Lecture 21 - The MapReduce Algorithm

DSE 512

Drew Schmidt
2022-04-12

From Last Time

- Homework 3
  - Originally due Saturday April 9
  - Now due Wednesday April 13
  - THERE WILL BE NO ADDITIONAL EXTENSIONS
  - New homework (no earlier than) some time next week(?)
- Questions?

Today

- The MapReduce Algorithm

Background

MapReduce

MapReduce: Simplified Data Processing on Large Clusters

by Jeffrey Dean and Sanjay Ghemawat

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large datasets that is amenable to a broad variety of real-world tasks. Users specify the computation in terms of a *map* and a *reduce* function, and the underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks. Programmers find the system easy to use: more than ten thousand distinct MapReduce programs have been implemented internally at Google over the past four years, and an average of one hundred thousand MapReduce jobs are executed on Google's clusters every day, processing a total of more than twenty petabytes of data per day.

1 Introduction

Prior to our development of MapReduce, the authors and many others at Google implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, Web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of Web documents, summaries of the number of pages crawled per host, and the set of most frequent queries in a given day. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs. The programming model can also be used to parallelize computations across multiple cores of the same machine.

Section 2 describes the basic programming model and gives several examples. In Section 3, we describe an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that

Algorithm Overview

- Step 1: Read
- Step 2: Map
- Step 3: Shuffle
- Step 4: Reduce
- Step 5: Write

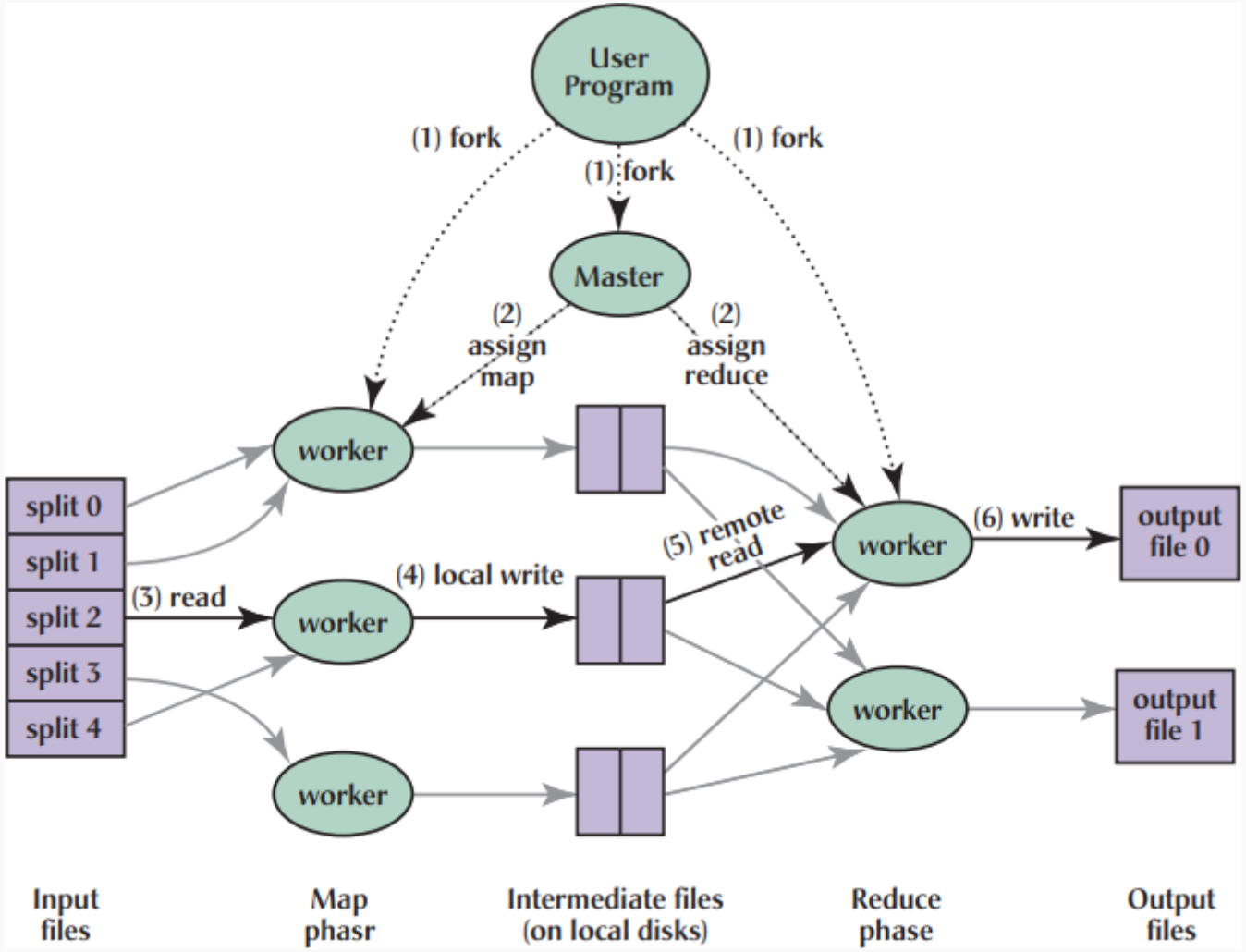
How It Works: User's Perspective

- Everything operates on key/value pairs
- input pair $\xrightarrow{map}$ intermediate pair
- intermediate pairs are automatically grouped
- intermediate key + iterator $\xrightarrow{reduce}$ values

map	(k1,v1)	→ list(k2,v2)
