New Developments in Spark
And Rethinking APIs for Big Data

Matei Zaharia and many others
What is Spark?

Unified computing engine for big data apps
  > Batch, streaming and interactive

Collection of high-level APIs
  > One of first widely used systems with a functional API
  > Libraries for SQL, ML, graph, ...

SQL  |  Streaming  |  MLlib  |  GraphX
---   |            |        |      |
Spark
## Project Growth

<table>
<thead>
<tr>
<th></th>
<th>June 2013</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code</td>
<td>70,000</td>
</tr>
<tr>
<td>Total contributors</td>
<td>80</td>
</tr>
<tr>
<td>Monthly contributors</td>
<td>20</td>
</tr>
<tr>
<td>Largest cluster</td>
<td>400 nodes</td>
</tr>
</tbody>
</table>
# Project Growth

<table>
<thead>
<tr>
<th></th>
<th>June 2013</th>
<th>January 2016</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code</td>
<td>70,000</td>
<td>450,000</td>
</tr>
<tr>
<td>Total contributors</td>
<td>80</td>
<td>1000</td>
</tr>
<tr>
<td>Monthly contributors</td>
<td>20</td>
<td>140</td>
</tr>
<tr>
<td>Largest cluster</td>
<td>400 nodes</td>
<td>8000 nodes</td>
</tr>
</tbody>
</table>

Most active open source project in big data
This Talk

Original Spark vision

How did the vision hold up?

New APIs: DataFrames + Spark SQL

New capabilities under these APIs

Ongoing research
Original Spark Vision

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

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

General batch processing

Specialized systems for new workloads

Hard to compose into pipelines
Motivation: Unification

General batch processing → Specialized systems for new workloads → Unified engine
Motivation: Concise API

Much of data analysis is exploratory / interactive

Answer: Resilient Distributed Datasets (RDDs)
   > Distributed collections with simple functional API

```
lines = spark.textFile("hdfs://...")
points = lines.map(line => parsePoint(line))
points.filter(p => p.x > 100).count()
```
This Talk

Original Spark vision

How did the vision hold up?

New APIs: DataFrames + Spark SQL

New capabilities under these APIs

Ongoing research
How Did the Vision Hold Up?

Mostly well

Users really appreciate unification

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

Spark SQL
relational

Spark Streaming
real-time

MLlib
machine learning

GraphX
graph

Spark Core

Largest integrated library for big data
Which Libraries do People Use?

80% of users use more than one component
60% use three or more

- Spark SQL: 69%
- Streaming: 58%
- MLlib: 54%
- GraphX: 18%
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 the engine

> Functions passed in are arbitrary blocks of code
> Data stored is arbitrary Java/Python objects

Users can mix APIs in suboptimal ways
<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 People Do

pairs = data.map(word => (word, 1))

groups = pairs.groupByKey()

groups.map((k, vs) => (k, vs.sum))

Materializes all groups as lists of integers

(“the”, [1, 1, 1, 1, 1, 1])
(“quick”, [1, 1])
(“fox”, [1, 1])

Then sums each list

(“the”, 6)
(“quick”, 2)
(“fox”, 2)

Better code: pairs.reduceByKey(_ + _)
Challenge: Data Representation

Object graphs much larger than underlying data

class User(name: String, friends: Array[Int])

```
User 0x... 0x...
  ↓
String 0 5 0x...
  ↓
  ↓
int[] 3 1 2
  ↓
  ↓
char[] 5 Bobby
```
This Talk

Original Spark vision

How did the vision hold up?

New APIs: DataFrames + Spark SQL

New capabilities under these APIs

Ongoing research
DataFrames and Spark SQL

Efficient library for working with structured data

> Two interfaces: SQL for data analysts + external apps, DataFrames for programmers

Optimized computation and storage
Spark SQL Architecture

- SQL
- Data Frames
- Logical Plan
- Optimizer
- Physical Plan
- Code Generator
- RDDs
- Catalog
- Data Source API
- HDFS
- Cassandra
- HBase
- elasticsearch
- PostgreSQL
- Hive
- ...
DataFrame API

DataFrames hold **rows** with a known schema and offer **relational operations** on them through a DSL

```python
users = sql("select * from users")
ma_users = users[users.state == "MA"]
ma_users.count()   # Expression AST
ma_users.groupBy("name").avg("age")
ma_users.map(lambda u: u.name.toUpper())
```
API Details

Based on data frame concept in R and Python
  > Spark is first system to make this API declarative
Integrated with the rest of Spark
  > MLlib takes DataFrames as input/output
  > Easily convert RDDs ↔ DataFrames

Google trends for “data frame”
What DataFrames Enable

1. Compact binary representation
   • Columnar format outside Java heap

2. Optimization across operators (join reordering, predicate pushdown, etc)

3. Runtime code generation
Performance

<table>
<thead>
<tr>
<th>Method</th>
<th>Time for aggregation benchmark (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDD Python</td>
<td>10</td>
</tr>
<tr>
<td>RDD Scala</td>
<td>4</td>
</tr>
</tbody>
</table>
Performance

- DataFrame SQL
- DataFrame R
- DataFrame Python
- DataFrame Scala
- RDD Python
- RDD Scala

Time for aggregation benchmark (s)
DataFrames vs SQL

Easier to compose into large programs: organize code into functions, classes, etc

“[DataFrames are] concise and declarative like SQL, but I can name intermediate values”

Spark 1.6 adds static typing over DataFrames (Datasets: tinyurl.com/spark-datasets)
This Talk

Original Spark vision

How did the vision hold up?

