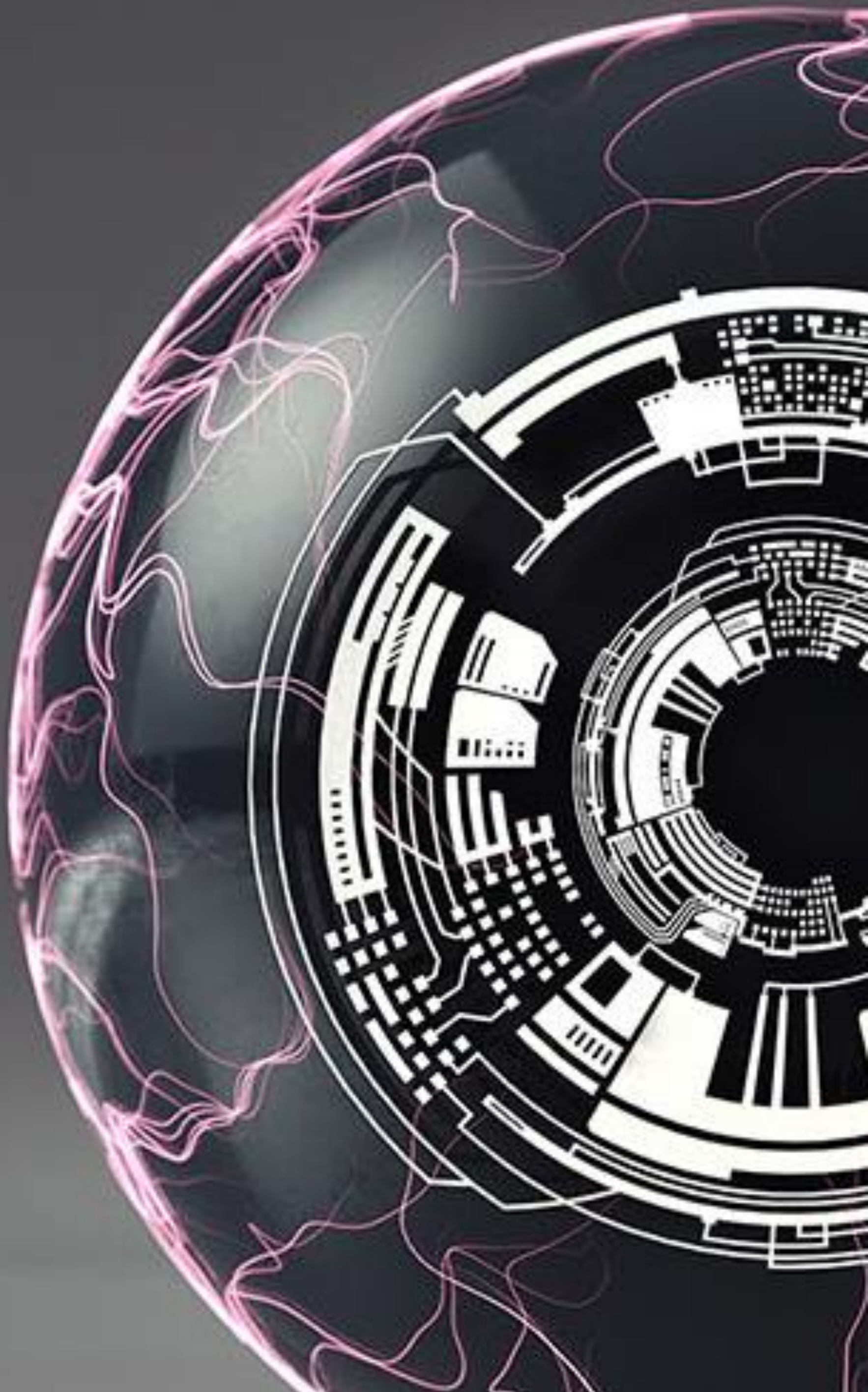
What’s Changing in Big Data?

