What's Changing in Big Data?

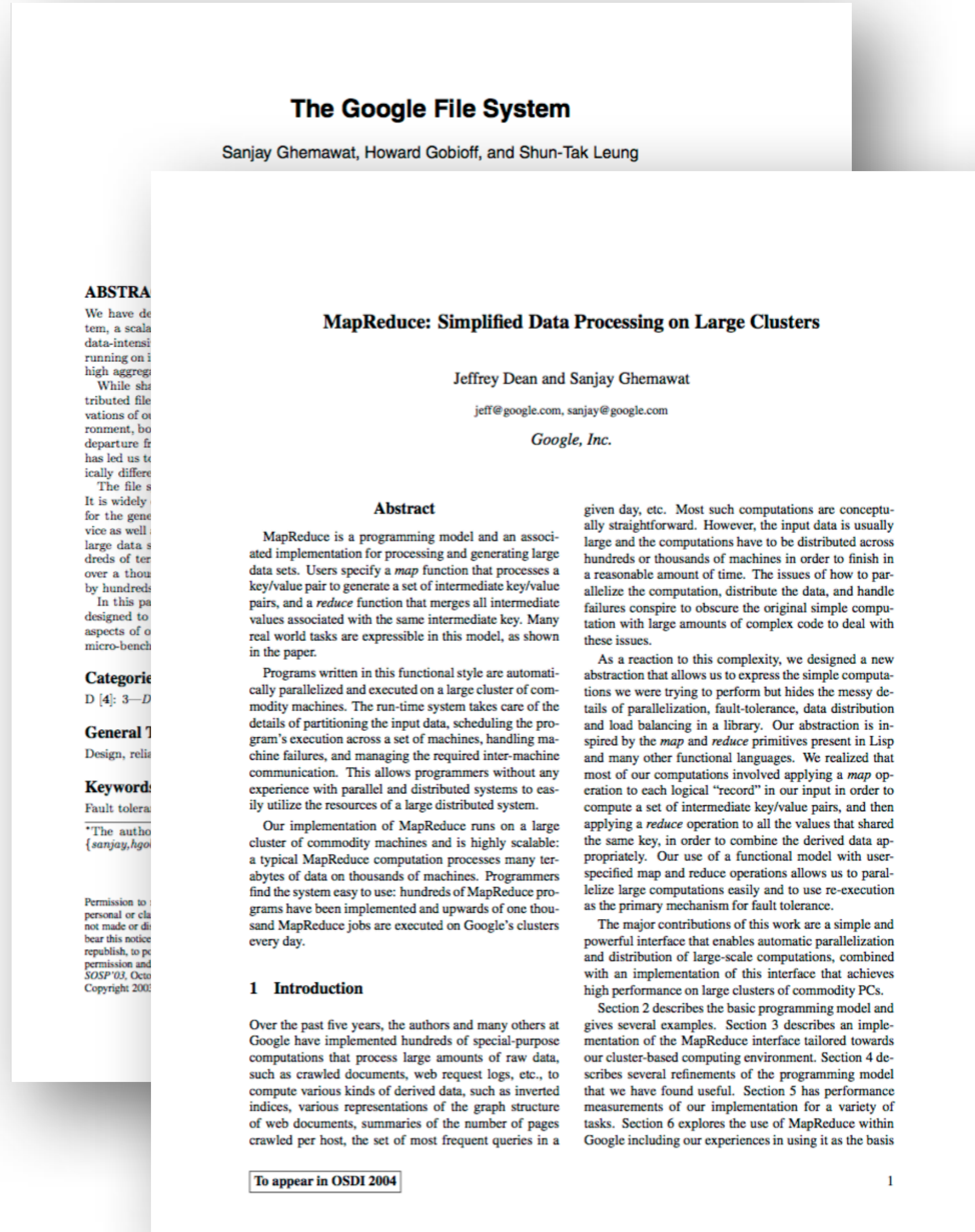
Matei Zaharia

June 21, 2016

Background

Big data systems became a popular research topic nearly 10 years ago

- Large-scale, commodity clusters



The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

To appear in OSDI 2004

1

What has changed since then?

