Who am I?

Reynold Xin

Spark PMC member
Databricks cofounder & architect

UC Berkeley AMPLab PhD (on leave)
Agenda

1. MapReduce Review
2. Introduction to Spark and RDDs
3. Generality of RDDs (e.g. streaming, ML)
4. DataFrames
5. Internals (time permitting)
Agenda

1. MapReduce Review
2. Introduction to Spark and RDDs
3. Generality of RDDs (e.g. streaming, ML)
4. DataFrames
5. Internals (time permitting)
Google Datacenter

How do we program this thing?
Traditional Network Programming

Message-passing between nodes (MPI, RPC, etc)

Really hard to do at scale:

• How to split problem across nodes?
  – Important to consider network and data locality

• How to deal with failures?
  – If a typical server fails every 3 years, a 10,000-node cluster sees 10 faults/day!

• Even without failures: stragglers (a node is slow)

Almost nobody does this!
Data-Parallel Models

Restrict the programming interface so that the system can do more automatically

“Here’s an operation, run it on all of the data”
  • I don’t care *where* it runs (you schedule that)
  • In fact, feel free to run it *twice* on different nodes
MapReduce Programming Model

Data type: key-value records

Map function:

$$ (K_{in}, V_{in}) \rightarrow \text{list}(K_{inter}, V_{inter}) $$

Reduce function:

$$ (K_{inter}, \text{list}(V_{inter})) \rightarrow \text{list}(K_{out}, V_{out}) $$
MapReduce Programmability

Most real applications require multiple MR steps
  • Google indexing pipeline: 21 steps
  • Analytics queries (e.g. count clicks & top K): 2 – 5 steps
  • Iterative algorithms (e.g. PageRank): 10’s of steps

Multi-step jobs create spaghetti code
  • 21 MR steps -> 21 mapper and reducer classes
  • Lots of boilerplate code per step
MapReduce: A major step backwards

By David DeWitt on January 17, 2008 4:20 PM | Permalink | Comments (44) | TrackBacks (1)
[Note: Although the system attributes this post to a single author, it was written by David J. DeWitt and Michael Stonebraker]

On January 8, a Database Column reader asked for our views on new distributed database research efforts, and we’ll begin here with a discussion of one of them, since the recent trade press has been filled with news of the revolution of so-called "cloud computing." This paradigm involves linking together a large number of processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a large number of smaller number of high-end servers.

For example, IBM and Google have announced plans to make a 1,000 processor cluster available to a few select universities to test a software tool called MapReduce [1]. Berkeley has gone so far as to plan on teaching their freshman how to program using the MapReduce language.

As both educators and researchers, we are amazed at the hype that the MapReduce proponents have spread about how it represents a revolution in data-intensive applications. MapReduce may be a good idea for writing certain types of general-purpose computations, but to the extent it is a "new" or "revolutionary" paradigm, we believe it is not.

1. A giant step backward in the programming paradigm for large-scale data intensive applications

2. A sub-optimal implementation, in that it uses brute force instead of indexing

3. Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago

4. Missing most of the features that are routinely included in current DBMS

Problems with MapReduce

MapReduce use cases showed two major limitations:

1. difficulty of programming directly in MR.
2. Performance bottlenecks

In short, MR doesn’t compose well for large applications

Therefore, people built high level frameworks and specialized systems.
Higher Level Frameworks

In reality, 90+% of MR jobs are generated by Hive SQL

A = load 'foo';
B = group A all;
C = foreach B generate COUNT(A);

SELECT count(*) FROM users
Specialized Systems

General Batch Processing

Specialized Systems:
iterative, interactive, streaming, graph, etc.
Agenda

1. MapReduce Review
2. Introduction to Spark and RDDs
3. Generality of RDDs (e.g. streaming, ML)
4. DataFrames
5. Internals (time permitting)
Spark: A Brief History

- 2002: MapReduce @ Google
- 2004: MapReduce paper
- 2006: Hadoop @ Yahoo!
- 2008: Hadoop Summit
- 2010: Spark paper
- 2012
- 2014: Apache Spark top-level
Spark Summary

Unlike the various specialized systems, Spark’s goal was to generalize MapReduce to support new apps.