New APIs: DataFrames + Spark SQL

New capabilities under these APIs

Ongoing research
New Capabilities under Spark SQL

Uniform and efficient access to data sources

Rich optimization across libraries
Data Sources

Having a uniform API for structured data lets apps migrate across data sources

> Hive, MySQL, Cassandra, JSON, ...

API semantics allow query pushdown into sources (not possible with old RDD API)

users[users.age > 20]

select id from users
Examples

**JSON:**
```
{  
  "text": "hi",
  "user": {  
    "name": "bob",
    "id": 15 
  }
}
tweets.json
```

**JDBC:**
```
select age from users where lang = "en"
```

**Together:**
```
select t.text, u.age  
from tweets t, users u  
where t.user.id = u.id  
and u.lang = "en"
```
Library Composition

One of our goals was to unify processing types

Problem: optimizing across libraries
  > Big data is expensive to copy & scan
  > Libraries are written in isolation

Spark SQL gives more semantics to do this
Example: ML Pipelines

New API in MLlib that lets users express and optimize end-to-end workflows

- Feature preparation, training, evaluation
- Similar to scikit-learn, but declarative

```
tokenizer = Tokenizer()
tf = HashingTF(features=1000)
lr = LogisticRegression(r=0.1)

p = Pipeline(tokenizer, tf, lr)
p.fit(df)
```

```
CrossValidator.fit(p, df, args)
```

Filters pushed into data source

Fused into one pass over data

Repeated queries
This Talk

Original Spark vision

How did the vision hold up?

New APIs: DataFrames + Spark SQL

New capabilities under these APIs

Ongoing research
The Problem

Hardware has changed a lot since big data systems were first designed.

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

Hardware has changed a lot since big data systems were first designed.

<table>
<thead>
<tr>
<th>Hardware Type</th>
<th>2010</th>
<th>2015</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>50+MB/s (HDD)</td>
<td>500+MB/s (SSD)</td>
</tr>
<tr>
<td>Network</td>
<td>1Gbps</td>
<td>10Gbps</td>
</tr>
<tr>
<td>CPU</td>
<td>~3GHz</td>
<td>~3GHz</td>
</tr>
</tbody>
</table>
## The Problem

Hardware has changed a lot since big data systems were first designed.

<table>
<thead>
<tr>
<th></th>
<th>2010</th>
<th>2015</th>
<th>10x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage (HDD)</td>
<td>50+MB/s</td>
<td>500+MB/s</td>
<td></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>

New bottleneck in Spark, Hadoop, etc.
To Make Matters Worse

In response to the slowdown of Moore’s Law, hardware is becoming more diverse

Have to optimize separately for each platform!
Observation

Many common algorithms can be written with “embarrassingly” data-parallel operations

> See how many run on MapReduce / Spark

Focus on optimizing these as opposed to general programs (e.g. C++)
The Goal

- SQL
- machine learning
- graph algorithms
- intermediate language
- CPUs
- GPUs
- ...

transformations
Nested Vector Language (NVL)

Functional-like parallel language
  > Captures SQL, machine learning, and graphs, but very easy to analyze

Closed under composition (nested calls) and common transformations (e.g. loop fusion)
  > Unlike relational algebra, OpenCL, NESL
Example Transformations

```python
def query(products: vec[{dept:int, price:int}]):
    sum = 0
    for p in products:
        if p.dept == 20: sum += p.price
```

**row-to-column**

```python
def query(dept: vec[int], price: vec[int]):
    sum = 0
    for i in 0..len(users):
        if dept[i] == 20: sum += price[i]
```

**vectorization**

```python
for i in 0..len(products) by 4:
    sum += price[i..i+4] * (dept[i..i+4] == [20,20,20,20])
```
Results: TPC-H Q6

Python: 0.53 seconds
Java: 0.14 seconds
C: 0.08 seconds
HyPer Database: 0.11 seconds
NVL: 0.03 seconds
Effect of Transformations

Transformations usable on any NVL program
Library Composition API

Disjoint libraries can take & return “NVL objects” to build up a combined program

Example: optimize across Spark and NumPy

```python
data = sql("select features from users where age>20")
scores = data.map(lambda vec: scoreMatrix * vec)
mean = scores.mean()
```
Conclusion

Large data volumes + changing hardware pose a formidable challenge for next-generation apps

Spark shows a unified API for data apps is useful

NVL targets a new range of optimizations and environments