databricks

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

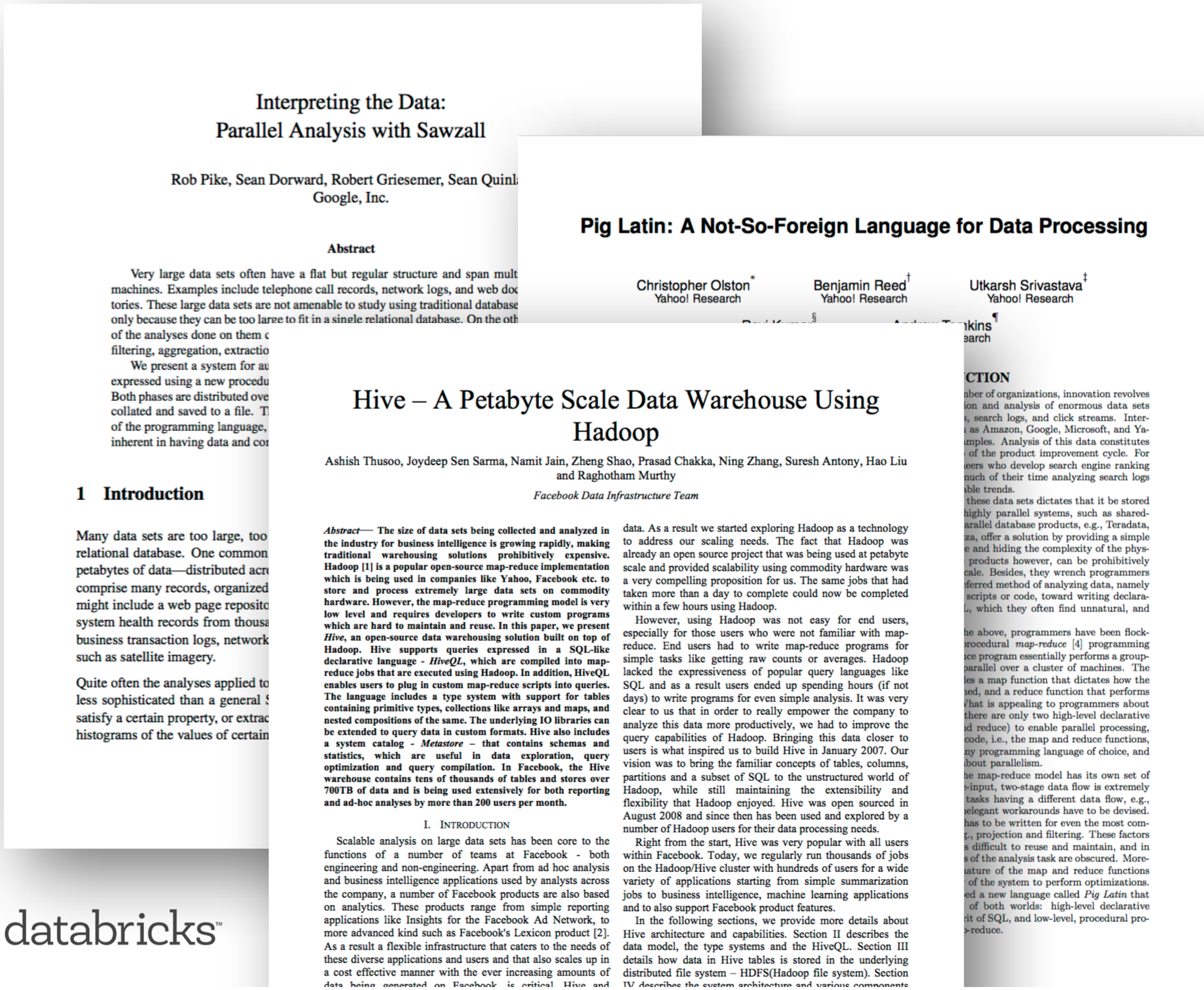
Initial Big Data Users

lying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks. 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)

databricks™

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!

