What’s Changing in Big Data?

Matei Zaharia
June 21, 2016
Big data systems became a popular research topic nearly 10 years ago.

- Large-scale, commodity clusters

What has changed since then?
My Perspective

Open source processing engine and set of libraries

Cloud data processing service based on Apache Spark
Three Key Changes

1. **Users**: engineers ➔ analysts

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

3. **Delivery**: strong trend toward cloud
Changing Users
Initial Big Data Users

Programmers find the system easy to use: more than ten thousand distinct MapReduce programs have been implemented internally at Google over the past four years, and an average of one hundred thousand MapReduce jobs are executed on

Software engineers:

• Use Java, C++, etc to create large projects
• Build applications out of low-level operators
Expanding the User Base

Scripting / query languages inspired by SQL, awk, etc

Used by new roles:
- Data scientists (technical domain experts, e.g. ML)
- Analysts (business)
Challenges for Non-Engineers

API familiarity

Performance predictability & debugging
  Can’t hide that it’s large-scale

Access from small data tools
  E.g. Excel, Tableau

Worse with more familiar APIs!
Case Study: Apache Spark

Cluster computing engine that generalizes MapReduce

Collection of APIs and libraries
- APIs in Scala, Java, Python and R
- Streaming, SQL, ML, graph, ...

1000+ deployments, max > 8000 nodes

Spark

SQL + DataFrames
Streaming
MLlib
GraphX
Languages Used for Spark

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

2015 Languages Used
- Scala: 71%
- Java: 31%
- Python: 58%
- R: 18%
Original Spark API

Functional API aimed at Java / Scala developers

Resilient Distributed Datasets (RDDs): distributed collections with functional transformations

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

Looks high-level, but **hides** many semantics of computation

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

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

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

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

groups = pairs.groupByKey()

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

Materializes all groups as Seq[Int] objects

Then promptly aggregates them
Challenge: Data Representation

Java objects often many times larger than underlying fields

class User(name: String, friends: Array[Int])
new User(“Bobby”, Array(1, 2))
Structured APIs: DataFrames + Spark SQL
DataFrames and Spark SQL

Efficient library for structured data (data with a known schema)

- Two interfaces: SQL for analysts + apps, DataFrames for programmers

Optimized computation and storage, similar to RDBMS
Execution Steps

- SQL
- Data Frames
- Logical Plan
- Optimizer
- Physical Plan
- RDDs
- Catalog
- Data Source API

APIs:
- HDFS
- Cassandra
- Apache HBase
- elasticsearch
- Hive
- PostgreSQL
- ...
DataFrame API

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()
Expression AST
massUsers.groupBy("name").avg("age")
massUsers.map(row => row.getString(0).toUpperCase())
```
Why DataFrames?

Based on data frame concept in R and Python
  • Spark is the first to make this a declarative API

Integrates with other data science libraries
  • MLlib, GraphFrames, …

Google trends for “data frame”
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
Space Usage

Memory Usage when Caching

Data Size (GB)

DataFrame

RDD
Uptake

DataFrames were released in March 2015, but already see high use:

- 62% of users in 2015 survey use DataFrames
- 69% of users use Spark SQL
- SQL & Python are the top languages on Databricks
Other High-Level APIs

Machine Learning Pipelines
Modular API based on scikit-learn

GraphFrames
Relational + graph operations

Structured Streaming
Declarative streaming API in Spark 2.0

Many high-level data science APIs can be declarative
Changing Hardware
Hardware Trends

Storage

Network

CPU
Hardware Trends

2010

Storage  50+MB/s (HDD)

Network  1Gbps

CPU      ~3GHz
## Hardware Trends

<table>
<thead>
<tr>
<th>Category</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>Category</th>
<th>2010</th>
<th>2016</th>
<th>Difference</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, compute efficiency matters even for data-intensive apps
  • Getting harder with more diverse hardware, e.g. GPUs, FPGAs

Future: network cards $\approx$ DRAM bandwidth
Spark Effort: Project Tungsten

Optimize Apache Spark’s CPU and memory usage, via:

1. Runtime code generation
2. Exploiting cache locality
3. Off-heap memory management
Tungsten’s Binary Encoding

(123, “data”, “bricks”)
DataFrame Code / SQL

```java
df.where(df("year") > 2015)
```

Logical Expressions

```
GreaterThan(year#234, Literal(2015))
```

Low-level Bytecode

```
bool filter(Object baseObject) {
    int offset = baseOffset + bitSetWidthInBytes + 3*8L;
    int value = Platform.getInt(baseObject, offset);
    return value34 > 2015;
}
```

JVM intrinsic JIT-ed to pointer arithmetic
Recent Additions

Whole-stage code generation
- Fuse across multiple operators

Optimized input / output
- Apache Parquet + built-in cache

<table>
<thead>
<tr>
<th>Version</th>
<th>Rows/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spark 1.6</td>
<td>14M rows/s</td>
</tr>
<tr>
<td>Spark 2.0</td>
<td>125M rows/s</td>
</tr>
<tr>
<td>Parquet in 1.6</td>
<td>11M rows/s</td>
</tr>
<tr>
<td>Parquet in 2.0</td>
<td>90M rows/s</td>
</tr>
</tbody>
</table>
Not Limited to Spark

Results from Nested Vector Language (NVL) project at MIT

- HyPer Database
- GraphMat PageRank
- TensorFlow Word2Vec

Current systems
Hand tuned code
Challenges

How to get this high performance while keeping the ease of use for non-programmers?

Can optimizations compose across libraries / systems?
Cloud Delivery
The Public Cloud is Here

Many Fortune 100 companies have multiple PB of data in S3

Amazon Web Services up to $10B revenue

Especially attractive for big data

- 51% of respondents in 2015 Spark survey run on public cloud
Benefits

For cloud users:

• Purchase an end-to-end experience, not just bits
• Rapidly experiment with new solutions (same data & infrastructure)

For software vendors:

• Better products: end-to-end service, high visibility
• Fast iteration and uniform adoption
Challenges

Requires new way to build software that is not well understood by researchers (or traditional software companies)

- **Multi-tenant**: with untrusted tenants
- **Highly available**, yet with continuous updates
- **Highly monitored** for billing and security
Example Challenges

Deploying updates while keeping the service up
  • And rolling back if needed!

Knowing whether the service is up

Unexpected use, especially by code calling APIs

Performance isolation of tenants at all levels

Little academic research these
Example: Databricks

End-to-end data processing platform based around Apache Spark

Access control, collaboration, auditing, production workflows

200+ customers and thousands of individual users
Lessons

Cloud development model is superior
• Two week releases, immediate feedback, visibility

State management is very hard at scale
• Per-tenant configuration, local data, VM images, etc

Careful testing strategy is crucial
• Feature flags, stress tests, 70/20/10 testing pyramid

Design to maximize dogfooding
Research Perspective

Computer systems is largely a social field: about interactions between users ⇔ machines, users ⇔ users, and machines ⇔ machines.

Cloud greatly changes the way users develop and consume software.

Not much research beyond using it to parallelize stuff.
Example Research Problems

Composing security interfaces of different cloud providers
  • E.g. Databricks access controls + Amazon IAM

Deterministic updates and rollback for complex systems

“Elastic-first” systems for price and demand variability
Conclusion

Big data systems made great strides since they first came out
  + They’re used well beyond tech companies
  - Not fully keeping up with new users & hardware

The cloud offers fantastic opportunities for research
  + People can try your new thing in production right away!
  - Not much research fully embraces it
Thanks!

Databricks is Hiring
Full-timers and interns
matei@databricks.com