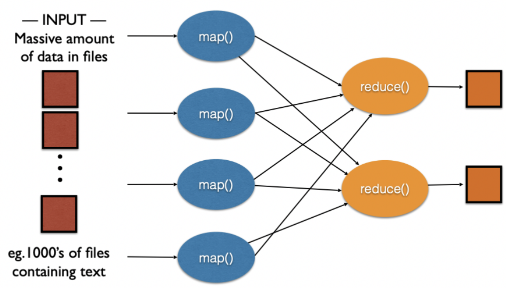
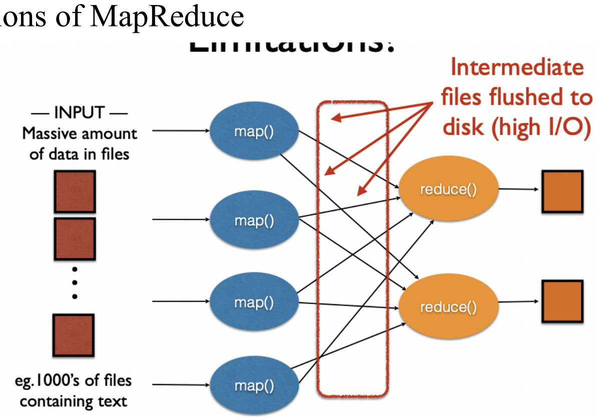
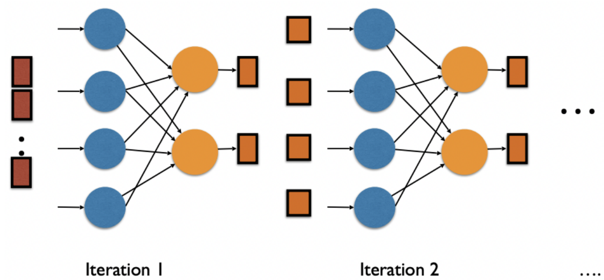
CAS CS 350
Lec 20

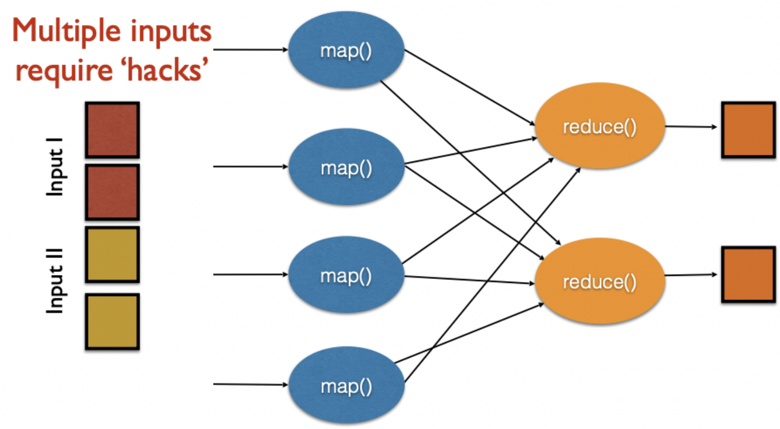
Spark

1. Why Spark?
    a. General-purpose cluster computing framework
    b. Better performance than Hadoop (open-source MapReduce)
    c. Open source protocol
    d. User-friendly fault-tolerant abstractions (RDDs)
    e. Rich dataflow API (incl. join(), flatMap(), groupByKey(), etc.)
    f. A big commercial success
2. MapReduce
    a. 

    b. User must specify two functions → map and reduce functions
3. Limitations of MapReduce
    a. 

    b. Intermediate files flushed to disk (high I/O)
    c. Users do not know anything about the file regarding memory and etc. but is up to the system

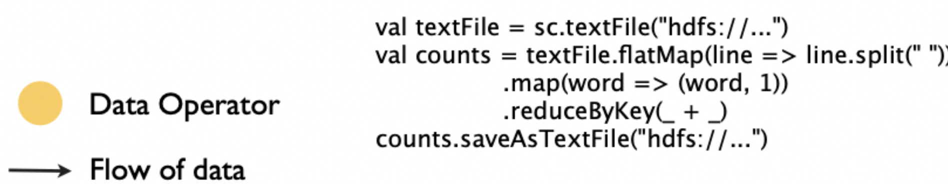
d. Iterations require manual work by users → can only implement one map and one reduce function at a time (does not support building map and reduce functions together) → users have to know where the output data is stored to send to another mapReduce protocol repeatedly
e. 

f. Multiple inputs require 'hacks'
g. 

