CompSci 516
Data Intensive Computing Systems

Lecture 3
Map-Reduce
and
Spark

Guest Lecturer: Junghoon Kang
Reading Material

- Recommended (optional) readings:
  - Chapter 2 (Sections 1,2,3) of Mining of Massive Datasets, by Rajaraman and Ullman: [http://i.stanford.edu/~ullman/mmds.html](http://i.stanford.edu/~ullman/mmds.html)
  - MapReduce: Simplified Data Processing on Large Cluster - Jeffrey Dean, et al. – 2004
Announcement

● Jung - I have switched my office hours from Thursdays 1pm - 2pm to Thursdays 3pm - 4pm at the same location, N303B.

● In HW2, you will be writing Spark applications and run them on AWS EC2 instances.
Google MapReduce

CompSci 516
Junghoon Kang
Big Data

it cannot be stored in one hard disk drive

need to split it into multiple machines

Google File System

it cannot be processed by one CPU

parallelize computation on multiple machines

MapReduce

Today!
Where does Google use MapReduce?

Input

MapReduce

Output

- crawled documents
- web request logs
- inverted indices
- graph structure of web documents
- summaries of the number of pages crawled per host
- the set of most frequent queries in a day
What is MapReduce?

It is a programming model that processes large data by:

- apply a function to each logical record in the input (map)
- categorize and combine the intermediate results into summary values (reduce)
Understanding MapReduce
(example by Yongho Ha)

I am a class president
An English teacher asks you:

“Could you count the number of occurrences of each word in this book?”
Um... Ok...
Let's divide the workload among classmates.
And let few combine the intermediate results. reduce

cloud 1
data 1

parallel 1
data 1
computer 1

map 1
cloud 1
parallel 1

computer 1
map 1
scientist 1

reduce 1
map 1
scientist 1

I will collect from A ~ G

H ~ Q

R ~ Z

reduce 1

map 3

parallel 2

scientist 2

cloud 2
computer 2
data 2

borrowed slide
Why did MapReduce become so popular?
Is it because Google uses it?
Distributed Computation
Before MapReduce

Things to consider:

● how to divide the workload among multiple machines?
● how to distribute data and program to other machines?
● how to schedule tasks?
● what happens if a task fails while running?
● ... and ... and ...
Distributed Computation
After MapReduce

Things to consider:

● how to write **Map** function?
● how to write **Reduce** function?
MapReduce has made distributed computation an easy thing to do!

Developers needed before MapReduce

Developers needed after MapReduce
Given the brief intro to MapReduce,

let’s begin our journey to real implementation details in MapReduce!
Key Players in MapReduce

One Master
- coordinates many workers.
- assigns a task* to each worker.
  (* task = partition of data + computation)

Multiple Workers
- Follow whatever the Master asks to do.
1. The MapReduce library in the user program first splits the input file into $M$ pieces.
2. The MapReduce library in the user program then starts up many copies of the program on a cluster of machines: one master and multiple workers.
There are $M$ map tasks and $R$ reduce tasks to assign.

(The figures below depicts task = data + computation)
3. The master picks idle workers and assigns each one a map task.
4. Map Phase (each mapper node)

1) Read in a corresponding input partition.

2) Apply the user-defined map function to each key/value pair in the partition.

3) Partition the result produced by the map function into \( R \) regions using the partitioning function.

4) Write the result into its local disk (not GFS).

5) Notify the master with the locations of each partitioned intermediate result.
Map Phase

1. assign map task

Google File System

2. where is my partition

3. here is your input partition

Inside kth map task

4. here are the locations of partitioned intermediate results

master  mapper

partition_k

map function

hash (mod R)

temp_k1  temp_k2  \ldots  temp_kR
5. After all the map tasks are done, the master picks idle workers and assigns each one a reduce task.
6. Reduce Phase (each reducer node)

1) Read in all the corresponding intermediate result partitions from mapper nodes.

2) Sort the intermediate results by the intermediate keys.

3) Apply the user-defined reduce function on each intermediate key and the corresponding set of intermediate values.

4) Create one output file.
Reduce Phase

1. assign reduce task
2. send intermediate result to this reducer
3. here are your intermediate result partitions
4. store the output file into GFS (reduce phase will generate the total of R output files)

Google File System
Fault Tolerance

Although the probability of a machine failure is low, the probability of a machine failing among thousands of machines is common.
How does MapReduce handle machine failures?

