Data-Intensive Distributed Computing
CS 431/631 451/651 (Fall 2019)

Part 1: MapReduce Algorithm Design (2/4)

Ali Abedi

These slides are available at https://www.student.cs.uwaterloo.ca/~cs451/

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States
See http://creativecommons.org/licenses/by-nc-sa/3.0/us/ for details
What’s different?

Data-intensive vs. Compute-intensive
Focus on *data-parallel* abstractions

Coarse-grained vs. Fine-grained parallelism
Focus on *coarse-grained data-parallel* abstractions
Logical vs. Physical

Different levels of design:
“Logical” deals with abstract organizations of computing
“Physical” deals with how those abstractions are realized

Examples:
Scheduling
Operators
Data models
Network topology

Why is this important?
Roots in Functional Programming

Simplest data-parallel abstraction
Process a large number of records: “do” something to each

Map

We need something more for sharing partial results across records!
MapReduce = Functional programming + distributed computing!
Functional Programming in Scala

```scala
scala> val t = Array(1, 2, 3, 4, 5)
t: Array[Int] = Array(1, 2, 3, 4, 5)

scala> t.map(n => n*n)
res0: Array[Int] = Array(1, 4, 9, 16, 25)

scala> t.map(n => n*n).foldLeft(0)((m, n) => m + n)
res1: Int = 55
```

Imagine parallelizing the map and fold across a cluster...
A Data-Parallel Abstraction

Process a large number of records

Map “Do something” to each

Group intermediate results

“Aggregate” intermediate results

Reduce

Write final results

Key idea: provide a functional abstraction for these two operations
Waterloo is a city in Ontario, Canada. It is the smallest of three cities in the Regional Municipality of Waterloo (and previously in Waterloo County, Ontario), and is adjacent to the city of Kitchener.

MapReduce “word count” example

Map

- (waterloo, 1)
- (is, 1)
- (a, 1)
- (smallest, 1)
- (of, 1)
- (three, 1)
- (municipality, 1)
- (county, 1)
- (ontario, 1)

Group by key

- (waterloo, [1, 1, 1])
- (is, [1])
- (smallest, [1])
- (of, [1, 1])
- (municipality, [1])
- (county, [1])
- (a, 1)
- (three, [1])
- (ontario, [1])

Reduce

- (waterloo, 3)
- (is, 1)
- (smallest, 1)
- (of, 2)
- (municipality, 1)
- (county, 1)
- (a, 1)
- (three, 1)
- (ontario, 1)

Big document
MapReduce “word count” pseudo-code

def map(key: Long, value: String) = {
    for (word <- tokenize(value)) {
        emit(word, 1)
    }
}

def reduce(key: String, values: Iterable[Int]) = {
    for (value <- values) {
        sum += value
    }
    emit(key, sum)
}
MapReduce

Programmer specifies two functions:

- **map** \((k_1, v_1) \rightarrow \text{List}[(k_2, v_2)]\)
- **reduce** \((k_2, \text{List}[v_2]) \rightarrow \text{List}[(k_3, v_3)]\)

All values with the same key are sent to the same reducer

*What does this actually mean?*

The execution framework handles everything else...
group values by key

map

map

map

map

reduce

reduce

reduce

r₁ s₁

r₂ s₂

r₃ s₃
MapReduce

Programmer specifies two functions:

map \( (k_1, v_1) \rightarrow \text{List}[(k_2, v_2)] \)
reduce \( (k_2, \text{List}[v_2]) \rightarrow \text{List}[(k_3, v_3)] \)

All values with the same key are sent to the same reducer

The execution framework handles everything else...

What’s “everything else”? 
MapReduce “Runtime”

Handles scheduling
Assigns workers to map and reduce tasks

Handles “data distribution”
Moves processes to data

Handles synchronization
Groups intermediate data

Handles errors and faults
Detects worker failures and restarts

Everything happens on top of a distributed FS (later)
MapReduce

Programmer specifies two functions:

\[
\text{map} \ (k_1, v_1) \rightarrow \text{List}[(k_2, v_2)] \\
\text{reduce} \ (k_2, \text{List}[v_2]) \rightarrow \text{List}[(k_3, v_3)]
\]

All values with the same key are sent to the same reducer

The execution framework handles everything else...