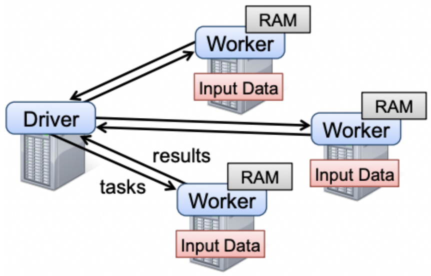
h. MapReduce does not support more than one inputs (only considers that we have a big logical file separated into many text files that belong to one input data)
i. If you want to insert multiple inputs, you somehow have to include this information into the input record
j. Data is stored on disk
k. How would map() and reduce() distinguish between k-v pairs from different inputs?
    i. Put it inside the record manually by the user
l. Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse
m. They do not provide abstractions for more general reuse, e.g., to let a user load several datasets into memory and run ad-hoc queries across them

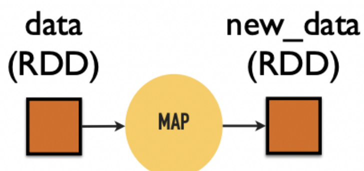
4. Spark
    a. A new abstraction called resilient distributed datasets (RDDs) that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.
    b. RDD
        i. Abstraction
        ii. Data structure that users can manipulate
        iii. Fault-tolerant
        iv. Collection of data records that is physically distributed across machines
5. Dataflow Graphs

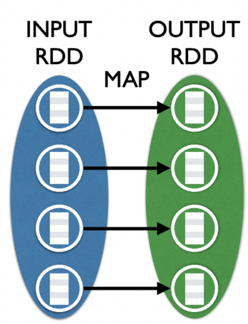
    a.
    b. Each data takes inputs as RDDs (one or more) and generates RDDs
    c. Example of wordcount operation by using Spark



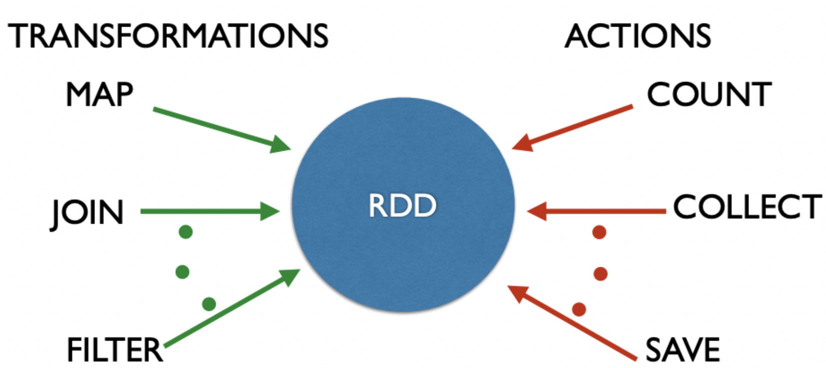
    d.
    e. Just specify the logic
    f. textFile → name for RDD (handle the file you read from disk)
    g. flatMap takes each record in the RDD (as line) and splits it into words
    h. The output (input of words) of it goes as input to map
    i. Map creates a new record and outputs RDD that is given input to ReduceByKey
    j. The ReduceByKey outputs a RDD
    k. Counts → another RDD that defines the output of the operator
    l. The last line of code saves the RDD in the distributed file system
    m. Everything in Spark is made of RDDs → input, output, between the functions