Matei Zaharia

Stanford University



Background

The first big data systems were designed 10 years ago

What's changed since then?

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks

Michael Isard, Microsoft Research, Silicon Valley
Mihai Budiu, Microsoft Research, Silicon Valley
Yuan Yu, Microsoft Research, Silicon Valley
Andrew Birrell, Microsoft Research, Silicon Valley
Dennis Fetterly, Microsoft Research, Silicon Valley

ABSTRACT

Dryad is a general-purpose distributed execution engine for coarse-grain data-parallel applications. A Dryad application combines computational "vertices" with communication "channels" to form a dataflow graph. Dryad runs the application by executing the vertices of this graph on a set of available computers, communicating as appropriate through files, TCP pipes, and shared-memory FIFOs.

The vertices provided by the application developer are quite simple and are usually written as sequential programs with no thread creation or locking. Concurrency arises from Dryad scheduling vertices to run simultaneously on multiple computers, or on multiple CPU cores within a computer. The application can discover the size and placement of data at run time, and modify the graph as the computation progresses to make efficient use of the available resources.

Dryad is designed to scale from powerful multi-core single computers, through small clusters of computers, to data centers with thousands of computers. The Dryad execution engine handles all the difficult problems of creating a large distributed, concurrent application: scheduling the use of computers and their CPUs, recovering from communication or computer failures, and transporting data between vertices.

Categories and Subject Descriptors

D.1.3 [PROGRAMMING TECHNIQUES]: Concurrent Programming—*Distributed programming*

General Terms

Performance, Design, Reliability

Keywords

Concurrency, Distributed Programming, Dataflow, Cluster Computing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'07, March 21–23, 2007, Lisboa, Portugal.
Copyright 2007 ACM 978-1-59593-636-3/07/0003 ...$5.00.

1. INTRODUCTION

The Dryad project addresses a long-standing problem: how can we make it easier for developers to write efficient parallel and distributed applications? We are motivated both by the emergence of large-scale internet services that depend on clusters of hundreds or thousands of general-purpose servers, and also by the prediction that future advances in local computing power will come from increasing the number of cores on a chip rather than improving the speed or instruction-level parallelism of a single core [3]. Both of these scenarios involve resources that are in a single administrative domain, connected using a known, high-performance communication topology, under centralized management and control. In such cases many of the hard problems that arise in wide-area distributed systems may be sidestepped: these include high latency and unreliable networks, control of resources by separate federated or competing entities, and issues of identity for authentication and access control. Our primary focus is instead on the simplicity of the programming model and the reliability, efficiency and scalability of the applications.

For many resource-intensive applications, the simplest way to achieve scalable performance is to exploit data parallelism. There has historically been a great deal of work in the parallel computing community both on systems that automatically discover and exploit parallelism in sequential programs, and on those that require the developer to explicitly expose the data dependencies of a computation. There are still limitations to the power of fully-automatic parallelization, and so we build mainly on ideas from the latter research tradition. Condor [37] was an early example of such a system in a distributed setting, and we take more direct inspiration from three other models: shader languages developed for graphic processing units (GPUs) [30, 36], Google's MapReduce system [16], and parallel databases [18]. In all these programming paradigms, the system dictates a communication graph, but makes it simple for the developer to supply subroutines to be executed at specified graph vertices. All three have demonstrated great success, in that large numbers of developers have been able to write concurrent software that is reliably executed in a distributed fashion.