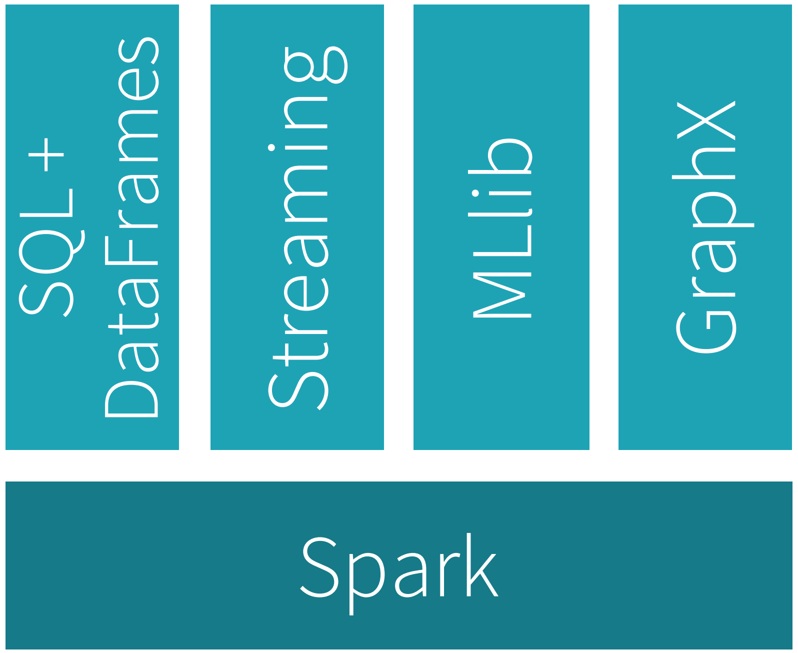
Case Study: Apache Spark

Cluster computing engine that generalizes MapReduce

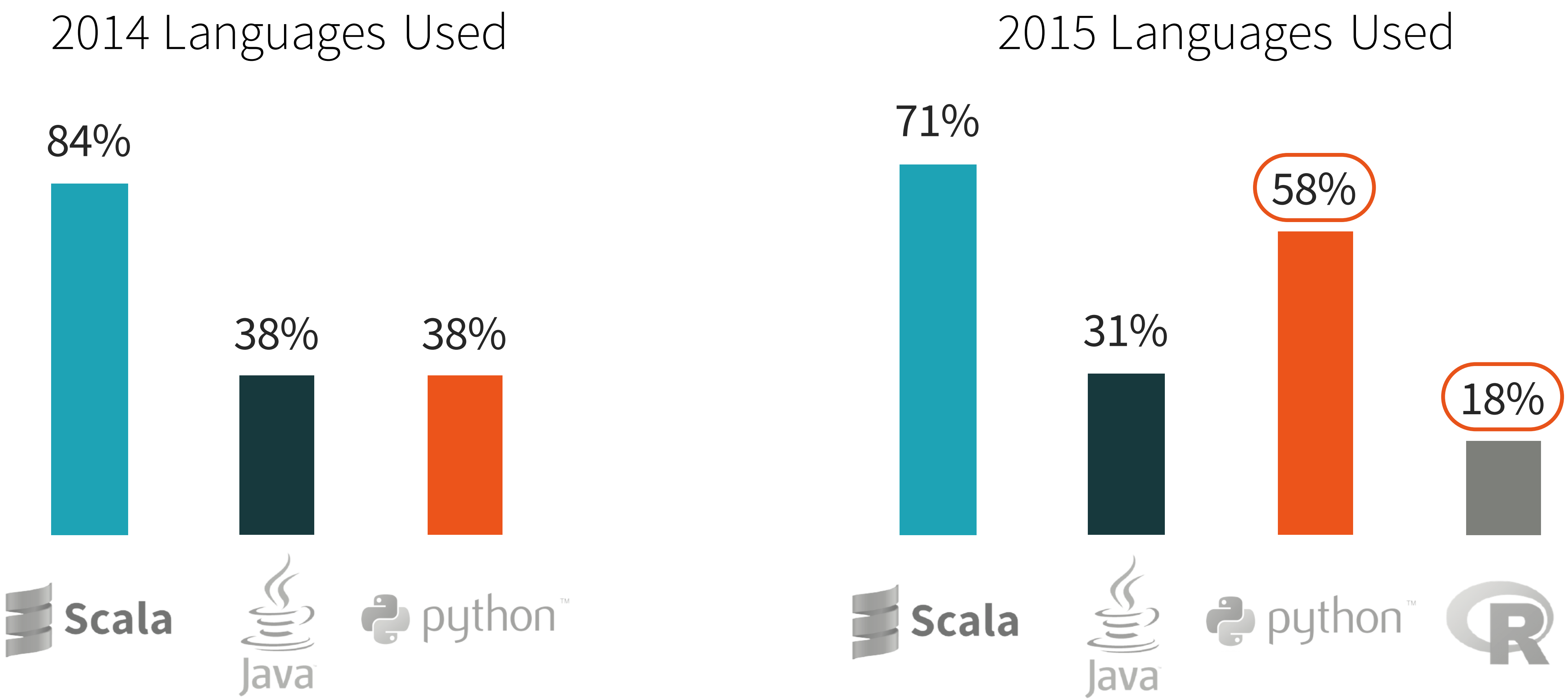
Collection of APIs and libraries

- APIs in Scala, Java, Python and R
- Streaming, SQL, ML, graph, …

1000+ deployments, max > 8000 nodes



Languages Used for Spark



databricks

Original Spark API

Functional API aimed at Java / Scala developers

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

```
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))
```

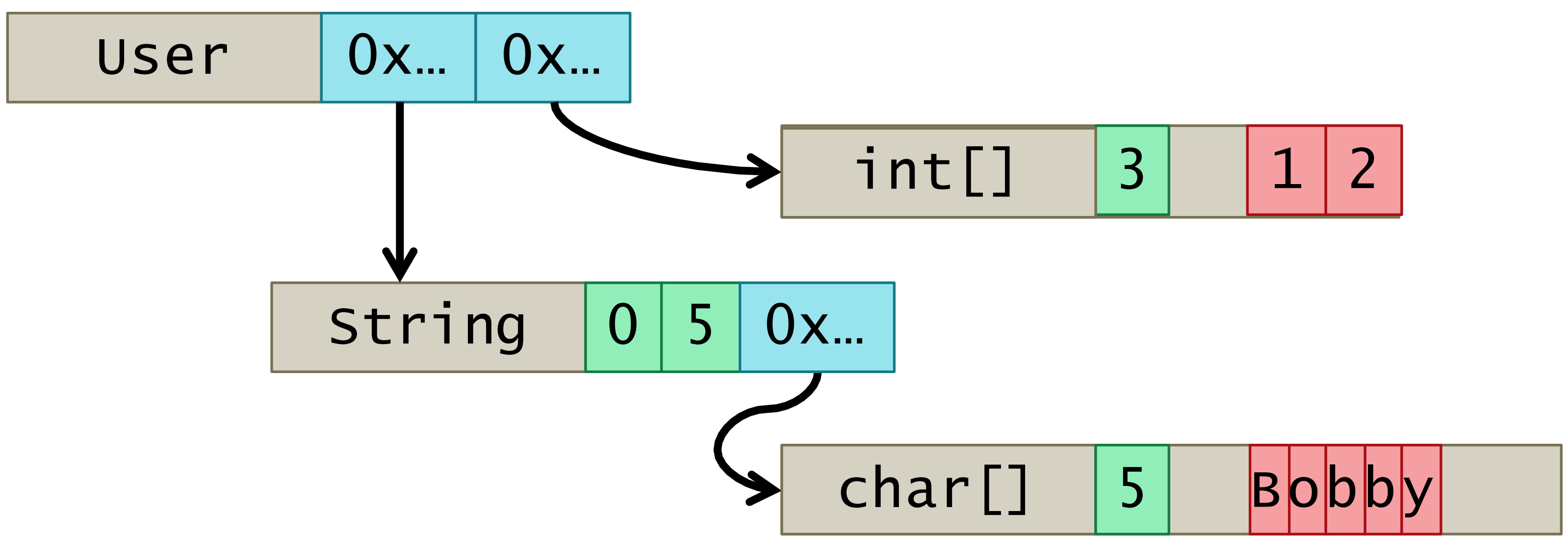
Materializes all groups as Seq[Int] objects