Two small additions are enough:

- fast data sharing
- general DAGs

More efficient engine, and simpler for the end users.
Spark Ecosystem

BlinkDB
Approximate SQL

Spark
SQL

Spark Streaming
Streaming

MLlib
Machine Learning

GraphX
Graph Computation

Spark R
R on Spark

Spark Core Engine
Note: not a scientific comparison.
Programmability

WordCount in 50+ lines of Java MR

WordCount in 3 lines of Spark
Performance
Time to sort 100TB

2013 Record: Hadoop
2100 machines
72 minutes

2014 Record: Spark
207 machines
23 minutes

Also sorted 1PB in 4 hours

Source: Daytona GraySort benchmark, sortbenchmark.org
RDD: Core Abstraction

Write programs in terms of **distributed datasets**

and **operations** on them

Resilient Distributed Datasets

- Collections of objects spread across a cluster, stored in RAM or on Disk
- Built through parallel transformations
- Automatically rebuilt on failure

Operations

- Transformations (e.g. map, filter, groupBy)
- Actions (e.g. count, collect, save)
Resilient Distributed Datasets are the primary abstraction in Spark – a fault-tolerant collection of elements that can be operated on in parallel.

Two types:

- **parallelized collections** – take an existing single-node collection and parallel it
- **Hadoop datasets**: files on HDFS or other compatible storage
Operations on RDDs

Transformations $f(\text{RDD}) \Rightarrow \text{RDD}$
- Lazy (not computed immediately)
- E.g. “map”

Actions:
- Triggers computation
- E.g. “count”, “saveAsTextFile”
Working With RDDs

textFile = sc.textFile("SomeFile.txt")
Working With RDDs

```
linesWithSpark = textFile.filter(lambda line: "Spark" in line)
textFile = sc.textFile("SomeFile.txt")
```
Working With RDDs

```python
linesWithSpark = textFile.filter(lambda line: "Spark" in line)

textFile = sc.textFile("SomeFile.txt")

df.head()  # Apache Spark
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns.
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()  

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining
Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining
Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t") [2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```

Cache your data ➔ Faster Results

Full-text search of Wikipedia
- 60GB on 20 EC2 machines
- 0.5 sec from mem vs. 20s for on-disk
Language Support

**Python**

```python
lines = sc.textFile(...)
lines.filter(lambda s: “ERROR” in s).count()
```

**Scala**

```scala
val lines = sc.textFile(...)
lines.filter(x => x.contains(“ERROR”)).count()
```

**Java**

```java
JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>() {
    Boolean call(String s) {
        return s.contains(“error”);
    }
}).count();
```

**Standalone Programs**
Python, Scala, & Java

**Interactive Shells**
Python & Scala

**Performance**
Java & Scala are faster due to static typing

…but Python is often fine
Expressive API

map  reduce
Expressive API

map  reduce  sample
filter  count  take
groupBy  fold  first
sort  reduceByKey
union  groupByKey
join  cogroup  partitionBy
leftOuterJoin  cross  mapWith
rightOuterJoin  zip  pipe

save ...
Fault Recovery

RDDs track *lineage* information that can be used to efficiently reconstruct lost partitions

Ex:

```
messages = textFile(...).filter(_.startsWith("ERROR"))
  .map(_.split("\t")(2))
```
Fault Recovery Results

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Iteration time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>119</td>
</tr>
<tr>
<td>2</td>
<td>57</td>
</tr>
<tr>
<td>3</td>
<td>56</td>
</tr>
<tr>
<td>4</td>
<td>58</td>
</tr>
<tr>
<td>5</td>
<td>58</td>
</tr>
<tr>
<td>6</td>
<td>81</td>
</tr>
<tr>
<td>7</td>
<td>57</td>
</tr>
<tr>
<td>8</td>
<td>59</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
</tr>
<tr>
<td>10</td>
<td>59</td>
</tr>
</tbody>
</table>

Failure happens
Example: Logistic Regression

Goal: find best line separating two sets of points
Example: Logistic Regression