We believe that a major reason for the success of GPU shader languages, MapReduce and parallel databases is that the developer is explicitly forced to consider the data parallelism of the computation. Once an application is cast into this framework, the system is automatically able to provide the necessary scheduling and distribution. The developer

My Perspective

Open source processing engine and set of libraries

Cloud service based on Spark

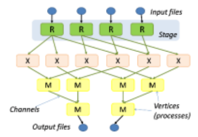
Three Key Changes

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



Dryad

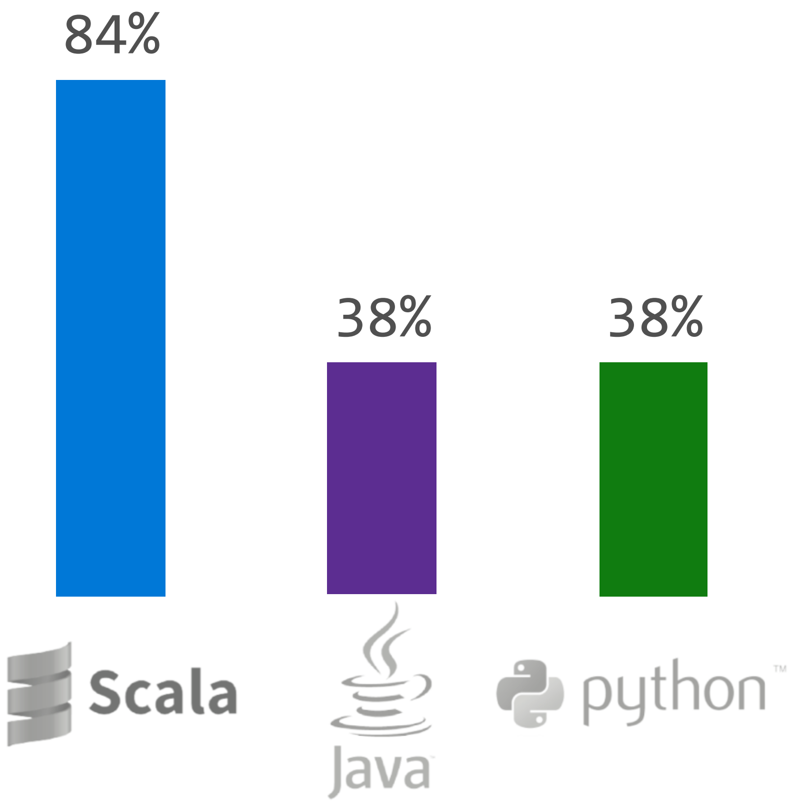
New users: data scientists & analysts

- SQL-like and scripting languages
- BI tools, e.g. Tableau

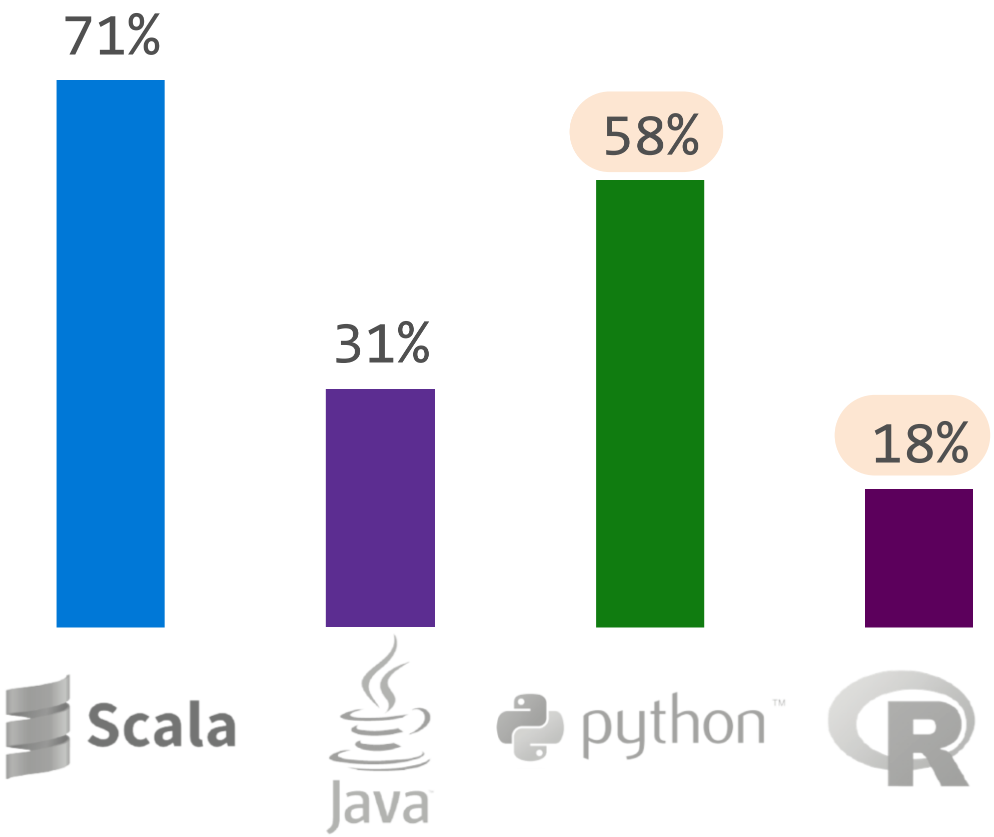
Pig

Example: Languages Used for Spark

2014 Languages Used



2015 Languages Used



Original Spark API

