The Hadoop Stack, Part 3

Introduction to Spark

CSE 40822 – Cloud Computing – Fall 2018
Prof. Douglas Thain
University of Notre Dame
Three Case Studies

• Workflow: Pig Latin
  • A dataflow language and execution system that provides an SQL-like way of composing workflows of multiple Map-Reduce jobs.

• Storage: HBase
  • A NoSQL storage system that brings a higher degree of structure to the flat-file nature of HDFS.

• Execution: Spark
  • An in-memory data analysis system that can use Hadoop as a persistence layer, enabling algorithms that are not easily expressed in Map-Reduce.
References

  • http://dl.acm.org/citation.cfm?id=1863103.1863113

• Holden Karau et al., Learning Spark : Lightning-Fast Big Data Analytics, O’ Reilly 2014.
  • http://shop.oreilly.com/product/0636920028512.do

• Apache Spark Documentation
  • http://spark.apache.org
Overview

• The Map-Reduce paradigm is fundamentally limited in expressiveness.
• Hadoop implementation of Map-Reduce is designed for out-of-core data, not in-memory data.
• Idea: Layer an in-memory system on top of Hadoop.
• Achieve fault-tolerance by re-execution instead of replication.
Map-Reduce Limitations

• As a general programming model:
  • It is perfect.... If your goal is to make a histogram from a large dataset!
  • Hard to compose and nest multiple operations.
  • No means of expressing iterative operations.
  • Not obvious how to perform operations with different cardinality.
    • Example: Try implementing All-Pairs efficiently.

• As implemented in Hadoop (GFS):
  • All datasets are read from disk, then stored back on to disk.
  • All data is (usually) triple-replicated for reliability.
  • Optimized for simple operations on a large amount of data.
  • Java is not a high performance programming language.
A Common Iterative Pattern in Data Mining

X = initial value
for( i=0; ; i++ ) {
    set $S_{i+1}$ = apply F to set $S_i$
    value X = extract statistic from $S_{i+1}$
    if( X is good enough ) break;
}

On Board: Implement in Map-Reduce
Can we do better?
The Working Set Idea

  - http://dl.acm.org/citation.cfm?id=363141

- Idea: conventional programs on one machine generally exhibit a high degree of locality, returning to the same data over and over again.

- The entire operating system, virtual memory system, compiler, and micro architecture are designed around this assumption!

- Exploiting this observation makes programs run 100X faster than simply using plain old main memory in the obvious way.

- (But in Map-Reduce, access to all data is equally slow.)
The Working Set Idea in Spark

• The user should identify which datasets they want to access.
• Load those datasets into memory, and use them multiple times.
• Keep newly created data in memory until explicitly told to store it.
• Master-Worker architecture: Master (driver) contains the main algorithmic logic, and the workers simply keep data in memory and apply functions to the distributed data.
• The master knows where data is located, so it can exploit locality.
• The driver is written in a functional programming language (Scala), so let’s detour to see what that means.
Detour: Pure Functional Programming

• Functions are first class citizens:
  • The primary means of structuring a program.
  • A function need not have a name!
  • A function can be passed to another program as a value.
  • A pure function has no side effects.

• In a pure functional programming language like LISP
  • There are no variables, only values.
  • There are no side effects, only values.

• Hybrid languages that have functional capabilities, but do not prohibit non-functional idioms: Scala, F#, JavaScript...
By the way, Map-Reduce is Inspired by LISP:

\[
\text{map}( (\lambda(x)( * x x )) (1 2 3 4 ) )
\]

\[
\text{reduce}( (\lambda(x y) (+ x y)) (1 2 3 4 ) )
\]
Functions in Scala:

Define a function in the ordinary way:

```scala
def name (arguments) { code }
```

Construct an anonymous func as a value:

```scala
( arguments ) => code
```

Accept an anonymous func as a parameter:

```scala
name: ( arguments ) => code
```

Example code:

```scala
def oncePerSecond(callback: () => Unit) {
    while( true ) { callback(); Thread.sleep 1000 }
}

def main(args: Array[String]) {
    oncePerSecond(
        () => println("time flies like an arrow...")
    )
}
```

val n = 10;
for( i <- 1 to n ) {
    // run code each value of i in parallel
}
var items = List(1,2,3);
for ( i <- items ) {
    // run code for each value of i in parallel
}
Back to Spark, Using Scala

• A program to count all the error lines in a large text file:

```scala
val file = spark.textFile("hdfs://path/to/file");
val errs = file.filter(_.contains("ERROR"));
val ones = errs.map(_ => 1);
val count = ones.reduce(_+_);
```

_
means “the default thing
that should go here.”

```scala
val file = spark.textFile("hdfs://path/to/file");
val errs = file.filter( x => x.contains("ERROR"));
val ones = errs.map( x => 1);
val count = ones.reduce( (x,y) => x+y );
```

On Board: Implement in Spark
Logistic Regression in Spark

val points = spark.textFile( ... ).map(parsePoint).cache()

var w = Vector.random(D)

for( i <- 1 to ITERATIONS ) {
    val grad = spark.accumulator( new Vector(D) )

    for( p <- points ) {
        val s = (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
        grad += s * p.x
    }
    w -= grad.value
}
Fault Tolerance via Recomputation

(Work out on the board.)
Result: Spark is 10-100X faster than Hadoop on equivalent iterative problems.

(It does everything in memory instead of disk.)