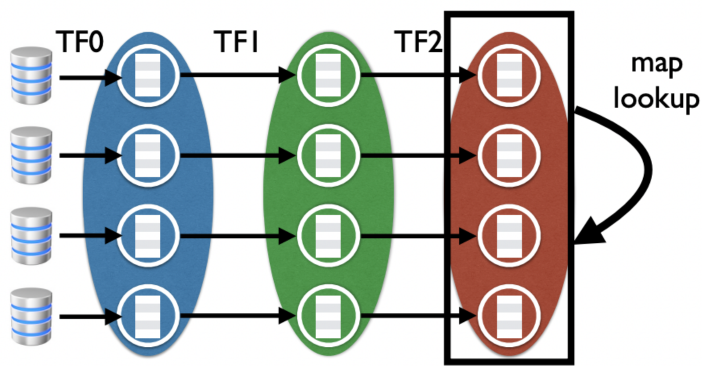
6. Spark Application
    a. 

    b. User writes application that is managed by the ‘driver’ (just like coordinator in MapReduce)
    c. Driver defines one or more RDD and invokes actions on them
    d. Workers can store data in RAM across operations (unlike MapReduce)
    e. More like writing a program than MapReduce - explicitly naming data and operating on it
7. RDD (collection of data records distributed in machines → memory or disk)
    a. RDDs provide an interface based on coarse-grained transformations
    b. 

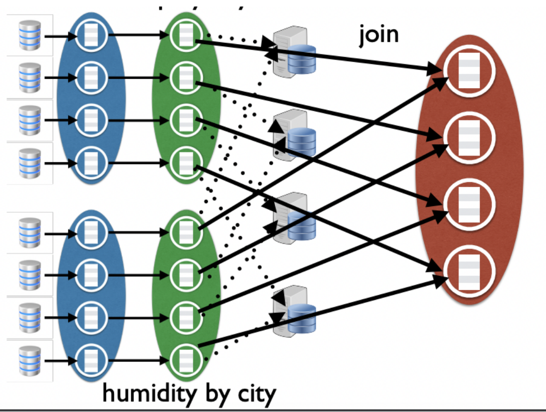
    c. Generated by an input dataset or by existing RDD
    d. Coarse grain operations that hide fine-grain memory R/Ws
    e. Same operation across many items within the object
    f. Restricted interface provides basis for Fault-Tolerance - log of operations per RDD (lineage)
8. Read-Only & SIMD (ensures fault tolerance)
    a. 

    b. Created never modified
    c. Computations is a series of transformations that produce create a new RDD
    d. Partitions of data items
    e. Data parallel model of computing - single operation applied in parallel to many data items

9. Lazy Evaluation
   a. 

   b. Green
      i. Compute a new RDD from existing RDDs
      ii. This just specifies a plan
      iii. Runtime is lazy - doesn't have to materialize (computer) so it doesn't
         1. Does not actually process data available until it has to
         2. No RDD is generated, it is stored in driver
         3. The output does not collect all the RDDs, just the output
         4. Action → specific
   c. Red
      i. Where some effect is requested: result to be stored, get specific value, etc. causes RDDs to materialize - return a value to app or to stable storage
10. Persistence
   a. Users can indicate which RDD's they will reuse and choose a storage strategy for them (eg. in-memory)
   b. 

   c. Users can set priorities of the output (ex. Written to disk first when there is no enough memory)

11. Partitioning
    a. Users can ask that an RDD's records be partition across machine based on a key in each record
    b. 

    c. Output has all pairs of records of the city

12. Logistic Regression

```
val points = spark.textFile(...)
                    .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

transformations

action

(cf. Table 2 in the paper)

    a. • w is sent with the closure to the nodes
    b. Points → RDD (keeping memory), persist() materializes in memory by default assuming memory is enough
    c. Map operators → data transformations (iterate through all record and apply the action (reduce))
    d. Action → materialize the RDD
    e. w is sent with the closure to the nodes
    f.

```
val points = spark.textFile(...)
                    .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

for each point p in points do:
p.x * f(w,p) * p.y

g. For each point p in points do:
   i. P.x * f(w,p) * p.y → not materialized
h. Reducer sums all input vectors and materialize a new RDD ‘gradient’
i.