Functional API targeting Java / Scala developers

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

```
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?

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

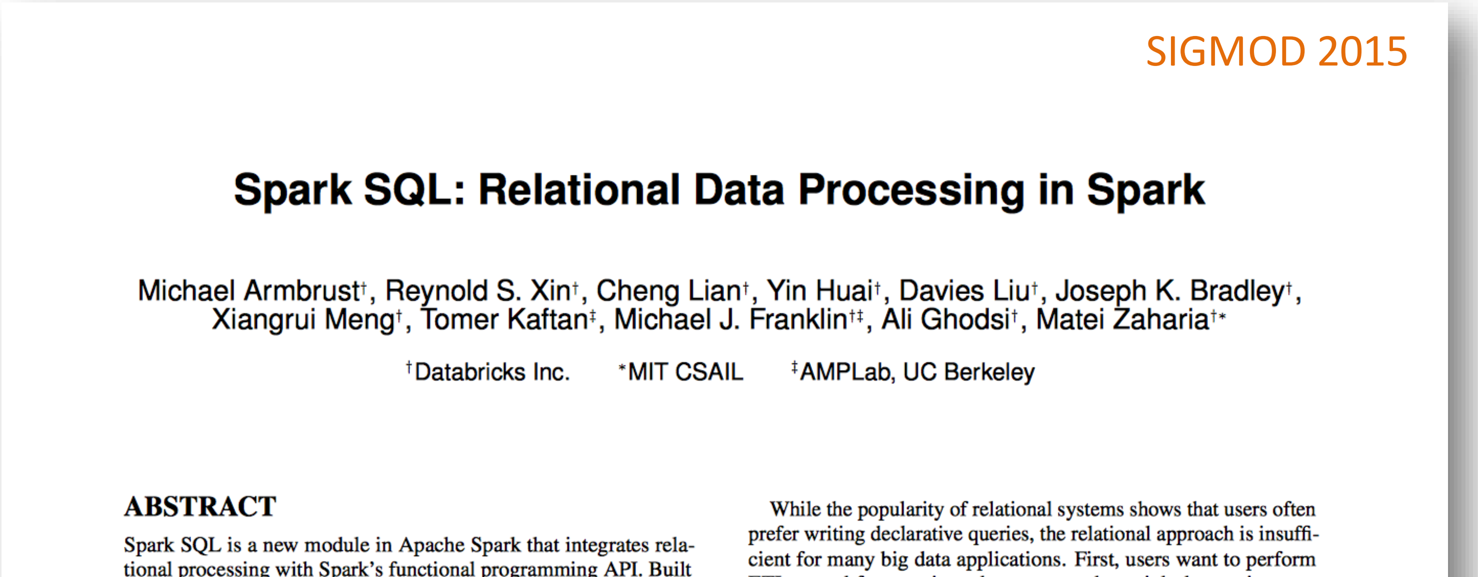
Then promptly aggregates them

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


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[†*]

