Who am I?

Reynold Xin

PMC member, Apache Spark
Cofounder & Chief Architect, Databricks

PhD on leave (ABD), UC Berkeley AMPLab
Today’s Talk

1. MapReduce review
2. Spark and RDDs
3. Structure and DataFrames
4. Project Tungsten execution engine
MapReduce review
Google Datacenter

How do we program this thing?
Traditional Network Programming

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

Really difficult to do:

- 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 by hand!
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 to discuss it, since the recent trade press has been filled with news of the revolution of so-called "cloud computing." This paradigm of processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a much 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 teach software tool called MapReduce [1]. Berkeley has gone so far as to plan on teaching their freshman how to program using the Mi

As both educators and researchers, we are amazed at the hype that the MapReduce proponents have spread about how it represents a good idea for writing certain types of general-purpose computations, but to the

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
Higher Level Frameworks

SELECT count(*) FROM users

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

A = load 'foo';
B = group A all;
C = foreach B generate COUNT(A);
Spark and RDDs
A Large Community

Most active open source project in big data
IBM calls Apache Spark “most important new open source project in a decade”

June 15, 2015 Written by Business Cloud News

IBM said it will throw its weight behind Apache Spark, an open source community developing a processing engine for large-scale datasets, putting thousands of internal developers to work on Spark-related projects and contributing its machine learning technology to the code ecosystem.

Spark, an Apache open source project born in 2009, is essentially an engine that can process vast amounts of data very quickly. It runs in Hadoop clusters through YARN or as a standalone deployment and can process data in HDFS, HBase, Cassandra, Hive, and any Hadoop InputFormat; it currently supports Scala, Java and Python.

IBM is throwing its weight behind Apache Spark in a bid to bolster its IoT strategy

It is designed to perform general data...
“Spark is the Taylor Swift of big data software.”

- Derrick Harris, Fortune
## V. Top Paying Tech

<table>
<thead>
<tr>
<th>Top Paying Tech in US</th>
<th>Top Paying Tech Worldwide</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spark</td>
<td>$125,000</td>
</tr>
<tr>
<td>Scala</td>
<td>$125,000</td>
</tr>
<tr>
<td>Cassandra</td>
<td>$115,000</td>
</tr>
<tr>
<td>F#</td>
<td>$115,000</td>
</tr>
<tr>
<td>Hadoop</td>
<td>$115,000</td>
</tr>
<tr>
<td>Cloud (AWS, GAE, Azure, etc.)</td>
<td>$105,000</td>
</tr>
<tr>
<td>Redis</td>
<td>$105,000</td>
</tr>
</tbody>
</table>
Apache Spark Stack

Spark SQL + DataFrames

Streaming

MLlib
*Machine Learning*

GraphX
*Graph Computation*

Spark Core API

R
SQL
Python
Scala
Java
A slide from 2013 …

Spark

Fast and expressive cluster computing system interoperable with Apache Hadoop

Improves efficiency through:
- In-memory computing primitives
- General computation graphs

Improves usability through:
- Rich APIs in Scala, Java, Python
- Interactive shell

→ Up to 100× faster (2-10× on disk)

→ Often 5× less code
On-Disk Sort Record:
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
Programmability

WordCount in 3 lines of Spark

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

WordCount in 50+ lines of Java MR
**RDD**

**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 $f(RDD) \Rightarrow RDD$
- Lazy (not computed immediately)
- E.g. “map”