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))
```



databricks

Structured APIs: DataFrames + Spark SQL

databricks™

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

SIGMOD 2015

Spark SQL: Relational Data Processing in Spark

Michael Armbrust[†], Reynold S. Xin[†], Cheng Lian[†], Yin Huai[†], Davies Liu[†], Joseph K. Bradley[†], Xiangrui Meng[†], Tomer Kaftan[‡], Michael J. Franklin[†‡], Ali Ghodsi[†], Matei Zaharia[†*]

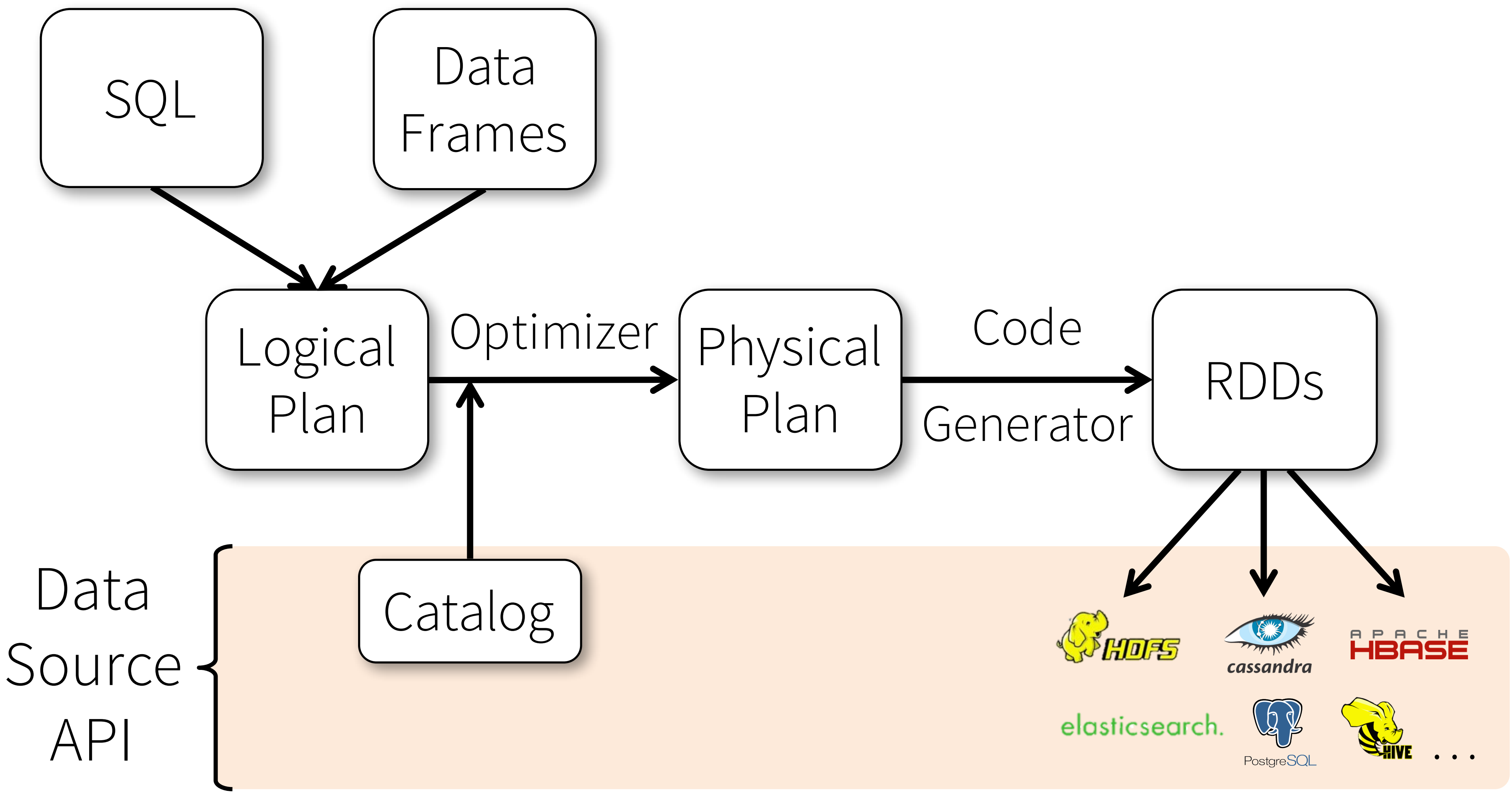
[†]Databricks Inc. [*]MIT CSAIL [‡]AMPLab, UC Berkeley

ABSTRACT

Spark SQL is a new module in Apache Spark that integrates relational processing with Spark's functional programming API. Built on our experience with Shark, Spark SQL lets Spark program-

While the popularity of relational systems shows that users often prefer writing declarative queries, the relational approach is insufficient for many big data applications. First, users want to perform ETL to and from various data sources that might be semi- or un-

Execution Steps



DataFrame API

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

```
val c = new HiveContext()
val users = c.sql("select * from users")

val massUsers = users(users("state") === "MA")

massUsers.count()

massUsers.groupBy("name").avg("age")