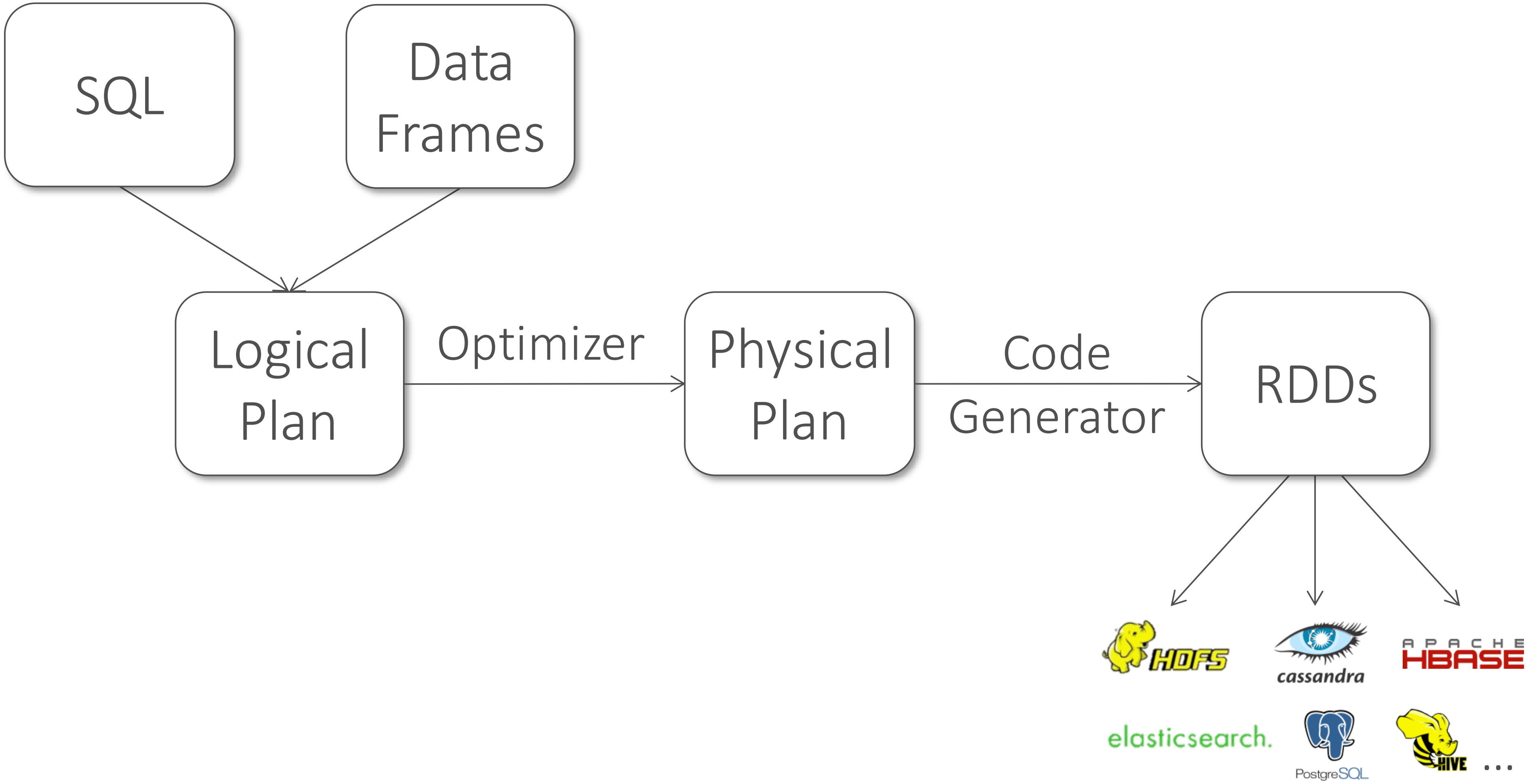
[†]Databricks Inc. [*]MIT CSAIL [‡]AMPLab, UC Berkeley

ABSTRACT

Spark SQL is a new module in Apache Spark that integrates relational processing with Spark's functional programming API. Built

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

Execution Steps



Programming Model

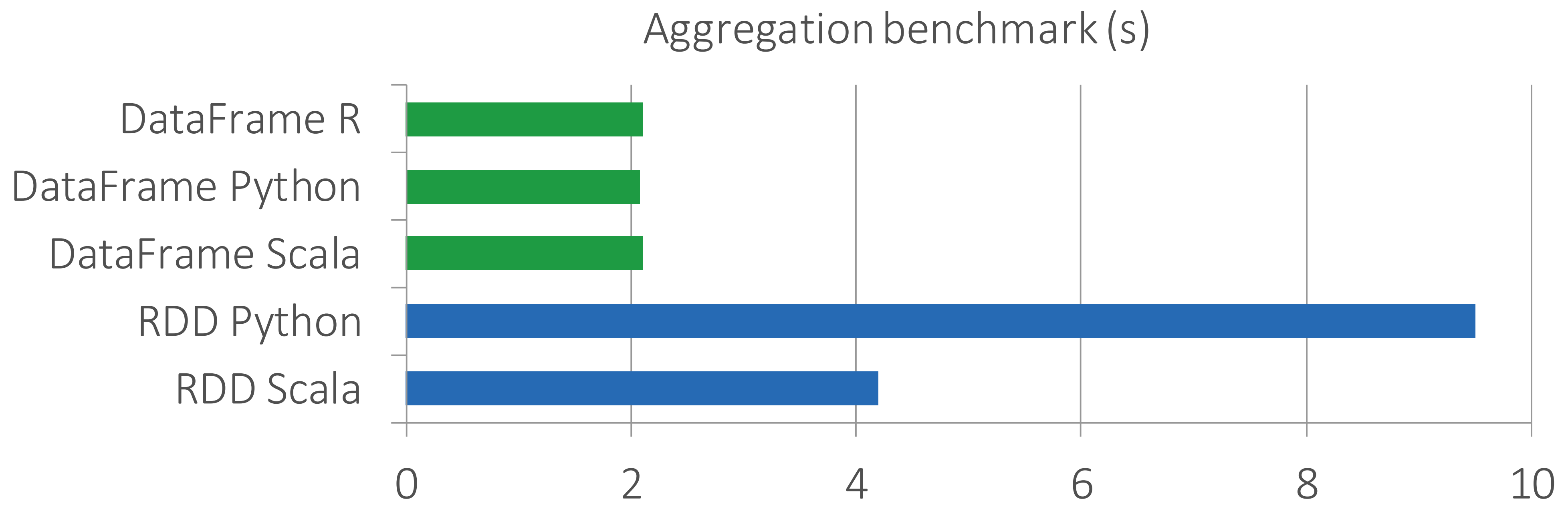
DataFrames hold rows with a known schema and offer relational ops through a DSL