reduce	(k2,list(v2))	→ list(v2)

That is, the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

How It Works: Computer's Perspective



Implementations

- Hadoop
- Spark
- mrmpi

Scalability?

- "scale up"
- "scale out"
- What do these mysterious phrases mean?

Vertical (up) vs Horizontal (out) Scaling

- Industry jargon
- (Opinion) Not really that well-defined
- More about *hardware* than *applications*
- Example
  - *Scale up* - Get a better processor, add a GPU, ...
  - *Scale out* - Add more nodes of what you already have
- Which is MapReduce designed for?

The MapReduce 'Hello World'

Count the number of occurrences of a word across many text documents

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

Mapreduce (Spark) Pros and Cons

Pros

- No need to worry about parallel details
- Scaling "out" is straightforward: just add nodes
- I/O usually simple (many split text files)
- Can operate in memory or out-of-core
- Modern interfaces have SQL-like ETL
- Fault tolerance built in

Cons

- Not every problem naturally maps to key/value pairs
- Fault tolerance can't be turned off!
- Very difficult to set up
- Debugging borderline impossible
- Performance is *bad*

Hadoop vs RevoScaleR

Approach	Platform	Time to fit
1: SAS	16-core Sun Server	5 hours
2: rmr / map-reduce	10-node (8 cores / node) Hadoop cluster	> 10 hours
3: Open source R	250 GB Server	Impossible (> 3 days)
4: RevoScaleR	5-node (4 cores / node) LSF cluster	5.7 minutes

https://blog.revolutionanalytics.com/2012/10/allstate-big-data-glm.html

Spark vs pbdR

- A problem that takes several hours on Apache Spark [was analyzed] in less than a minute using R on OLCF high-performance hardware.
- "... for situations where one needs interactive near-real-time analysis, the pbdR approach is much better."

https://www.hpcwire.com/2016/07/06/olcf-researchers-scale-r-tackle-big-science-data-sets/

So Why Even Bother?

- For analytics, ***phenomenal*** question
- For ETL however…
  - Ask yourself
  - Not rhetorically
  - Actually ask yourself

> How would you implement an SQL join on distributed data with factors?

