Agenda

• Lambda Architecture
• Spark Internals
• Spark on Bluemix
• Spark Education
• Spark Demos
Lambda Architecture

Is a data-processing architecture designed to handle massive quantities of data by taking advantage of both batch and stream processing methods.

• Spark - one of the few, if not the only, data processing framework that allows you to have both batch and stream processing of terabytes of data in the same application.
Lambda Architecture - Spark
Lambda Architecture Layers

- **Batch Layer**
  - Managing the master dataset, an immutable, append only set of raw data
  - Pre-computing arbitrary query functions, called batch views

- **Serving Layer**
  - Indexes batch views so they can be queried in adhoc with low latency

- **Speed Layer**
  - Accommodates all requests that are subject to low latency requirements. Using fast and incremental algorithms, deals with recent data only
Spark Internals - Streaming

- The input stream (DStream) goes into Spark Streaming
- Breaks up into batches
- Feeds into the Spark engine for processing
- Generate the final results in streams of batches

- Sliding window operations
  - Windowed computations
    - Window length
    - Sliding interval
    - reduceByKeyAndWindow
Strategy

• Partition For Scale
• Replicate For Resiliency
• Share Nothing
• Asynchronous Message Passing
• Parallelism
• Isolation
• Location Transparency
What We Need

- Fault Tolerant
- Failure Detection
- Fast - low latency, distributed, data locality
- Masterless, Decentralized Cluster Membership
- Span Racks and Datacenters
- Hashes The Node Ring
- Partition-Aware
- Elasticity
- Asynchronous - message-passing system
- Parallelism
- Network Topology Aware
Spark Cluster

Components:
- Driver
- Cluster Master
- Executors

Yarn App Master

Yarn App Containers

Worker Node
- Executor
- Cache
- Task
- Task
RDDs: Resilient Distributed Dataset

Spark’s basic unit of data

RDDs are immutable, distributed, and fault-tolerant

Two types of operations:

- Transformations:
  - Data lineage DAG (Directed Acyclic Graph)
  - Single run with many stages, versus multiple jobs with MR
  - Lazy evaluations

- Actions:
  - Performs transformations and action
  - Returns a value
  - Reusable

Fault tolerance: If data in memory is lost it will be recreated from lineage

Caching, persistence (memory, spilling, disk), and check-pointing

- Data is distributed into partitions spread across a cluster
- Each partition is processed independently and in parallel
- Logical view of the data – not materialized
DAG: Distributed Acyclic Graph

An example of a workload consists of five MR jobs

Spark DAG and Action
Spark Data Model

Resilient Distributed Dataset (RDD)
A collection:
• Immutable
• Iterable
• Serializable
• Distributed
• Parallel
• Lazy
Check Pointing

Allows saving enough of information to a fault-tolerant storage to allow the RDDs
- **Metadata** - the information defining the streaming computation
- Data (RDDs)

Usage
- With `updateStateByKey`, `reduceByKeyAndWindow` – stateful transformations
- To **recover from failures** in Spark Streaming apps

Can affect performance, depending on
- The **data** and or **batch sizes**
- The speed of the file system that is being used for checkpointing
Data Frames

Cheap!
- No serialization
- No IO
- Pipelined

Expensive!
- Serialize Data
- Write to disk
- Transfer over network
- Deserialize Data
Compare to MapReduce Word Count

### Hadoop MapReduce

```java
public static class WordCountMapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
        String line = value.toString();
        StringTokenizer itr = new StringTokenizer(line);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, one);
        }
    }
}
```

```java
public static class WordCountReduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```

### Spark

