What is Spark?

http://spark.apache.org

Not a modified version of Hadoop

Separate, fast, MapReduce-like engine
  » In-memory data storage for very fast iterative queries
  » General execution graphs and powerful optimizations
  » Up to 40x faster than Hadoop

Compatible with Hadoop’s storage APIs
  » Can read/write to any Hadoop-supported system, including HDFS, HBase, SequenceFiles, etc.
What is **Spark SQL**?
(Formally called Shark)

Port of Apache **Hive** to run on **Spark**

Compatible with existing Hive data, metastores, and queries (HiveQL, UDFs, etc)

Similar speedups of up to 40x
Project History

Spark project started in 2009 at UC Berkeley AMP lab, open sourced 2010

Became Apache Top-Level Project in Feb 2014

Shark/Spark SQL started summer 2011

Built by 250+ developers and people from 50 companies

Scale to 1000+ nodes in production

In use at Berkeley, Princeton, Klout, Foursquare, Conviva, Quantifind, Yahoo! Research, …

http://en.wikipedia.org/wiki/Apache_Spark
Why a New Programming Model?

MapReduce greatly simplified big data analysis.

But as soon as it got popular, users wanted more:

» More **complex**, multi-stage applications (e.g. iterative graph algorithms and machine learning)
» More **interactive** ad-hoc queries
Why a New Programming Model?

MapReduce greatly simplified big data analysis

But as soon as it got popular, users wanted more:
  » More complex, multi-stage applications (e.g. iterative graph algorithms and machine learning)
  » More interactive ad-hoc queries

Require faster data sharing across parallel jobs
Is MapReduce dead? Not really.


http://www.reddit.com/r/compsci/comments/296aqr/on_the_death_of_mapreduce_at_google/

As an employee, I was surprised by this headline, considering I just ran some mapreduces this past week. After digging further, this headline and article is rather inaccurate.

Cloud DataFlow is the external name for what is internally called Flume. Flume is a layer that runs on top of MapReduce that abstracts away the complexity into something that is much easier.
Data Sharing in MapReduce

- Input
  - HDFS read
  - Iter. 1
  - HDFS write
  - Iter. 2
  - HDFS read
  - HDFS write
  - ...

- Input
  - HDFS read
  - Query 1
  - Result 1
  - Query 2
  - Result 2
  - Query 3
  - Result 3
  - ...

- Each iteration involves reading from HDFS, processing (query), and writing back to HDFS.
Data Sharing in MapReduce

Input

HDFS read

iter. 1

HDFS write

HDFS read

iter. 2

HDFS write

HDFS read

query 1

result 1

query 2

result 2

query 3

result 3

Slow due to replication, serialization, and disk IO
Data Sharing in **Spark**

[Diagram showing data sharing process in Spark, including input data, distributed memory, iterations, and one-time processing]

- Input
- Distributed memory
- Iter. 1
- Iter. 2
- Query 1
- Query 2
- Query 3
- One-time processing
Data Sharing in **Spark**

Input

iter. 1

iter. 2

... 

Input

Distributed memory

one-time processing

query 1

query 2

query 3

... 

**10-100×** faster than network and disk
Spark Programming Model

Key idea: resilient distributed datasets (RDDs)
  » Distributed collections of objects that can be cached in memory across cluster nodes
  » Manipulated through various parallel operators
  » Automatically rebuilt on failure

Interface
  » Clean language-integrated API in Scala
  » Can be used interactively from Scala, Python console
  » Supported languages: Java, Scala, Python, R
Functional programming in D3: [http://sleptons.blogspot.com/2015/01/functional-programming-d3js-good-example.html](http://sleptons.blogspot.com/2015/01/functional-programming-d3js-good-example.html)

Example: Log Mining

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

Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

http://ananthakumaran.in/2010/03/29/scala-underscore-magic.html
http://www.slideshare.net/normation/scala-dreaded
Example: Log Mining

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

Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

http://ananthakumaran.in/2010/03/29/scala-underscore-magic.html
http://www.slideshare.net/normation/scala-dreaded
Example: Log Mining

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

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()
```

Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

http://ananthakumaran.in/2010/03/29/scala-underscore-magic.html
http://www.slideshare.net/normation/scala-dreaded
Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split(\t')(2))
cachedMsgs = messages.cache()
```

**Result:** scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

http://ananthakumaran.in/2010/03/29/scala-underscore-magic.html
http://www.slideshare.net/normation/scala-dreaded
Example: Log Mining

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

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split("\t")(2))
cachedMsgs = messages.cache()
```

Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

http://ananthakumaran.in/2010/03/29/scala-underscore-magic.html
http://www.slideshare.net/normation/scala-dreaded
Example: Log Mining

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

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
```

**Result:** scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

http://ananthakumaran.in/2010/03/29/scala-underscore-magic.html
http://www.slideshare.net/normation/scala-dreaded
Example: Log Mining

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

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split(\t\')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
```

**Result:** scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

http://ananthakumaran.in/2010/03/29/scala-underscore-magic.html
http://www.slideshare.net/normation/scala-dreaded
Example: Log Mining

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

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split("\t")(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
```

**Result:** scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

http://ananthakumaran.in/2010/03/29/scala-underscore-magic.html
http://www.slideshare.net/normation/scala-dreaded
Example: Log Mining

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

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split("\t")(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
...
```

**Result:** scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

http://ananthakumaran.in/2010/03/29/scala-underscore-magic.html
http://www.slideshare.net/normation/scala-dreaded
Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split("\t")(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
...
```

**Result:** scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

http://ananthakumaran.in/2010/03/29/scala-underscore-magic.html
http://www.slideshare.net/normation/scala-dreaded
Fault Tolerance

RDDs track the series of transformations used to build them (their *lineage*) to recompute lost data.