Worker Failure
- The master sends heartbeat to each worker node.
- If a worker node fails, the master reschedules the tasks handled by the worker.

Master Failure
- The whole MapReduce job gets restarted through a different master.
Locality

- The input data is managed by GFS.
- Choose the cluster of MapReduce machines such that those machines contain the input data on their local disk.
- We can conserve network bandwidth.
Task Granularity

- It is preferable to have the number of tasks to be multiples of worker nodes.
- Smaller the partition size, faster failover and better granularity in load balance.

But it incurs more overhead. Need a balance.
Backup Tasks

- In order to cope with a straggler, the master schedules backup executions of the remaining *in-progress* tasks.
MapReduce Pros and Cons

- MapReduce is **good** for off-line batch jobs on large data sets.
- MapReduce is **not good** for iterative jobs due to high I/O overhead as each iteration needs to read/write data from/to GFS.
- MapReduce is **bad** for jobs on small datasets and jobs that require low-latency response.
Apache Hadoop

Apache Hadoop is an open-source version of GFS and Google MapReduce.

<table>
<thead>
<tr>
<th></th>
<th>Google</th>
<th>Apache Hadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>File System</td>
<td>GFS</td>
<td>HDFS</td>
</tr>
<tr>
<td>Data Processing</td>
<td>Google MapReduce</td>
<td>Hadoop MapReduce</td>
</tr>
<tr>
<td>Engine</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
References

● MapReduce: Simplified Data Processing on Large Cluster - Jeffrey Dean, et al. - 2004
● http://www.slideshare.net/yongho/2011-h3
Apache Spark

CompSci 516
Junghoon Kang
About Spark

- Spark is a distributed large-scale data processing engine that exploits in-memory computation and other optimizations.

- One of the most popular data processing engine in the industry these days; many large companies like Netflix, Yahoo, and eBay use Spark at massive scale.
More about Spark

- It started as a research project at UC Berkeley.
- Published the Resilient Distributed Datasets (RDD) paper in NSDI 2012.
- Best Paper award that year.
Motivation

Hadoop MapReduce indeed made analyzing large datasets easy. However, MapReduce was still not good for:

- iterative jobs, such as machine learning and graph computation
- interactive and ad-hoc queries
Can we do better?

The reason why MapReduce is not good for iterative jobs is because of the high I/O overhead as each iteration needs to read/write data from/to HDFS.

So, what if we use RAM between each iteration?
Instead of storing intermediate outputs into HDFS, using RAM would be faster.
Instead of reading input from HDFS every time you run query, bring the input into RAM first then run multiple queries.
Challenge

But RAM is a volatile storage…
What happens if a machine faults?

Although the probability of a machine failure is low, the probability of a machine failing among thousands of machines is common.

In other words, how can we create an efficient, fault-tolerant, and distributed RAM storage?
Some data processing frameworks, such as RAMCloud or Piccolo, also used RAM to improve the performance.

And they supported fine-grained update of data in RAM.

But it is hard to achieve fault tolerance with fine-grained update and good performance.
Spark’s Approach

What if we use RAM as read-only?

This idea is RDD, Resilient Distributed Datasets!

Which is the title of the Spark paper and the core idea behind Spark!
Resilient Distributed Datasets

What are the properties of RDD?
● read-only, partitioned collections of records
● you can only create RDD from input files in a storage or RDD

What’s good about RDD again?
● RDD is read-only (immutable). Thus, it hasn’t been changed since it got created.
● That means we can create it again if we record how it is created.
● So, if we just record how the RDD got created from its parent RDD (lineage), it becomes fault-tolerant!
How do you code in Spark?

Coding in Spark is creating a **lineage of RDDs** in a directed acyclic graph (DAG) form.

```scala
rdd1 = sc.textFile("input1.txt")
rdd2 = sc.textFile("input2.txt")
rdd3 = rdd1.join(rdd2).map(...)  # ...
ret = rdd3.collect()
```
## RDD Operators (two types)