```scala
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
    val gradient = data.map(p =>
        (1 / (1 + exp(-p.y*w dot p.x))) - 1) * p.y * p.x
    .reduce(_ + _)
    w -= gradient
}

println("Final w: " + w)
```

w automatically shipped to cluster
LR/K-Means Performance

K-Means Clustering

- Hadoop MR: 155
- Spark: 4.1

Logistic Regression

- Hadoop MR: 110
- Spark: 0.96

Time per Iteration (s)

10B points
Agenda

1. MapReduce Review
2. Introduction to Spark and RDDs
3. Generality of RDDs (e.g. streaming, ML)
4. DataFrames
5. Internals (time permitting)
Generality of RDDs

Spark

Streaming
real-time

Spark SQL

MLLib
machine
learning

GraphX

graph

Spark
Generality of RDDs

- DStream’s: Streams of RDD’s
- SchemaRDD’s
- RDD-Based Matrices
- RDD-Based Graphs

Spark Streaming real-time
Spark SQL
MLLib machine learning
GraphX graph

RDDs, Transformations, and Actions

Spark
Many important apps must process large data streams at second-scale latencies

- Site statistics, intrusion detection, online ML

To build and scale these apps users want:

- **Integration**: with offline analytical stack
- **Fault-tolerance**: both for crashes and stragglers
- **Efficiency**: low cost beyond base processing
Discretized Stream Processing

$t = 1$: input

$t = 2$: input

batch operation

immutable dataset (stored reliably)

immutable dataset (output or state); stored in memory as RDD
Programming Interface

Simple functional API

views = readStream("http:...", "ls")
one = views.map(ev => (ev.url, 1))
counts = ones.runningReduce(_ + _)

Interoperates with RDDs

// Join stream with static RDD
counts.join(historicCounts).map(...)

// Ad-hoc queries on stream state
counts.slice("21:00","21:05").topK(10)
Inherited “for free” from Spark

RDD data model and API

Data partitioning and shuffles

Task scheduling

Monitoring/instrumentation

Scheduling and resource allocation
Powerful Stack – Agile Development

non-test, non-example source lines
Powerful Stack – Agile Development

- Hadoop MapReduce
- Storm (Streaming)
- Impala (SQL)
- Giraph (Graph)
- Spark

non-test, non-example source lines
Powerful Stack – Agile Development

non-test, non-example source lines
Powerful Stack – Agile Development

non-test, non-example source lines
Powerful Stack – Agile Development

- Hadoop MapReduce
- Storm (Streaming)
- Impala (SQL)
- Giraph (Graph)
- Spark

non-test, non-example source lines
Benefits for Users

High performance data sharing
• Data sharing is the bottleneck in many environments
• RDD’s provide in-place sharing through memory

Applications can compose models
• Run a SQL query and then PageRank the results
• ETL your data and then run graph/ML on it

Benefit from investment in shared functionality
• E.g. re-usable components (shell) and performance optimizations
Agenda

1. MapReduce Review
2. Introduction to Spark and RDDs
3. Generality of RDDs (e.g. streaming, ML)
4. DataFrames
5. Internals (time permitting)
From MapReduce to Spark

```scala
val f = sc.textFile(inputPath)
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(_ + _).saveAsText(outputPath)
```
Beyond Hadoop Users

Spark early adopters

Users
- Understands MapReduce & functional APIs

Data Engineers
- Data Scientists
- Statisticians
- R users
- PyData …
pdata.map(lambda x: (x.dept, [x.age, 1])) \
  .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
  .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
  .collect()

data.groupBy("dept").avg("age")
DataFrames in Spark

Distributed collection of data grouped into named columns (i.e. RDD with schema)

DSL designed for common tasks

• Metadata
• Sampling
• Project, filter, aggregation, join, …
• UDFs

Available in Python, Scala, Java, and R (via SparkR)
Not Just Less Code: Faster Implementations

- **Dataframe SQL**
- **Dataframe Python**
- **Dataframe Scala**
- **RDD Python**
- **RDD Scala**

Time to Aggregate 10 million int pairs (secs)
DataFrame Internals

Represented internally as a “logical plan”

Execution is lazy, allowing it to be optimized by a query optimizer
Plan Optimization & Execution

DataFrames and SQL share the same optimization/execution pipeline

Maximize code reuse & share optimization efforts
\[ \text{joined} = \text{users.join(events, users.id == events.uid)} \]
\[ \text{filtered} = \text{joined.filter(events.date >= "2015-01-01")} \]

This join is expensive ➔

**Logical plan**
- **Filter**
- **Join**
  - **Scan (users)**
  - **Scan (events)**

**Physical plan**
- **Join**
  - **Scan (users)**
  - **Filter**
    - **Scan (events)**
Data Sources supported by DataFrames

**built-in**
- Parquet
- JDBC
- { JSON }
- PostgreSQL
- MySQL
- Hive
- S3
- H2

**external**
- AVRO
- CSV
- dBase
- HBase
- elasticsearch.
- cassandra
- Amazon Redshift
- and more …
More Than Naïve Scans

Data Sources API can automatically prune columns and push filters to the source

- Parquet: skip irrelevant columns and blocks of data; turn string comparison into integer comparisons for dictionary encoded data
- JDBC: Rewrite queries to push predicates down
```python
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date > "2015-01-01")
```
Our Experience So Far

SQL is wildly popular and important
- 100% of Databricks customers use some SQL

Schema is very useful
- Most data pipelines, even the ones that start with unstructured data, end up having some implicit structure
- Key-value too limited
- That said, semi-/un-structured support is paramount

Separation of logical vs physical plan
- Important for performance optimizations (e.g. join selection)
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol="words", outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.01)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

df = sqlCtx.load("/path/to/data")
model = pipeline.fit(df)
R Interface (SparkR)

Spark 1.4 (June)

Exposes DataFrames, and ML library in R

df = jsonFile("tweets.json")

summarize(
  group_by(
    df[df$user == "matei",,
    "date"),
  sum("retweets"))
Data Science at Scale

Higher level interfaces in Scala, Java, Python, R

Drastically easier to program Big Data
  • With APIs similar to single-node tools
Goal: unified engine across data sources, workloads and environments
Agenda

1. MapReduce Review
2. Introduction to Spark and RDDs
3. Generality of RDDs (e.g. streaming, ML)
4. DataFrames
5. Internals (time permitting)
Spark Application

Your program (JVM / Python)