```
users = ctx.sql("select * from hive.users")

ca_users = users[users.state == "CA"]

ca_users.count()

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

ca_users.map(lambda row: row.name.upper())
```

Expression AST

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 *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

	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	☹

Summary

In 2005-2010, I/O was the name of the game

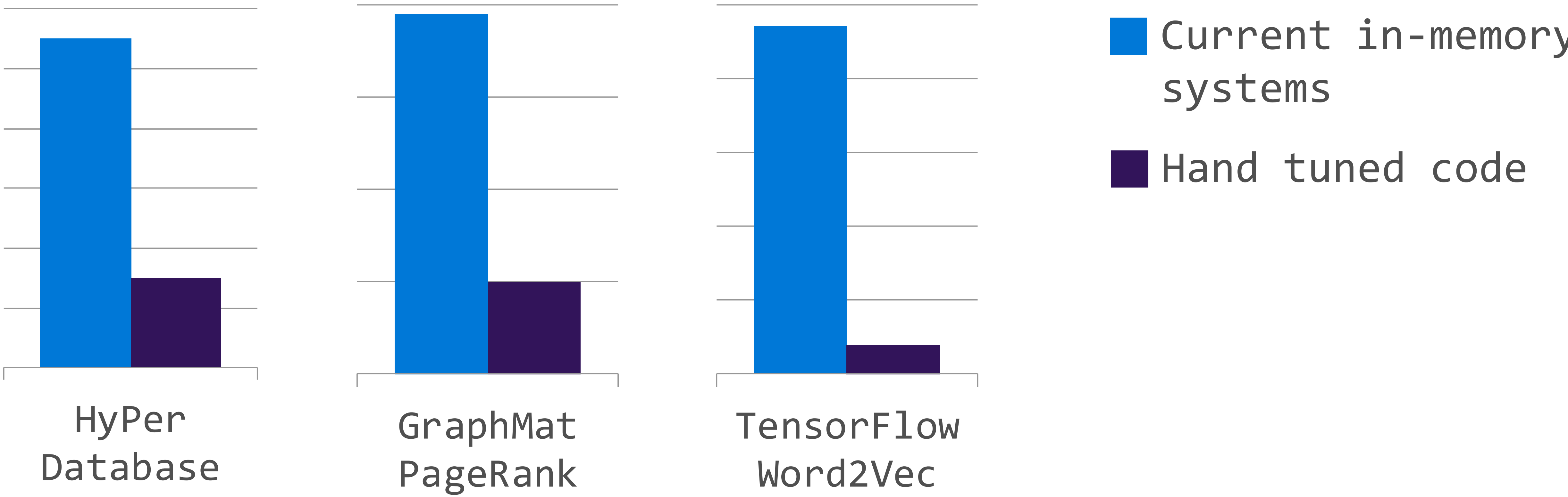
- Network locality, compression, in-memory caching

Now, CPU and DRAM are often bottlenecks

- Many current systems are 2-10x off peak performance

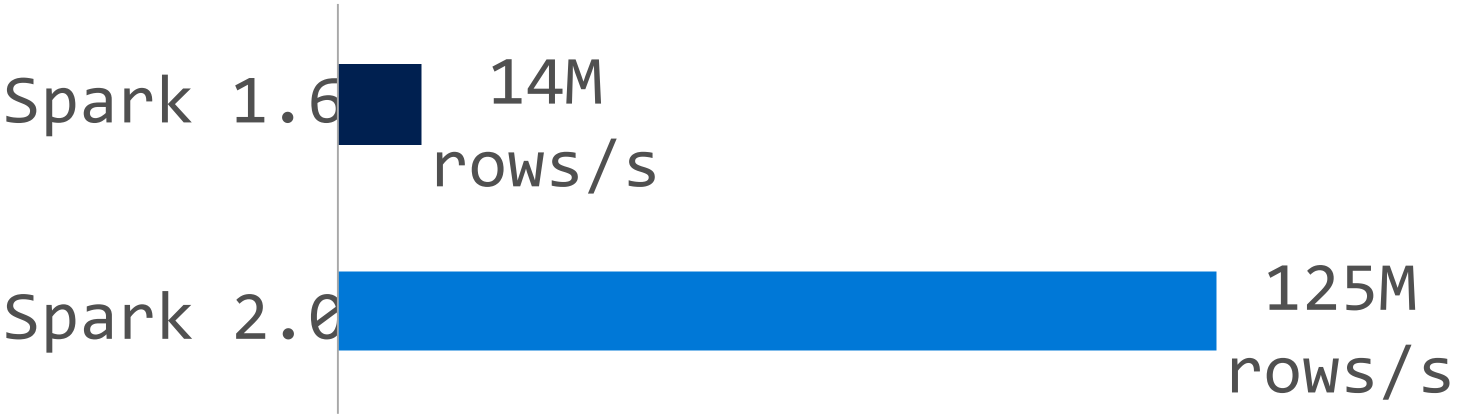
In-Memory Performance Gap

Results from Nested Vector Language (NVL) project at MIT



Spark Effort: Project Tungsten

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



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