Data Science

Introduction to Big Data processing in Apache Spark

Maciej Malawski
AGH University of Science and Technology
Department of Computer Science,

al. Mickiewicza 30, 30-059 Kraków, Poland

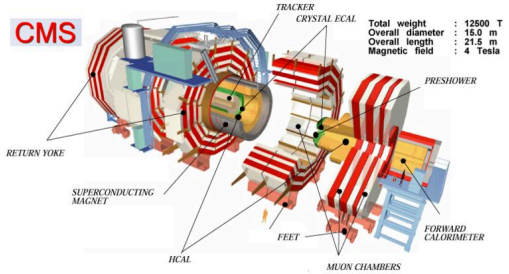
Outline

- Big Data processing - problems
- Why parallel processing
- Map Reduce model
- Spark, RDD and DataFrames
- Big Data in clouds
- Examples from CERN

Big data challenges and solutions

Big Data – example challenges

- Social network: Facebook
  - 2.7 bilion ”Like” actions and 300 million photos per day
  - 500+ TB processed each day
  - 100 PB stored in a single Hadoop disk cluster
- Science: CERN
  - 1 PB raw physics data / second
  - 200+ PB data stored on tapes, 100+ PB stored on disks



Limits of processing resources

- The biggest HDD Ever: Samsung PM1633a
  - 16 TB,
  - 1200MB/s read
  - 900MB/s write speed
- The fastest CPU: Intel® Core™ i7-8086K Processor
  - 6 Cores
  - Max Frequency: 5.00 GHz
  - Performance: 147.42 GFLOPS