### Transformations & Actions

<table>
<thead>
<tr>
<th>Transformations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>map(f : T \rightarrow U)</td>
<td>(RDD[T] \rightarrow RDD[U])</td>
</tr>
<tr>
<td>filter(f : T \rightarrow \text{Bool})</td>
<td>(RDD[T] \rightarrow RDD[T])</td>
</tr>
<tr>
<td>flatMap(f : T \Rightarrow \text{Seq}[U])</td>
<td>(RDD[T] \Rightarrow RDD[U])</td>
</tr>
<tr>
<td>sample(\text{fraction} : \text{Float})</td>
<td>(RDD[T] \Rightarrow RDD[T]) \ (Deterministic sampling)</td>
</tr>
<tr>
<td>groupByKey()</td>
<td>(RDD[(K, V)] \Rightarrow RDD[(K, \text{Seq}[V])])</td>
</tr>
<tr>
<td>reduceByKey(f : (V, V) \Rightarrow V)</td>
<td>(RDD[(K, V)] \Rightarrow RDD[(K, V)])</td>
</tr>
<tr>
<td>union()</td>
<td>(RDD[(K, V)], RDD[(K, W)] \Rightarrow RDD[(K, (V, W))])</td>
</tr>
<tr>
<td>join()</td>
<td>(RDD[(K, V)], RDD[(K, W)] \Rightarrow RDD[(K, (V, W))])</td>
</tr>
<tr>
<td>cogroup()</td>
<td>(RDD[(K, V)], RDD[(K, W)] \Rightarrow RDD[(K, (\text{Seq}[V], \text{Seq}[W]))])</td>
</tr>
<tr>
<td>crossProduct()</td>
<td>(RDD[T], RDD[U]) \Rightarrow RDD[(T, U)])</td>
</tr>
<tr>
<td>mapValues(f : V \Rightarrow W)</td>
<td>(RDD[(K, V)] \Rightarrow RDD[(K, W)]) \ (Preserves partitioning)</td>
</tr>
<tr>
<td>sort(c : \text{Comparator}[K])</td>
<td>(RDD[(K, V)] \Rightarrow RDD[(K, V)])</td>
</tr>
<tr>
<td>partitionBy(p : \text{Partitioner}[K])</td>
<td>(RDD[(K, V)] \Rightarrow RDD[(K, V)])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count()</td>
<td>(RDD[T] \Rightarrow \text{Long})</td>
</tr>
<tr>
<td>collect()</td>
<td>(RDD[T] \Rightarrow \text{Seq}[T])</td>
</tr>
<tr>
<td>reduce(f : (T, T) \Rightarrow T)</td>
<td>(RDD[T] \Rightarrow T)</td>
</tr>
<tr>
<td>lookup(k : K)</td>
<td>(RDD[(K, V)] \Rightarrow \text{Seq}[V]) \ (On hash/range partitioned RDDs)</td>
</tr>
<tr>
<td>save(\text{path} : \text{String})</td>
<td>Outputs RDD to a storage system, \textit{e.g.}, HDFS</td>
</tr>
</tbody>
</table>

Table 2: Transformations and actions available on RDDs in Spark. \(\text{Seq}[T]\) denotes a sequence of elements of type \(T\).
Lazy Execution

- Transformation functions simply creates a lineage of RDDs.
- An action function that gets called in the end triggers the computation of the whole lineage of transformation functions and outputs the final value.
Two Types of Dependencies

Narrow Dependencies:
- map, filter
- union

Wide Dependencies:
- groupByKey
- 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.
Narrow Dependency

- The task can be done in one node.
- No need to send data over network to complete the task.
- Fast.
Wide Dependency

- The task needs shuffle.
- Need to pull data from other nodes via network.
- Slow.
- Use wide dependencies wisely.
Job Scheduling

- One job contains one action function and possibly many transformation functions.
- A job is represented by the DAG of RDDs.
- Compute the job following the DAG.
- New stage gets created if a RDD requires shuffle from an input RDD.

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.
Task Distribution

- Similar to MR
- One master, multiple workers
- One RDD is divided into multiple partitions

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.
How fast is Spark?

- Skip the first iteration, since it’s just text parsing.

- In later iterations, Spark is much faster (black bar).

- HadoopBM writes intermediate data in memory not HDFS.

Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.
What if the number of nodes increases?

Figure 8: Running times for iterations after the first in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB.
Apache Spark Ecosystem

Spark SQL + DataFrames
Streaming
MLlib Machine Learning
GraphX Graph Computation

Spark Core API
R
SQL
Python
Scala
Java
References


● https://databricks.com/spark/about
● http://www.slideshare.net/yongho/rdd-paper-review
● https://www.youtube.com/watch?v=dmL0N3qfSc8
● http://www.tothenew.com/blog/spark-1o3-spark-internals/