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

```python
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"))
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://...")
events = lines.filter(lambda s: s.startswith("ERROR"))
messages = events.map(lambda s: s.split("\t")[-1])
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()
```
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()
```
Expressive API

map  reduce
## Expressive API

<table>
<thead>
<tr>
<th>map</th>
<th>reduce</th>
<th>sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>count</td>
<td>take</td>
</tr>
<tr>
<td>groupBy</td>
<td>fold</td>
<td>first</td>
</tr>
<tr>
<td>sort</td>
<td>reduceByKey</td>
<td>partitionBy</td>
</tr>
<tr>
<td>union</td>
<td>groupByKey</td>
<td>mapWith</td>
</tr>
<tr>
<td>join</td>
<td>cogroup</td>
<td>pipe</td>
</tr>
<tr>
<td>leftOuterJoin</td>
<td>cross</td>
<td>save</td>
</tr>
<tr>
<td>rightOuterJoin</td>
<td>zip</td>
<td>...</td>
</tr>
</tbody>
</table>
What Spark RDDs are vs MapReduce?

Functional APIs that generalize MapReduce

- Arbitrary DAG of tasks rather than simply map -> reduce

A better engineering implementation

- Faster scheduling (ms vs secs to launch tasks)
- Eliminate shuffle whenever possible
Structure and DataFrames
Original Spark Vision

1) Unified engine for big data processing
   • Combines batch, interactive, iterative, streaming

2) Concise, language-integrated API
   • Functional programming in Scala/Java/Python
Motivation: Unification

General batch processing

Specialized systems for new workloads

MapReduce

Pregel

Dremel

Giraph

Impala

Drill

Storm

Presto

S4

Hard to compose in pipelines
Motivation: Unification

MapReduce

General batch processing

Pregel
Dremel
Impala
Storm

Specialized systems for new workloads

Giraph
Drill
Presto
S4 ...

Unified engine

spark
Motivation: Concise API

Much of data analysis is exploratory / interactive

Spark solution: Resilient Distributed Datasets (RDDs)

- “Distributed collection” abstraction with simple functional API

```scala
lines = spark.textFile("hdfs://...")
points = lines.map(line => parsePoint(line))
points.filter(p => p.x > 100).count()
```
How Did the Vision Hold Up?

Mostly well!

Users really appreciate unification

Functional API causes some challenges, which we are tackling
Libraries Built on Spark

- SQL
- Streaming
- MLlib
- GraphX

Spark Core (RDDs)

Largest integrated standard library for big data
Which Libraries Do People Use?

- Spark SQL: 69%
- DataFrames: 62%
- Spark Streaming: 58%
- MLlib + GraphX: 58%

75% of users use more than one component.
Which Languages Do People Use?

2014 Languages Used
- Scala: 84%
- Java: 38%
- Python: 38%

2015 Languages Used
- Scala: 71%
- Java: 31%
- Python: 58%
- R: 18%
Main Challenge: Functional API

Looks high-level, but hides many semantics of computation from engine

- Functions are arbitrary blocks of code
- Data stored is arbitrary Java/Python objects

Users can mix APIs in suboptimal ways
Scaling Spark users

Early adopters

Hadoop

Users

Understands MapReduce & functional APIs

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")
Structure
[ˈstrək(t)SHər]

verb
1. construct or arrange according to a plan; give a pattern or organization to.
Why Structure?

By definition, structure will *limit* what can be expressed.

In practice, we can accommodate the vast majority of computations.

Limiting the space of what can be expressed enables optimizations.
DataFrames in Spark

Distributed data frame abstraction for Java, Python, R, Scala

Similar APIs as single-node tools (Pandas, dplyr), i.e. easy to learn

```r
> head(filter(df, df$waiting < 50))  # an example in R
## eruptions waiting
##1 1.750 47
##2 1.750 47
##3 1.867 48
```
DataFrames hold rows with a known schema and offer relational operations on them through a DSL