```scala
spark = new SparkContext(master, appName, [sparkHome], [jars])
file = spark.textFile("hdfs://...")
counts = file.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _).saveAsTextFile("hdfs://...")
```
Using Partitioners for Narrow Joins

**Advantages**

- Write data to hdfs
- Read from hdfs
- “Remember” data was written with a partitioner
Partitions, Partitions, Partitions

Partitions should be small

- Max partition size is 2GB*
- Small partitions help deal w/ stragglers
- Small partitions avoid overhead – take a closer look at internals …

Partitions should be big

- “For ML applications, the best setting to set the number of partitions to match the number of cores to reduce shuffle size.”
What data and where is it going?

- Narrow Dependencies (aka “OneToOneDependency”)
  - cheap

- Wide Dependencies (aka “shuffles”)
  - how much is shuffled
  - Is it skewed

- Driver bottleneck
Stages are not MapReduce Steps!

MapReduce Steps:
- Map
- Reduce
- Shuffle
- Map
- Reduce
- Shuffle
- ReduceByKey
- (mapside combine)
- Shuffle
- GroupByKey
- Collect
- FlatMap
- ReduceByKey
- Filter
- Map
Key Question

- How does a user program get translated into units of physical execution: jobs, stages, and tasks?
RDD API Refresher

RDDs are a distributed collection of records

- rdd = spark.parallelize(range(10000), 10)

*Transformations* create new RDDs from existing ones

- errors = rdd.filter(lambda line: “ERROR” in line)

*Actions* materialize a value in the user program

- size = errors.count()
RDD API Example

// Read input file
val input = sc.textFile("input.txt")

val tokenized = input
  .map(line => line.split(" "))
  .filter(words => words.size > 0) // remove empty lines

val counts = tokenized // frequency of log levels
  .map(words => (words(0), 1))
  .reduceByKey{ (a, b) => a + b, 2 }
RDD API Example

// Read input file
val input = sc.textFile(

val tokenized = input
.map(
.filter(

val counts = tokenized
.map(
.reduceByKey{
}
Transformations

\[
\text{sc.textFile().map().filter().map().reduceByKey()}
\]
DAG View of RDD’s

- **textFile()**
- **map()**
- **filter()**
- **map()**
- **reduceByKey()**

<table>
<thead>
<tr>
<th>Hadoop RDD</th>
<th>Mapped RDD</th>
<th>FilteredRDD</th>
<th>Mapped RDD</th>
<th>Shuffle RDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition 1</td>
<td>Partition 1</td>
<td>Partition 1</td>
<td>Partition 1</td>
<td>Partition 1</td>
</tr>
<tr>
<td>Partition 2</td>
<td>Partition 2</td>
<td>Partition 2</td>
<td>Partition 2</td>
<td>Partition 2</td>
</tr>
<tr>
<td>Partition 3</td>
<td>Partition 3</td>
<td>Partition 3</td>
<td>Partition 3</td>
<td>Partition 3</td>
</tr>
</tbody>
</table>

**Input**: tokenized

**Counts**:
Evaluation of the DAG

DAG’s are materialized through a method sc.runJob: def runJob[T, U](
rdd: RDD[T],

partitions: Seq[Int],

func: (Iterator[T]) => U)

: Array[U]

1. RDD to compute
2. Which partitions
3. Fn to produce results
→ results for each part
How runJob Works

• Needs to compute my parents, parents, parents etc. all the way back to an RDD with no dependencies (e.g. HadoopRDD)
Physical Optimizations

1. Certain types of transformations can be pipelined

2. If dependent RDD’s have already been cached (or persisted in a shuffle) the graph can be truncated

3. Once pipelining and truncation occur, Spark produces a set of stages each stage is composed of tasks
How runJob Works

- Needs to compute my parents, parents parents etc., all the way back to an RDD with no dependencies (e.g. HadoopRDD)
How runJob Works

- Needs to compute my parents, parents parents etc., all the way back to an RDD with no dependencies (e.g. HadoopRDD)
How runJob Works

- Needs to compute my parents, parents parents etc., all the way back to an RDD with no dependencies (e.g. HadoopRDD)
Stage Graph

Each task will:

1. Read Hadoop input
2. Perform maps and filters
3. Write partial sums

Stage 1

- Task 1
- Task 2
- Task 3

Stage 2

- Task 1
- Task 2

Each task will:
1. Read partial sums
2. Invoke user function passed to runJob
Units of Physical Execution

Jobs: Work required to compute RDD in runJob

Stages: A wave of work within a job, corresponding to one or more pipelined RDD’s