Setting Up a Spark Cluster

Setting Up a Spark Cluster

- ***This is really complicated***
- This is a ***full time job***
- Akin to "setting up ISAAC"
- Another analogy
  - Making SQL queries
  - Creating and maintaining SQL tables
  - Running the postgres database

Setting Up a Spark Cluster

We will be using this guide https://www.kdnuggets.com/2020/07/apache-spark-cluster-docker.html

- Dockerized
- Spark cluster with 1 master 2 workers
- Includes pyspark and Jupyter

```
docker-compose up
```

```
Creating spark-master ... done
Creating jupyterlab   ... done
Creating spark-worker-1 ... done
Creating spark-worker-2 ... done
spark-master      | Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
spark-master      | 22/04/10 14:53:08 INFO Master: Started daemon with process name: 7@ccf3714443b1
# ...
```

Master

Worker 0


localhost:8082
Spark 3.0.0
Spark Worker at 172.18.0.4:40697
ID: worker-20220410145310-172.18.0.4-40697
Master URL: spark://spark-master:7077
Cores: 1 (1 Used)
Memory: 512.0 MiB (512.0 MiB Used)
Resources:
Back to Master
Running Executors (1)
ExecutorID
State
Cores
Memory
Resources
Job Details
Logs
0
RUNNING
1
512.0 MiB
ID: app-20220410145357-0000
Name: pyspark-notebook
User: root
stdout stderr

Worker 1


localhost:8081
Spark 3.0.0
Spark Worker at 172.18.0.5:32891
ID: worker-20220410145310-172.18.0.5-32891
Master URL: spark://spark-master:7077
Cores: 1 (1 Used)
Memory: 512.0 MiB (512.0 MiB Used)
Resources:
Back to Master
Running Executors (1)
ExecutorID
State
Cores
Memory
Resources
Job Details
Logs
1
RUNNING
1
512.0 MiB
ID: app-20220410145357-0000
Name: pyspark-notebook
User: root
stdout stderr

Wordcount Example

Wordcount Step 1: Setup

```
import pyspark

sc = pyspark.SparkContext("local", "word count")
```

Wordcount Step 2: Import Data

```
import wget

url = "https://raw.githubusercontent.com/bwhite/dv_hadoop_tests/master/python-streaming/w
wget.download(url)
```

Wordcount Step 3: Import Data

```
words = sc.textFile("4300.txt").flatMap(lambda line: line.split(" "))
wordcounts = words.map(lambda word: (word, 1)).reduceByKey(lambda a,b: a + b)
wordcounts
```

```
PythonRDD[6] at RDD at PythonRDD.scala:53
```

Wordcount Step 3.5: Viewing the Data

```
wordcounts.take(10)
```

```
[('The', 1031),
 ('Project', 78),
 ('Gutenberg', 21),
 ('EBook', 2),
 ('of', 8127),
 ('Ulysses,', 3),
 ('by', 1173),
 ('James', 29),
 ('Joyce', 4),
 ('', 10116)]
```

Wordcount Step 4: Sorting

```
wordcounts_rbk = wordcounts.reduceByKey(lambda x,y: (x+y)).sortByKey()
wordcounts_rbk = wordcounts_rbk.map(lambda x: (x[1],x[0]))
wordcounts_rbk.sortByKey(False).take(10)
```

```
[(13600, 'the'),
 (10116, ''),
 (8127, 'of'),
 (6542, 'and'),
 (5845, 'a'),
 (4787, 'to'),
 (4606, 'in'),
 (3035, 'his'),
 (2712, 'he'),
 (2432, 'I')]
```

Wordcount Step 5: I/O

```
wordcounts.saveAsTextFile("counts")
```

Basic Dataframe Operations

Dataframes

- We'll be using the SQL operator interface
- There are other (arguably better) ones

Dataframes: Setup

```
import pyspark

spark = pyspark.sql.SparkSession.\
  builder.\
  appName("dataframe example").\
  master("spark://spark-master:7077").\
  config("spark.executor.memory", "512m").\
  getOrCreate()
```