E.g:

```scala
messages = textFile(...).filter(_.contains("error"))
  .map(_.split("\t")(2))
```

- **HadoopRDD**
  - path = hdfs://…

- **FilteredRDD**
  - func = _.contains(…)

- **MappedRDD**
  - func = _.split(…)
Example: Logistic Regression

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)
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)
```

Load data in memory once
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)
```

Initial parameter vector
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)
```

Repeated MapReduce steps to do gradient descent
Logistic Regression Performance

- Running Time (s)
  - First iteration: 174 s
  - Further iterations: 6 s

- Number of Iterations
  - 1
  - 5
  - 10
  - 20
  - 30

- Performance:
  - Hadoop: 127 s / iteration
  - Spark: 127 s / iteration

- Notes:
  - First iteration 174 s
  - Further iterations 6 s
### Supported Operators

- `map`
- `filter`
- `groupBy`
- `sort`
- `join`
- `leftOuterJoin`
- `rightOuterJoin`
- `reduce`
- `count`
- `reduceByKey`
- `groupByKey`
- `first`
- `union`
- `cross`
- `sample`
- `cogroup`
- `take`
- `partitionBy`
- `pipe`
- `save`
- ...
Spark SQL: Hive on Spark
Motivation

Hive is great, but Hadoop’s execution engine makes even the smallest queries take minutes.

Scala is good for programmers, but many data users only know SQL.

Can we extend Hive to run on Spark?
Hive Architecture

- Meta store
- Client
  - CLI
  - JDBC
- Driver
  - SQL Parser
  - Query Optimizer
  - Physical Plan
  - Execution
- MapReduce
- HDFS
Spark SQL Architecture

[Engle et al, SIGMOD 2012]
Using Spark SQL

CREATE TABLE mydata_cached AS SELECT …

Run standard HiveQL on it, including UDFs
  » A few esoteric features are not yet supported

Can also call from Scala to mix with Spark
Benchmark Query 1

\[
\text{SELECT * FROM grep WHERE field LIKE '\%XYZ\%';}
\]
**Benchmark Query 2**

```
SELECT sourceIP, AVG(pageRank), SUM(adRevenue) AS earnings
FROM rankings AS R, userVisits AS V ON R.pageURL = V.destURL
WHERE V.visitDate BETWEEN '1999-01-01' AND '2000-01-01'
GROUP BY V.sourceIP
ORDER BY earnings DESC
LIMIT 1;
```
Behavior with Not Enough RAM

% of working set in memory

Iteration time (s)

Cache disabled: 68.8 ± 11.5
25%: 58.1 ± 11.5
50%: 40.7 ± 11.5
75%: 29.7 ± 11.5
Fully cached: 11.5 ± 11.5
What’s Next?

Recall that Spark’s model was motivated by two emerging uses (interactive and multi-stage apps).

Another emerging use case that needs fast data sharing is **stream processing**

- Track and update state in memory as events arrive
- Large-scale reporting, click analysis, spam filtering, etc.
Streaming Spark

Extends Spark to perform streaming computations

Runs as a series of small (~1 s) batch jobs, keeping state in memory as fault-tolerant RDDs

Intermix seamlessly with batch and ad-hoc queries

tweetStream
  .flatMap(_.toLower.split)
  .map(word => (word, 1))
  .reduceByWindow("5s", _ + _)

[Zaharia et al, HotCloud 2012]
map() vs flatMap()

The best explanation:

https://www.linkedin.com/pulse/difference-between-map-flatmap-transformations-spark-pyspark-pandey

flatMap = map + flatten
Streaming Spark

Extends Spark to perform streaming computations

Runs as a series of small (~1 s) batch jobs, keeping state in memory as fault-tolerant RDDs

Intermix seamlessly with batch and ad-hoc queries

Result: can process 42 million records/second (4 GB/s) on 100 nodes at sub-second latency

[Zaharia et al, HotCloud 2012]
Spark Streaming

Create and operate on RDDs from live data streams at set intervals

Data is divided into batches for processing

Streams may be combined as a part of processing or analyzed with higher level transforms
GraphX

Parallel graph processing

Extends RDD -> Resilient Distributed Property Graph
  » Directed multigraph with properties attached to each vertex and edge

Limited algorithms
  » PageRank
  » Connected Components
  » Triangle Counts
MLlib (part of Spark 2.x)

- Basic statistics
  - summary statistics
  - correlations
  - stratified sampling
  - hypothesis testing
  - streaming significance testing
  - random data generation
- Classification and regression
  - linear models (SVMs, logistic regression, linear regression)
  - naive Bayes
  - decision trees
  - ensembles of trees (Random Forests and Gradient-Boosted Trees)
  - isotonic regression
- Collaborative filtering
  - alternating least squares (ALS)
- Clustering
  - k-means
  - Gaussian mixture
  - power iteration clustering (PIC)
  - latent Dirichlet allocation (LDA)
  - bisecting k-means
  - streaming k-means
- Dimensionality reduction
  - singular value decomposition (SVD)
  - principal component analysis (PCA)
- Feature extraction and transformation
- Frequent pattern mining
  - FP-growth
  - association rules
  - PrefixSpan
- Evaluation metrics
- PMML model export
- Optimization (developer)
  - stochastic gradient descent
  - limited-memory BFGS (L-BFGS)

https://spark.apache.org/docs/latest/mllib-guide.html