What’s Changing in Big Data?

Matei Zaharia
Stanford University
Background

The first big data systems were designed 10 years ago

What’s changed since then?
My Perspective

Open source processing engine and set of libraries

Cloud service based on Spark
Three Key Changes

1. **Users:** engineers → analysts

2. **Hardware:** I/O bottleneck → compute

3. **Delivery:** the public cloud
Changing Users

Initial users: **software engineers**
- Use Java, C#, C++ to create large projects
- Build apps out of low-level operators

New users: **data scientists & analysts**
- SQL-like and scripting languages
- BI tools, e.g. Tableau
Example: Languages Used for Spark

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

2015 Languages Used
- Scala: 71%
- Java: 31%
- Python: 58%
- R: 18%
Functional API targeting Java / Scala developers

- Resilient Distributed Datasets (RDDs): collections with functional operators

```scala
lines = spark.textFile("hdfs://...")
points = lines.map(line => parsePoint(line))
points.filter(p => p.x > 100).count()
```
Challenge with Functional API

Looks high-level, but hides many semantics of program

- Functions are arbitrary blocks of Java bytecode
- Data stored is arbitrary Java objects

Users can mix APIs in suboptimal ways
Which Operator Causes the Most Issues?

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operator</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>reduce</td>
<td>sample</td>
</tr>
<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>cogroup</td>
<td>mapWith</td>
</tr>
<tr>
<td>join</td>
<td>cross</td>
<td>pipe</td>
</tr>
<tr>
<td>leftOuterJoin</td>
<td>zip</td>
<td>save</td>
</tr>
<tr>
<td>rightOuterJoin</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Example Problem

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

groups = pairs.groupByKey()

groups.map((k, vs) =&gt; (k, vs.sum))

Materializes all groups as Seq[Int] objects

Then promptly aggregates them
Solution: DataFrames and Spark SQL

Efficient API for **structured data** (known schema)
- Based on the popular “data frame” API in Python and R

Optimized execution similar to RDBMS
Execution Steps

SQL → Logical Plan → Optimizer → Physical Plan → Code Generator → RDDs

Data Frames

RDDs

Libraries:
- Hops
- Cassandra
- Hbase
- Elasticsearch
- PostgreSQL
- Hive
Programming Model

DataFrames hold rows with a known schema and offer relational ops through a DSL

```
users = ctx.sql("select * from hive.users")
ca_users = users[users.state == "CA"]
ca_users.count()  # Expression AST
ca_users.groupBy("name").avg("age")
ca_users.map(lambda row: row.name.upper())
```
What DataFrames Enable

1. Compact binary representation
2. Optimization across operators (e.g. join ordering)
3. Runtime code generation
Other Declarative APIs in Spark

Machine Learning Pipelines
Modular API based on scikit-learn

GraphFrames
Relational + graph operations

Structured Streaming

All built on DataFrames enables \textit{cross-library} optimization
Three Key Changes

1. **Users**: engineers ➔ analysts

2. **Hardware**: I/O bottleneck ➔ compute

3. **Delivery**: the public cloud
Hardware Trends

2010

Storage 50+MB/s (HDD)

Network 1Gbps

CPU ~3GHz
## Hardware Trends

<table>
<thead>
<tr>
<th>Component</th>
<th>2010</th>
<th>2016</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>
## Hardware Trends

<table>
<thead>
<tr>
<th></th>
<th>2010</th>
<th>2016</th>
<th>2016 Speedup</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>
Summary

In 2005-2010, I/O was the name of the game
  • Network locality, compression, in-memory caching

Now, CPU and DRAM are often bottlenecks
  • Many current systems are 2-10x off peak performance
In-Memory Performance Gap

Results from Nested Vector Language (NVL) project at MIT

- **HyPer Database**
- **GraphMat PageRank**
- **TensorFlow Word2Vec**

- Current in-memory systems
- Hand tuned code
Spark Effort: Project Tungsten

Optimize Spark’s CPU and memory usage via manual memory management and code generation

- Spark 1.6: 14M rows/s
- Spark 2.0: 125M rows/s
Three Key Changes

1. **Users**: engineers ➔ analysts

2. **Hardware**: I/O bottleneck ➔ compute

3. **Delivery**: the public cloud
Cloud Requires a Rethink of Systems

- Multi-tenant
- Fully measured
- Elastic
- Continuously updated

Must design an organization, not a piece of software
Conclusion

Big data systems are now widely deployed, but still face big usability challenges.

If you want a large set of apps and libraries, Spark DataFrames, ML Pipelines, etc are open source.