Dataframes: Import Iris

```
import wget

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
wget.download(url)

iris = spark.read.csv("iris.data")
iris.show(n=5)
```

```
+---+---+---+---+-----------+
|_c0|_c1|_c2|_c3|        _c4|
+---+---+---+---+-----------+
|5.1|3.5|1.4|0.2|Iris-setosa|
|4.9|3.0|1.4|0.2|Iris-setosa|
|4.7|3.2|1.3|0.2|Iris-setosa|
|4.6|3.1|1.5|0.2|Iris-setosa|
|5.0|3.6|1.4|0.2|Iris-setosa|
+---+---+---+---+-----------+
only showing top 5 rows
```

Dataframes: Distributed?

```
iris.rdd.getNumPartitions()
```

```
1
```

Dataframes: Distributing

```
spark.conf.set("spark.sql.files.maxPartitionBytes", 1000)

iris = spark.read.csv("iris.data")
iris.rdd.getNumPartitions()
```

5

```
type(iris)
```

pyspark.sql.dataframe.DataFrame

Dataframes: Printing

```
iris.show(n=5)
```

```
+---+---+---+---+-----------+
|_c0|_c1|_c2|_c3|        _c4|
+---+---+---+---+-----------+
|5.1|3.5|1.4|0.2|Iris-setosa|
|4.9|3.0|1.4|0.2|Iris-setosa|
|4.7|3.2|1.3|0.2|Iris-setosa|
|4.6|3.1|1.5|0.2|Iris-setosa|
|5.0|3.6|1.4|0.2|Iris-setosa|
+---+---+---+---+-----------+
only showing top 5 rows
```

Dataframes: Renaming Columns

```
from functools import reduce
cn_old = iris.columns
cn_new = ['Sepal_Length', 'Sepal_Width', 'Petal_Length', 'Petal_Width', 'Species']
iris = reduce(lambda iris,idx: iris.withColumnRenamed(cn_old[idx], cn_new[idx]), range(le
iris.show(n=5)
```

```
+------------+-----------+------------+-----------+-----------+
|Sepal_Length|Sepal_Width|Petal_Length|Petal_Width|    Species|
+------------+-----------+------------+-----------+-----------+
|         5.1|        3.5|         1.4|        0.2|Iris-setosa|
|         4.9|        3.0|         1.4|        0.2|Iris-setosa|
|         4.7|        3.2|         1.3|        0.2|Iris-setosa|
|         4.6|        3.1|         1.5|        0.2|Iris-setosa|
|         5.0|        3.6|         1.4|        0.2|Iris-setosa|
+------------+-----------+------------+-----------+-----------+
only showing top 5 rows
```

Dataframes: Computing a Mean

```
from pyspark.sql.functions import *
iris.select('Sepal_Length').\
  agg(mean('Sepal_Length')).\
  show()
```

```
+-----------------+
|avg(Sepal_Length)|
+-----------------+
|5.843333333333333|
+-----------------+
```

```
mean(iris$Sepal.Length)
```

```
## [1] 5.843333
```

Dataframes: Group Means

```
iris.select('Sepal_Length', 'Species').\
  groupBy('Species').\
  agg(mean("Sepal_Length")).\
  show()
```

```
+---------------+-----------------+
|        Species|avg(Sepal_Length)|
+---------------+-----------------+
| Iris-virginica|6.587999999999999|
|    Iris-setosa|5.005999999999999|
|Iris-versicolor|5.935999999999999|
+---------------+-----------------+
```

```
aggregate(iris$Sepal.Length, list(iris$Species), FUN=mean)
```

```
##      Group.1     x
## 1     setosa 5.006
## 2 versicolor 5.936
## 3  virginica 6.588
```

Wrapup

Wrapup

- That's it for parallelism!
- What are the advantages/disadvantages of
  - Fork
  - MPI
  - MapReduce
- What are the advantages/disadvantages of
  - Manager/Worker
  - SPMD
- The homework will not involve MapReduce (code)

Questions?