Not quite...
What’s the most complex and slowest operation here?
MapReduce

Programmer specifies two functions:

- **map** \( (k_1, v_1) \rightarrow \text{List}[(k_2, v_2)] \)
- **reduce** \( (k_2, \text{List}[v_2]) \rightarrow \text{List}[(k_3, v_3)] \)

All values with the same key are sent to the same reducer

- **partition** \( (k', p) \rightarrow 0 \ldots p-1 \)
  Often a simple hash of the key, e.g., \( \text{hash}(k') \mod n \)
  Divides up key space for parallel reduce operations

- **combine** \( (k_2, \text{List}[v_2]) \rightarrow \text{List}[(k_2, v_2)] \)
  Mini-reducers that run in memory after the map phase
  Used as an optimization to reduce network traffic
group values by key

Important detail: reducers process keys in sorted order
MapReduce can refer to...

The programming model
The execution framework (aka “runtime”)
The specific implementation

Usage is usually clear from context!
MapReduce Implementations

Google has a proprietary implementation in C++

Bindings in Java, Python

Hadoop provides an open-source implementation in Java

Development begun by Yahoo, later an Apache project
Used in production at Facebook, Twitter, LinkedIn, Netflix, ...
Large and expanding software ecosystem
Potential point of confusion: Hadoop is more than MapReduce today

Lots of custom research implementations
Tackling Big Data
Logical View

\[
\begin{array}{cccc}
  k_1 & v_1 & k_2 & v_2 \\
  k_3 & v_3 & k_4 & v_4 \\
  k_5 & v_5 & k_6 & v_6 \\
\end{array}
\]

map \quad map \quad map \quad map

\[
\begin{array}{cccc}
  a & 1 & b & 2 \\
  c & 3 & c & 6 \\
  a & 5 & c & 2 \\
  b & 7 & c & 8 \\
\end{array}
\]

combine \quad combine \quad combine \quad combine

\[
\begin{array}{cccc}
  a & 1 & b & 2 \\
  c & 9 \\
  a & 5 & c & 2 \\
  b & 7 & c & 8 \\
\end{array}
\]

partition \quad partition \quad partition

**group values by key**

\[
\begin{array}{cccc}
  a & 1 & 5 \\
  b & 2 & 7 \\
  c & 2 & 9 & 8 \\
\end{array}
\]

reduce \quad reduce \quad reduce

\[
\begin{array}{cccc}
  r_1 & s_1 \\
  r_2 & s_2 \\
  r_3 & s_3 \\
\end{array}
\]

* Important detail: reducers process keys in sorted order
Physical View

1. **Submit**: The user program is submitted to the master node.
2. **Schedule Map**: The master schedules the map tasks to the worker nodes.
3. **Read**: The worker reads the input files.
4. **Local Write**: The worker writes data to the local disk.
5. **Remote Read**: The worker reads intermediate files from remote disks.
6. **Write**: The worker writes the final output.

Adapted from (Dean and Ghemawat, OSDI 2004)
The datacenter is the computer!
The datacenter *is* the computer!

It’s all about the right level of abstraction
Moving beyond the von Neumann architecture
What’s the “instruction set” of the datacenter computer?

Hide system-level details from the developers
No more race conditions, lock contention, etc.
No need to explicitly worry about reliability, fault tolerance, etc.

Separating the *what* from the *how*
Developer specifies the computation that needs to be performed
Execution framework (“runtime”) handles actual execution
The datacenter *is* the computer! “Big ideas”

Scale “out”, not “up” *
Limits of SMP and large shared-memory machines

Assume that components will break
Engineer software around hardware failures

Move processing to the data *
Cluster have limited bandwidth, code is a lot smaller

Process data sequentially, avoid random access
Seeks are expensive, disk throughput is good
Seek vs. Scans

Consider a 1 TB database with 100 byte records

We want to update 1 percent of the records

Scenario 1: Mutate each record
Each update takes $\sim 30$ ms (seek, read, write)
$10^8$ updates = $\sim 35$ days

Scenario 2: Rewrite all records
Assume 100 MB/s throughput
Time = 5.6 hours(!)

Lesson? Random access is expensive!

Source: Ted Dunning, on Hadoop mailing list
So you want to drive the elephant!
A tale of two packages...

org.apache.hadoop.mapreduce
org.apache.hadoop.mapred

Source: Wikipedia (Budapest)
MapReduce API*

Mapper<\(K_{in}, V_{in}, K_{out}, V_{out}\)>

void setup(Mapper.Context context)
Called once at the start of the task
void map(K_{in} key, V_{in} value, Mapper.Context context)
Called once for each key/value pair in the input split
void cleanup(Mapper.Context context)
Called once at the end of the task

Reducer<\(K_{in}, V_{in}, K_{out}, V_{out}\)>/
Combiner<\(K_{in}, V_{in}, K_{out}, V_{out}\)>

void setup(Reducer.Context context)
Called once at the start of the task
void reduce(K_{in} key, Iterable<\(V_{in}\)> values, Reducer.Context context)
Called once for each key
void cleanup(Reducer.Context context)
Called once at the end of the task

*Note that there are two versions of the API!
MapReduce API*

Partitioner<K, V>

int getPartition(K key, V value, int numPartitions)

Returns the partition number given total number of partitions

Job

Represents a packaged Hadoop job for submission to cluster
Need to specify input and output paths
Need to specify input and output formats
Need to specify mapper, reducer, combiner, partitioner classes
Need to specify intermediate/final key/value classes
Need to specify number of reducers (but not mappers, why?)
Don’t depend on defaults!