Tasks: A unit of work within a stage, corresponding to one RDD partition

Shuffle: The transfer of data between stages
Seeing this on your own

scala> counts.toDebugString

res84: String =

(2) ShuffledRDD[296] at reduceByKey at <console>:17

+- (3) MappedRDD[295] at map at <console>:17

| FilteredRDD[294] at filter at <console>:15

| MappedRDD[293] at map at <console>:15

| input.text MappedRDD[292] at textFile at <console>:13

| input.text HadoopRDD[291] at textFile at <console>:13

(indentations indicate a shuffle boundary)
Example: count() action

class RDD {
    def count(): Long = {
        results = sc.runJob(
            this,
            0 until partitions.size,
            it => it.size()
        )
        return results.sum
    }
}

1. RDD = self
2. Partitions = all partitions
3. Function = size of the partition
Example: take(N) action

class RDD {
    def take(n: Int) {
        val results = new ArrayBuffer[T]
        var partition = 0
        while (results.size < n) {
            result += sc.runJob(this, partition, it => it.toArray)
            partition = partition + 1
        }
        return results.take(n)
    }
}
Putting it All Together

Named after action calling runJob

Named after last RDD in pipeline
Internals of the RDD Interface

1) List of partitions
2) Set of dependencies on parent RDDs
3) Function to compute a partition, given parents
4) Optional partitioning info for k/v RDDs (Partitioner)

This captures all current Spark operations!
Example: Hadoop RDD

Partitions  = 1 per HDFS block

Dependencies  = None

compute(partition) = read corresponding HDFS block

Partitioner = None

> rdd = spark.hadoopFile("hdfs://click_logs/")
Example: Filtered RDD

Partitions = parent partitions

Dependencies = a single parent

compute(partition) = call parent.compute(partition) and filter

Partitioner = parent partitioner

> filtered = rdd.filter(lambda x: x contains “ERROR”)

This captures all current Spark operations!
Example: Joined RDD

- **Partitions** = number chosen by user or heuristics
- **Dependencies** = ShuffleDependency on two or more parents
- **compute(partition)** = read and join data from all parents
- **Partitioner** = HashPartitioner(# partitions)

This captures all current Spark operations!
A More Complex DAG

- Hadoop RDD
  - Partition 1
  - Partition 2
- JDBC RDD
  - Partition 1
  - Partition 2
- Filtered RDD
  - Partition 1
  - Partition 2
- Mapped RDD
  - Partition 1
  - Partition 2
- Joined RDD
  - Partition 1
  - Partition 2
  - Partition 3
- Filtered RDD
  - Partition 1
  - Partition 2
  - Partition 3

.count()
A More Complex DAG – cont’d

-- Diagram --

Stage 1
- Task 1
- Task 2

Stage 2
- Task 1
- Task 2

Stage 3
- Task 1
- Task 2
- Task 3

Shuffle Write
Shuffle Read
Narrow and Wide Transformations

FilteredRDD

Parent
Partition 1
Partition 2
Partition 3

RDD

Partition 1
Partition 2
Partition 3

JoinedRDD

Parent 1
Partition 1
Partition 2

Parent 2
Partition 1
Partition 2

RDD

Partition 1
Partition 2
Partition 3
Spark Resources

- Spark Related Books
- External Links
Resources: Books

*Programming Scala*
Dean Wampler, Alex Payne
O’Reilly (2009)

*Fast Data Processing with Spark*
Holden Karau
Packt (2013)

*Spark in Action*
Chris Fregly
Manning (2015)
[sparkinaction.com](http://sparkinaction.com)

*Learning Spark*
Holden Karau, Andy Kowinski, Matei Zaharia
O’Reilly (2015)

*Advanced Analytics with Spark*
Sandy Rees, Uri Laserver, Sean Owen & Josh Whi
Resources: External Links

Apache Spark Central:
- apachesparkcentral.com
- Independent, Great Source of Spark Knowledge
- Feared by Databricks

Databricks Spark Hub:
- forums.databricks.com
- databricks.com/blog: High-level, Partner Cross-Post
- databricks.com/spark/developer-resources
- spark-packages.org

Contribute to Spark and related OSS projects via the email lists:
- user@spark.apache.org – usage questions, help, announcements
- dev@spark.apache.org – for people who want to contribute code
Upcoming Events

- Spark + Logo
- Launch of Spark on Bluemix
Spark Demos

- [https://datascientistworkbench.com/demo](https://datascientistworkbench.com/demo)
- Personality Index – Resume
- Twitter Search – Mumbai Spark Meetup
THANK YOU