```scala
val c = new HiveContext()
val users = c.sql("select * from users")
val massUsers = users(users("state") === "MA")
massUsers.count()
massUsers.groupBy("name").avg("age")
```
Execution Process

- SQL
- Data Frames
- Datasets
- Catalog
- Logical Plan
- Physical Plan
- Code Generator
- RDDs
- Data Source API

Data Source API:
- HDFS
- Cassandra
- HBase
- Elasticsearch
- More...
What Structured APIs Enable

1. Compact binary representation
   - Columnar, compressed format for caching; rows for processing

2. Optimization across operators (join ordering, pushdown, etc)

3. Runtime code generation
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date >= "2015-01-01")

this join is expensive ➔
Project Tungsten

Can we speed up Spark by 10X?
Hardware Trends

Storage

Network

CPU
<table>
<thead>
<tr>
<th>Hardware Trends</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
</tr>
<tr>
<td><strong>Storage</strong></td>
</tr>
<tr>
<td>~50+MB/s (HDD)</td>
</tr>
<tr>
<td><strong>Network</strong></td>
</tr>
<tr>
<td>1Gbps</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
</tr>
<tr>
<td>~3GHz</td>
</tr>
</tbody>
</table>
# Hardware Trends

<table>
<thead>
<tr>
<th></th>
<th>2010</th>
<th>2016</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Storage</strong></td>
<td>50+MB/s (HDD)</td>
<td>500+MB/s (SSD)</td>
</tr>
<tr>
<td><strong>Network</strong></td>
<td>1Gbps</td>
<td>10Gbps</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>~3GHz</td>
<td>~3GHz</td>
</tr>
</tbody>
</table>
## Hardware Trends

<table>
<thead>
<tr>
<th>Domain</th>
<th>2010</th>
<th>2016</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>50+MB/s (HDD)</td>
<td>500+MB/s (SSD)</td>
<td>10x</td>
</tr>
<tr>
<td>Network</td>
<td>1Gbps</td>
<td>10Gbps</td>
<td>10x</td>
</tr>
<tr>
<td>CPU</td>
<td>~3GHz</td>
<td>~3GHz</td>
<td>🙁</td>
</tr>
</tbody>
</table>
Demo

http://bit.ly/1X8LKmH
Going back to the fundamentals

Difficult to get order of magnitude performance speed ups with profiling techniques

• For 10x improvement, would need to find top hotspots that add up to 90% and make them instantaneous
• For 100x, 99%

Instead, look bottom up, how fast should it run?
select count(*) from store_sales where ss_item_sk = 1000
Volcano Iterator Model

Standard for 30 years: almost all databases do it

Each operator is an “iterator” that consumes records from its input operator

```scala
class Filter {
  def next(): Boolean = {
    var found = false
    while (!found && child.next()) {
      found = predicate(child.fetch())
    }
    return found
  }

  def fetch(): InternalRow = {
    child.fetch()
  }
}
```
What if we hire a college freshman to implement this query in Java in 10 mins?

```java
select count(*) from store_sales
where ss_item_sk = 1000

var count = 0
for (ss_item_sk in store_sales) {
    if (ss_item_sk == 1000) {
        count += 1
    }
}
```
Volcano model
30+ years of database research

vs

college freshman
hand-written code in 10 mins
Volcano

13.95 million rows/sec

College freshman

125 million rows/sec

Note: End-to-end, single thread, single column, and data originated in Parquet on disk
How does a student beat 30 years of research?

Volcano
1. Many virtual function calls
2. Data in memory (or cache)
3. No loop unrolling, SIMD, pipelining

hand-written code
1. No virtual function calls
2. Data in CPU registers
3. Compiler loop unrolling, SIMD, pipelining

Take advantage of all the information that is known after query compilation
long count = 0;
for (ss_item_sk in store_sales) {
    if (ss_item_sk == 1000) {
        count += 1;
    }
}

Functionality of a general purpose execution engine; performance as if hand built system just to run your query
## Performance of Core Primitives

<table>
<thead>
<tr>
<th>primitive</th>
<th>cost per row (single thread)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Spark 1.6</td>
</tr>
<tr>
<td>filter</td>
<td>15 ns</td>
</tr>
<tr>
<td>sum w/o group</td>
<td>14 ns</td>
</tr>
<tr>
<td>sum w/ group</td>
<td>79 ns</td>
</tr>
<tr>
<td>hash join</td>
<td>115 ns</td>
</tr>
<tr>
<td>sort (8 bit entropy)</td>
<td>620 ns</td>
</tr>
<tr>
<td>sort (64 bit entropy)</td>
<td>620 ns</td>
</tr>
<tr>
<td>sort-merge join</td>
<td>750 ns</td>
</tr>
</tbody>
</table>

Intel Haswell i7 4960HQ 2.6GHz, HotSpot 1.8.0_60-b27, Mac OS X 10.11
Today’s Talk

1. MapReduce review
2. Spark and RDDs
3. Structure and DataFrames
4. Project Tungsten execution engine

Takeaway: evolution (MR->RDD->DF) reflects making data processing easier, faster, and smarter.
Want to Learn Apache Spark?

Databricks Community Edition offers free hands-on tutorials

databricks.com/ce
COMING TO SF THIS JUNE.
#sparksummit

SPARK SUMMIT 2016
JUNE 6-8, 2016 • SAN FRANCISCO

REGISTER TODAY

20% Discount code: reynold16
Thank you.

@rxin