*Note that there are two versions of the API!
Data Types in Hadoop: Keys and Values

Writable

- Defines a de/serialization protocol.
- Every data type in Hadoop is a Writable.

WritableComparable

- Defines a sort order.
- All keys must be of this type (but not values).

IntWritable
LongWritable
Text
...

Concrete classes for different data types.
Note that these are container objects.

SequenceFile

- Binary-encoded sequence of key/value pairs.


```

def map(key: Long, value: String) = {
    for (word <- tokenize(value)) {
        emit(word, 1)
    }
}

def reduce(key: String, values: Iterable[Int]) = {
    for (value <- values) {
        sum += value
    }
    emit(key, sum)
}
```

"""Hello World"" MapReduce: Word Count
private static final class MyMapper
  extends Mapper<LongWritable, Text, Text, IntWritable> {

  private final static IntWritable ONE = new IntWritable(1);
  private final static Text WORD = new Text();

  @Override
  public void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
    for (String word : Tokenizer.tokenize(value.toString())) {
      WORD.set(word);
      context.write(WORD, ONE);
    }
  }
}
private static final class MyReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

    private final static IntWritable SUM = new IntWritable();

    @Override
    public void reduce(Text key, Iterable<IntWritable> values,
            Context context) throws IOException,
            InterruptedException {
        Iterator<IntWritable> iter = values.iterator();
        int sum = 0;
        while (iter.hasNext()) {
            sum += iter.next().get();
        }
        SUM.set(sum);
        context.write(key, SUM);
    }
}
Getting Data to Mappers and Reducers

Configuration parameters
Pass in via Job configuration object

“Side data”
DistributedCache
Mappers/Reducers can read from HDFS in setup method
Complex Data Types in Hadoop

How do you implement complex data types?

The easiest way:

Encode it as Text, e.g., (a, b) = “a:b”
Use regular expressions to parse and extract data
Works, but janky

The hard way:

Define a custom implementation of Writable(Comparable)
Must implement: readFields, write, (compareTo)
Computationally efficient, but slow for rapid prototyping
Implement WritableComparator hook for performance

Somewhere in the middle:
Bespin (via lin.tl) offers various building blocks
Anatomy of a Job

Hadoop MapReduce program = Hadoop job

Jobs are divided into map and reduce tasks
An instance of a running task is called a task attempt
Each task occupies a slot on the tasktracker
Multiple jobs can be composed into a workflow

Job submission:
Client (i.e., driver program) creates a job,
configures it,
and submits it to jobtracker

That’s it! The Hadoop cluster takes over...
Adapted from (Dean and Ghemawat, OSDI 2004)
Anatomy of a Job

Behind the scenes:

Input splits are computed (on client end)
Job data (jar, configuration XML) are sent to jobtracker
Jobtracker puts job data in shared location, enqueues tasks
Tasktrackers poll for tasks
Off to the races...
Where’s the data actually coming from?
Input and Output

InputFormat
   TextInputFormat
   KeyValueTextInputFormat
   SequenceFileInputFormat
   ...

OutputFormat
   TextOutputFormat
   SequenceFileOutputFormat
   ...

Spark also uses these abstractions for reading and writing data!
Hadoop Workflow

Getting data in?  
Writing code?  
Getting data out?

Where’s the actual data stored?
Debugging Hadoop

First, take a deep breath
Start small, start locally
Build incrementally
Code Execution Environments

Different ways to run code:

Local (standalone) mode
Pseudo-distributed mode
Fully-distributed mode

Learn what’s good for what
Hadoop Debugging Strategies

Good ol’ System.out.println
Learn to use the webapp to access logs
Logging preferred over System.out.println
Be careful how much you log!

Fail on success
Throw RuntimeExceptions and capture state

Use Hadoop as the “glue”
Implement core functionality outside mappers and reducers
Independently test (e.g., unit testing)
Compose (tested) components in mappers and reducers
Questions?