massUsers.map(row => row.getString(0).toUpper())
```

Expression AST

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”

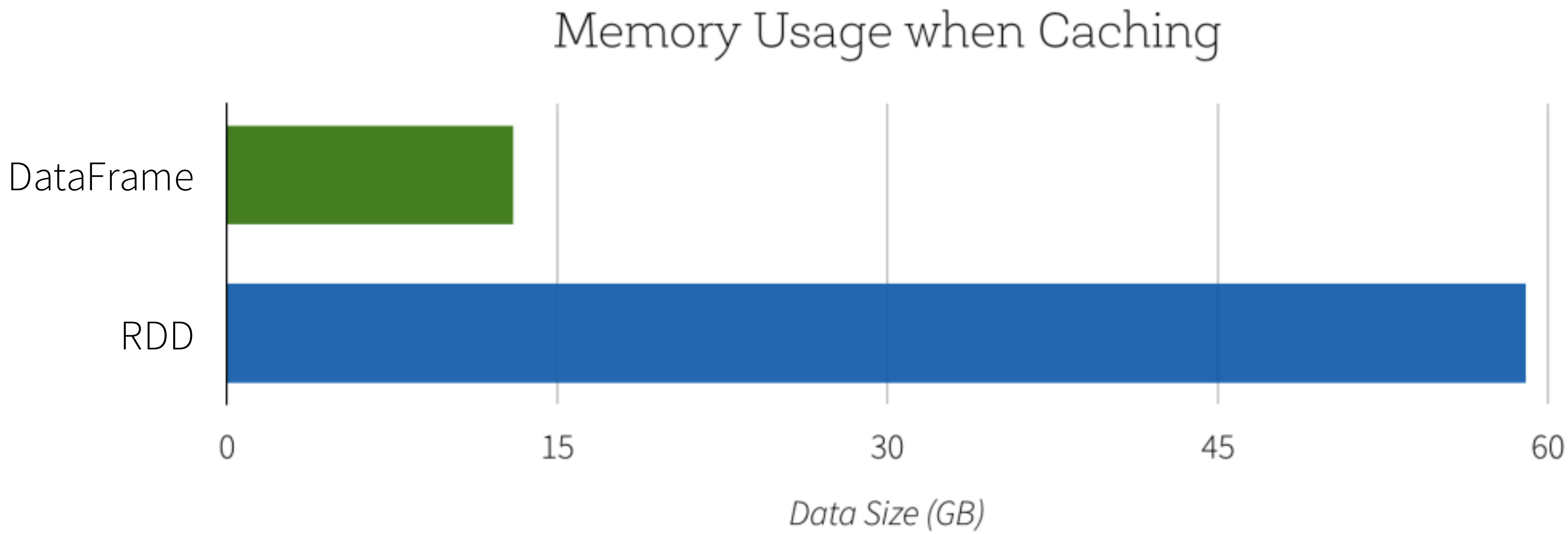
What Structured APIs Enable

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

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

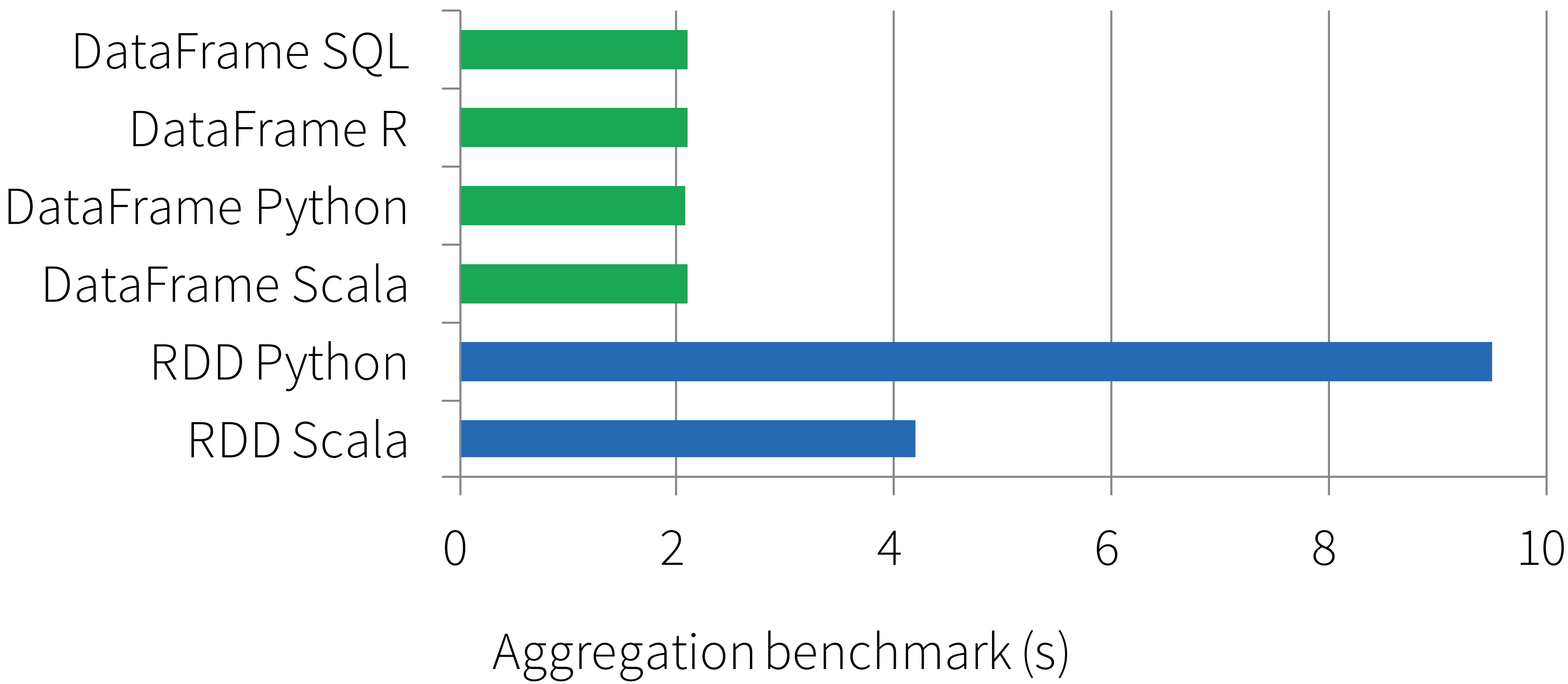
3. Runtime code generation

Space Usage



databricks

Performance



Uptake

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

62% of users in 2015 survey use DataFrames

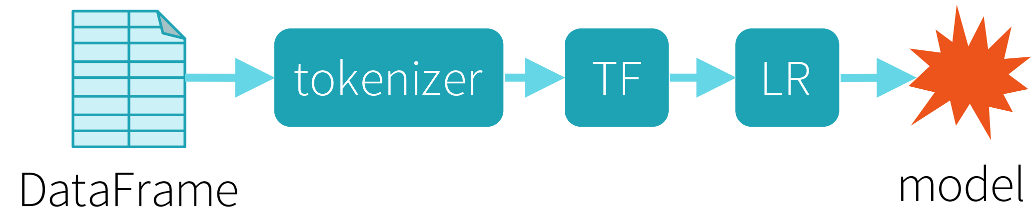
69% of users use Spark SQL