```
sc = new SparkContext
f = sc.textFile("...")

f.filter(…)

.count()
...
```

Spark driver (app master)

- RDD graph
- Scheduler
- Block tracker
- Shuffle tracker

Cluster manager

- Task threads
- Block manager

Spark executor (multiple of them)

HDFS, HBase, ...

A single application often contains multiple actions
RDD is an interface

1. Set of partitions (“splits” in Hadoop)
2. List of dependencies on parent RDDs
3. Function to compute a partition (as an Iterator) given its parent(s)
4. (Optional) partitioner (hash, range)
5. (Optional) preferred location(s) for each partition
Example: HadoopRDD

partitions = one per HDFS block

dependencies = none

compute\((part)\) = read corresponding block

preferredLocations\((part)\) = HDFS block location

partitioner = none
Example: Filtered RDD

**partitions** = same as parent RDD

**dependencies** = “one-to-one” on parent

**compute**(part) = compute parent and filter it

**preferredLocations**(part) = none (ask parent)

**partitioner** = none
RDD Graph (DAG of tasks)

Dataset-level view:

file:

**HadoopRDD**
path = hdfs://...

errors:

**FilteredRDD**
func = \_.contains\((\ldots)\)
shouldCache = true

Partition-level view:

Task1  Task2  ...

errors:
Example: JoinedRDD

`partitions` = one per reduce task

`dependencies` = “shuffle” on each parent

`compute(partition)` = read and join shuffled data

`preferredLocations(part)` = none

`partitioner` = HashPartitioner(numTasks)

Spark will now know this data is hashed!
Dependency Types

“Narrow” (pipeline-able)
- map, filter
- union
- join with inputs co-partitioned

“Wide” (shuffle)
- groupByKey on non-partitioned data
- join with inputs not co-partitioned
Execution Process

RDD Objects

- rdd1.join(rdd2)
- .groupBy(...) .filter(...)
- build operator DAG

DAG Scheduler

- split graph into stages of tasks
- submit each stage as ready

Task Scheduler

- launch tasks via cluster manager
- retry failed or straggling tasks

Worker

- execute tasks
- store and serve blocks
DAG Scheduler

Input: RDD and partitions to compute

Output: output from actions on those partitions

Roles:
- Build stages of tasks
- Submit them to lower level scheduler (e.g. YARN, Mesos, Standalone) as ready
- Lower level scheduler will schedule data based on locality
- Resubmit failed stages if outputs are lost
Job Scheduler

Captures RDD dependency graph
Pipelines functions into “stages”
Cache-aware for data reuse & locality
Partitioning-aware to avoid shuffles

Stage 1
A: map
B: groupBy

Stage 2
C: map
D: union
E: join

Stage 3
F: union
G: = cached partition
Goal: unified engine across data sources, workloads and environments
Thank you. Questions?

@rxin