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

	2010	2016
Storage	50+MB/s (HDD)	500+MB/s (SSD)
Network	1Gbps	10Gbps
CPU	~3GHz	~3GHz

Hardware Trends

	2010	2016	
Storage	50+MB/s (HDD)	500+MB/s (SSD)	10x
Network	1Gbps	10Gbps	10x
CPU	~3GHz	~3GHz	☹

databricks

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 ≅ DRAM bandwidth

Spark Effort: Project Tungsten

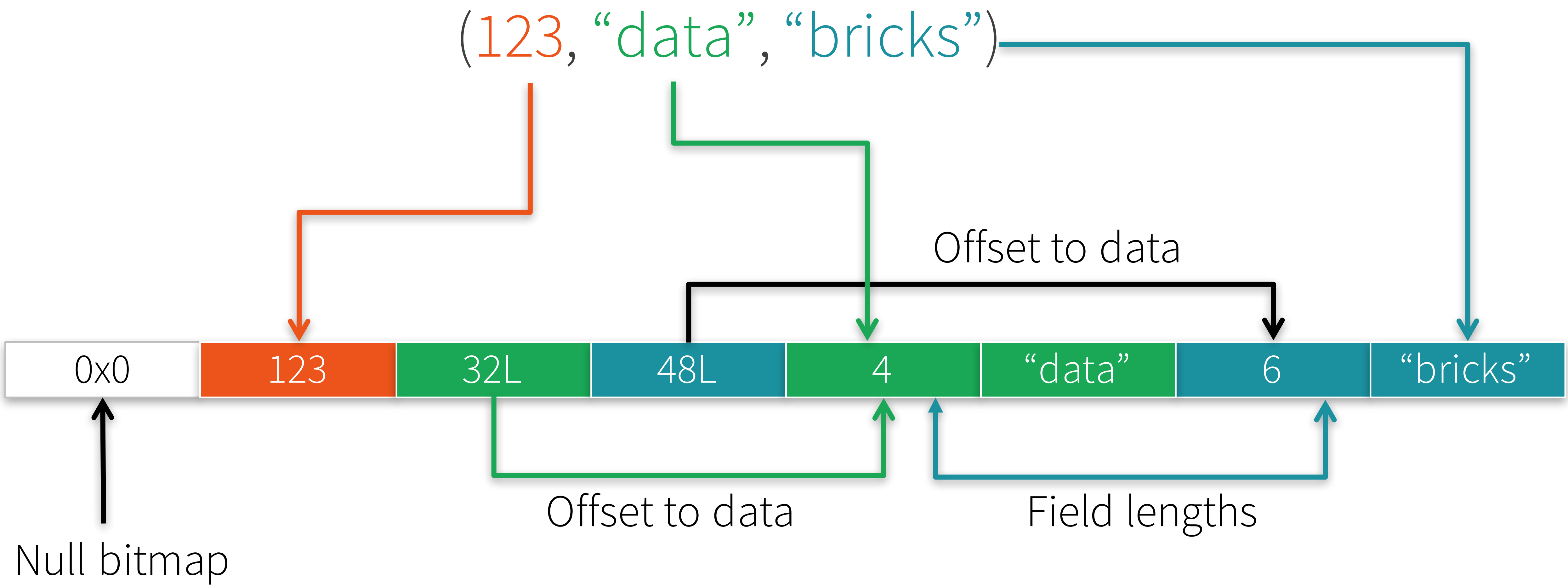
Optimize Apache Spark's CPU and memory usage, via:

(1) Runtime code generation

(2) Exploiting cache locality

(3) Off-heap memory management

Tungsten's Binary Encoding



databricks

Runtime Code Generation

DataFrame Code / SQL

```
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;
}
```

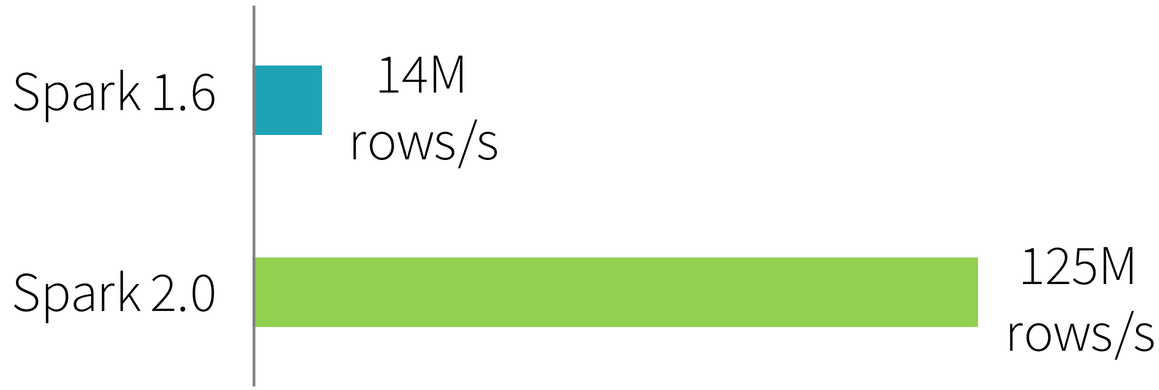
JVM intrinsic JIT-ed to pointer arithmetic

databricks™

Recent Additions

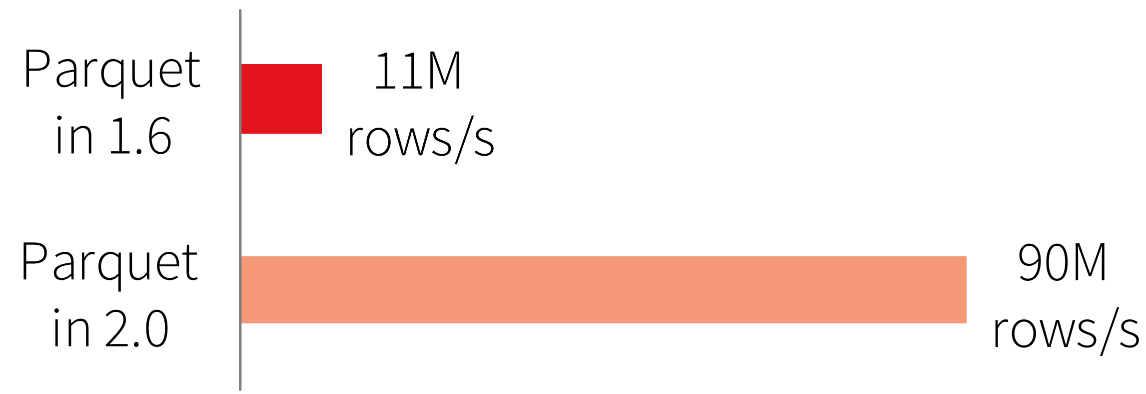
Whole-stage code generation

- Fuse across multiple operators



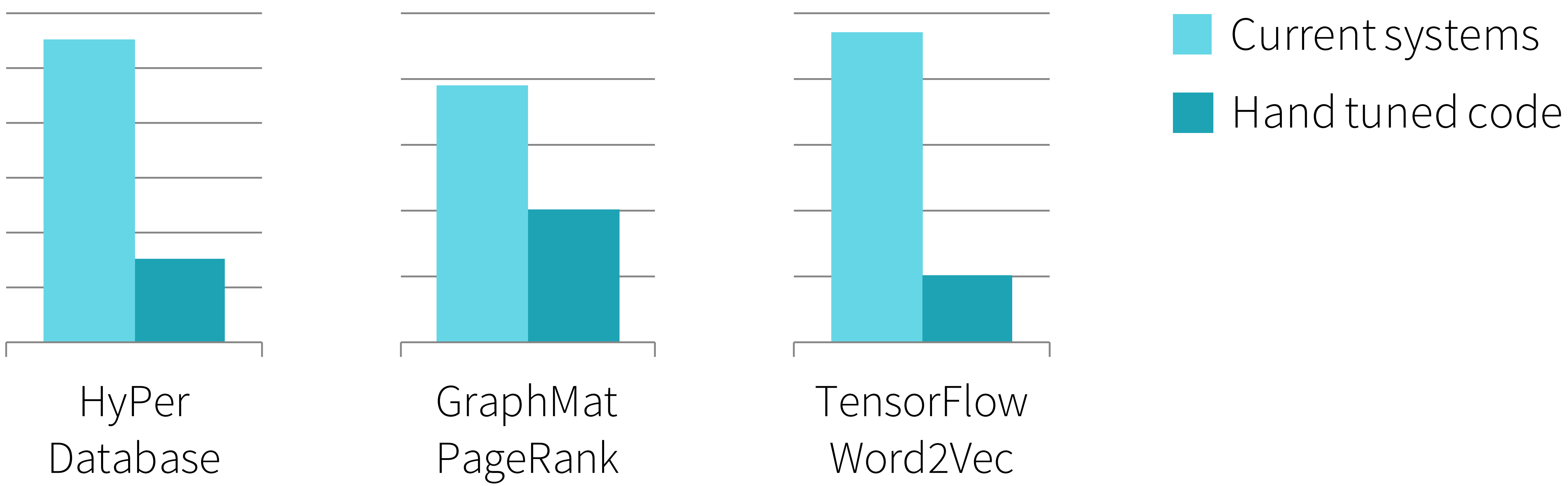
Optimized input / output

- Apache Parquet + built-in cache



Not Limited to Spark

Results from Nested Vector Language (NVL) project at MIT



databricks

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)

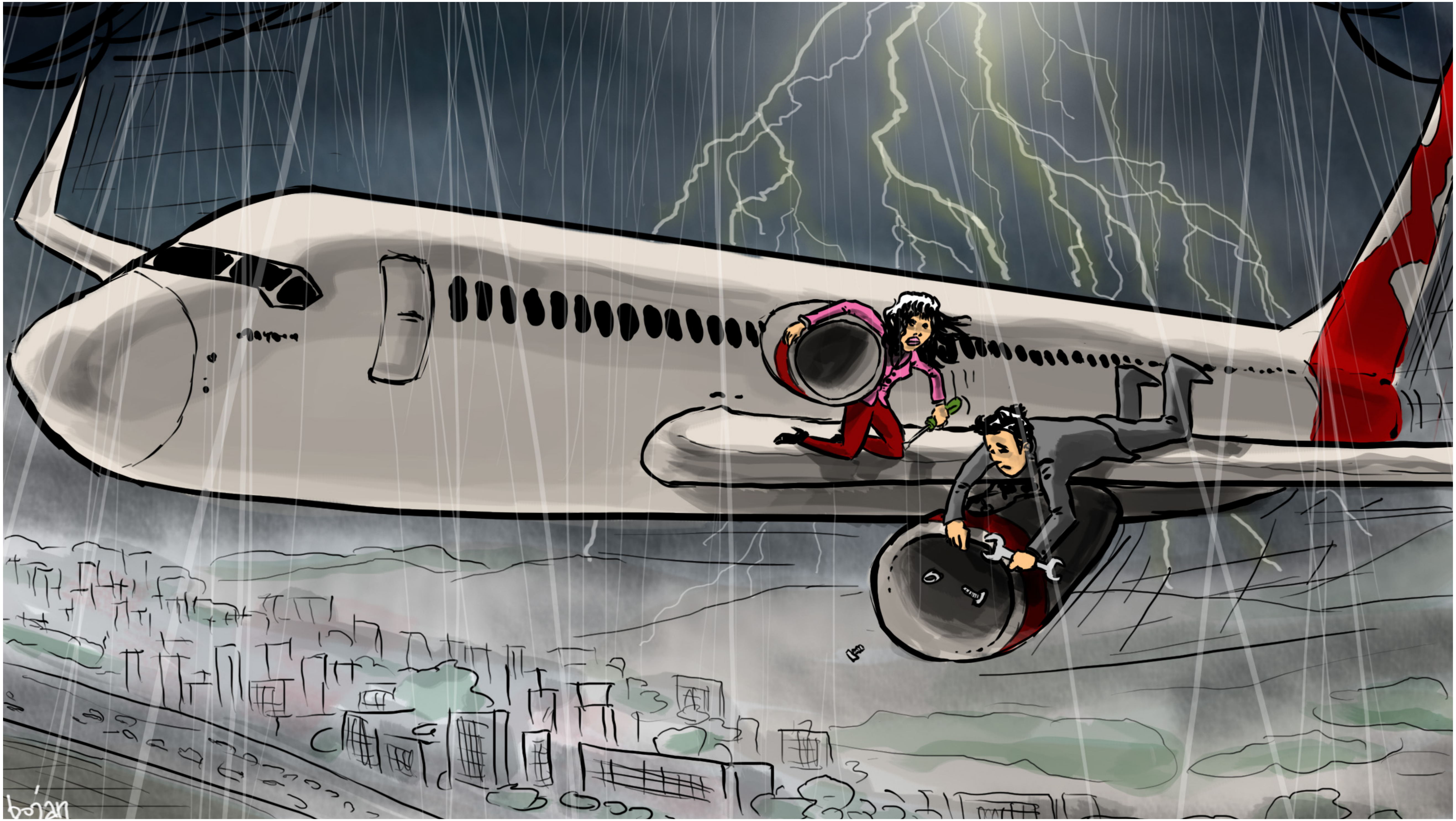
For software vendors:

- 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


bojan

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

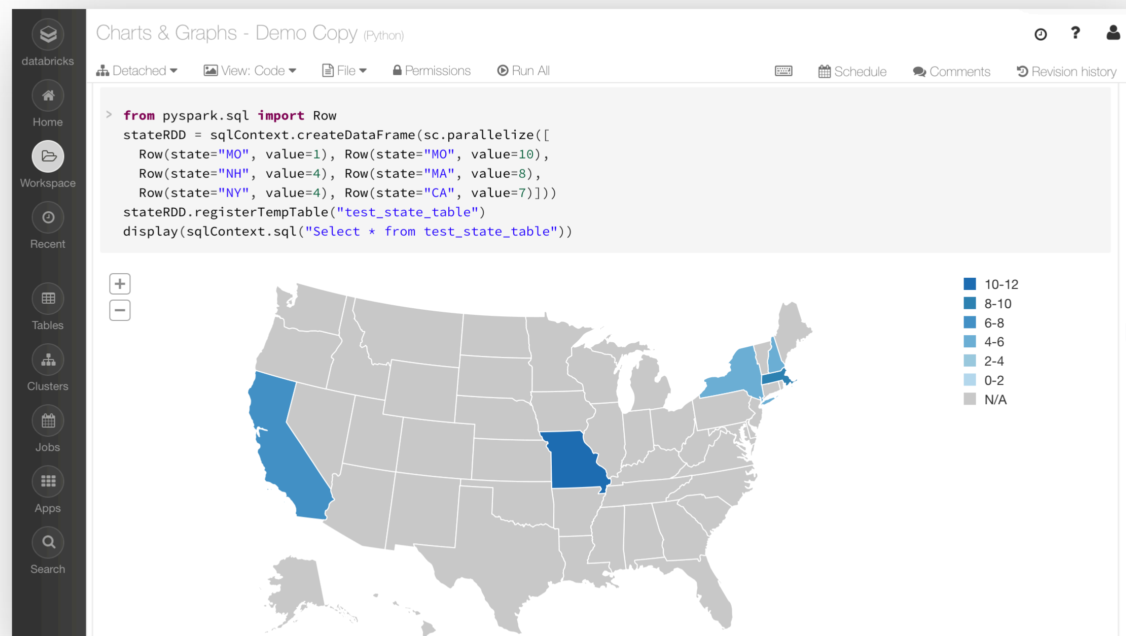
databricks

Example: Databricks

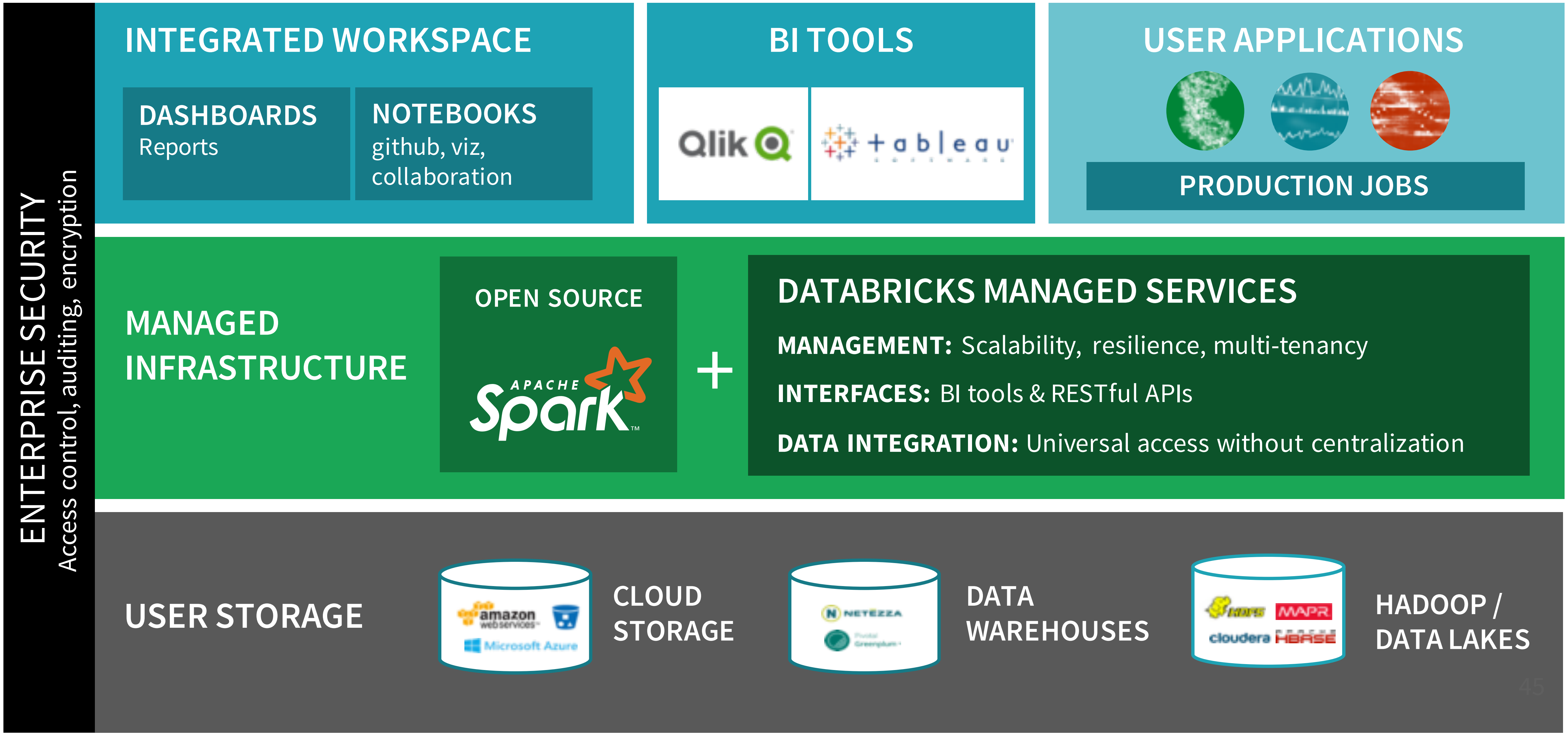
End-to-end data processing platform based around Apache Spark

Access control, collaboration, auditing, production workflows

200+ customers and thousands of individual users



databricks


ENTERPRISE SECURITY
Access control, auditing, encryption
INTEGRATED WORKSPACE
DASHBOARDS
Reports
NOTEBOOKS
github, viz, collaboration
BI TOOLS
Qlik
+ a b l e a u
USER APPLICATIONS
PRODUCTION JOBS
MANAGED INFRASTRUCTURE
OPEN SOURCE
APACHE Spark
+
DATABRICKS MANAGED SERVICES
MANAGEMENT: Scalability, resilience, multi-tenancy
INTERFACES: BI tools & RESTful APIs
DATA INTEGRATION: Universal access without centralization
USER STORAGE
amazon webservices
Microsoft Azure
CLOUD STORAGE
NETEZZA
DATA WAREHOUSES
MAPR
cloudera
HBASE
HADOOP / DATA LAKES

databricks

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