Source: https://hddmag.com/top-x-largest-hard-drives/

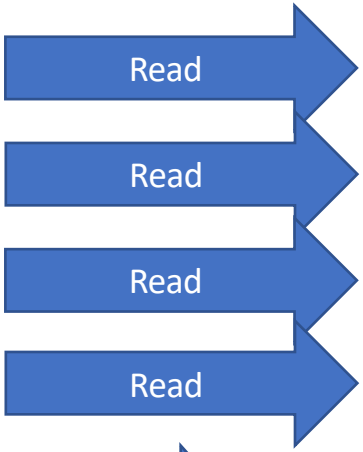
How much time to process 1 TB of data

Process → 1 TB / 150 GFLOP/s = 6 s

Read → 1 TB / 1200 MB/s = 833 s
= 13 minutes

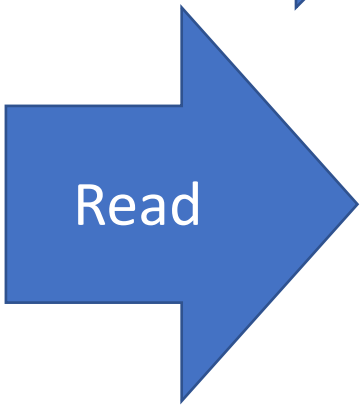
Answer: parallel processing

4



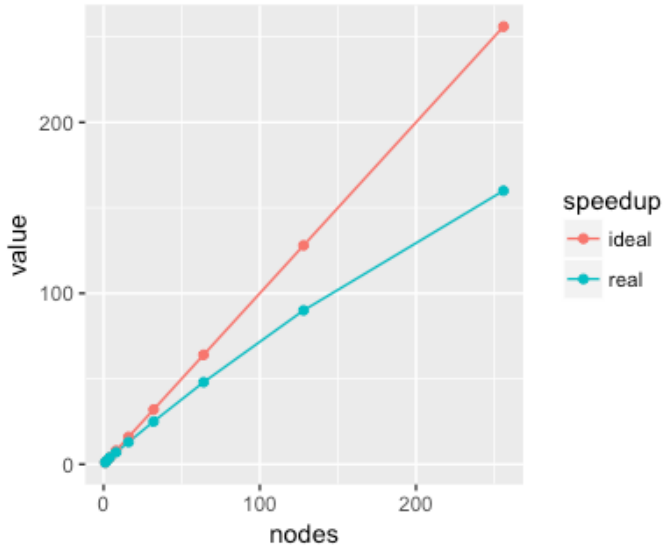
1 TB / 1200 MB/s /4 = **208 s**

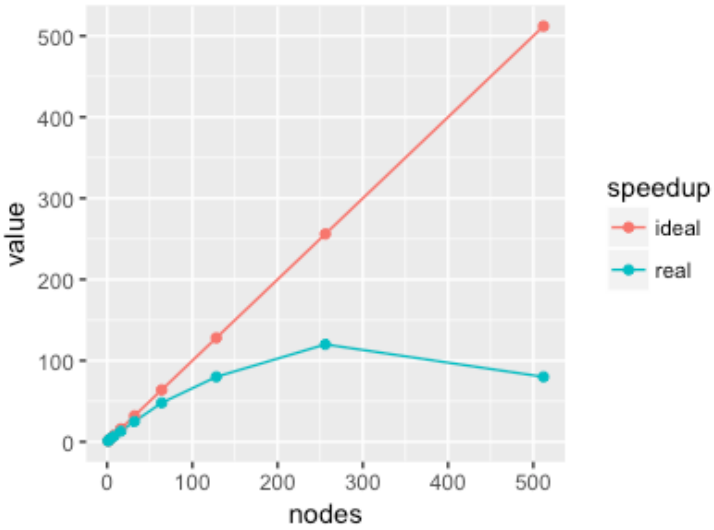
128



1 TB / 1200 MB/s / 128 = **6 s**

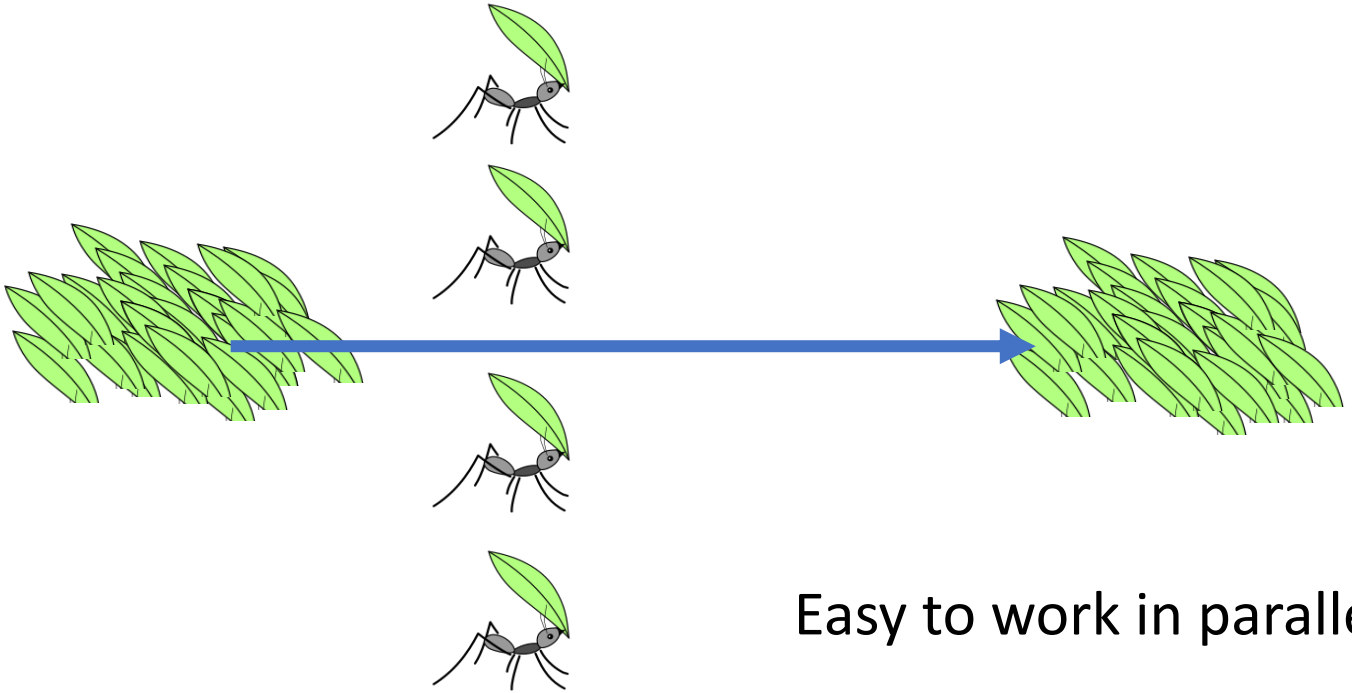
Speedup





- Speedup:
  - time on N nodes / time on 1 node
- Ideal: linear
- Real: less

Many small tasks

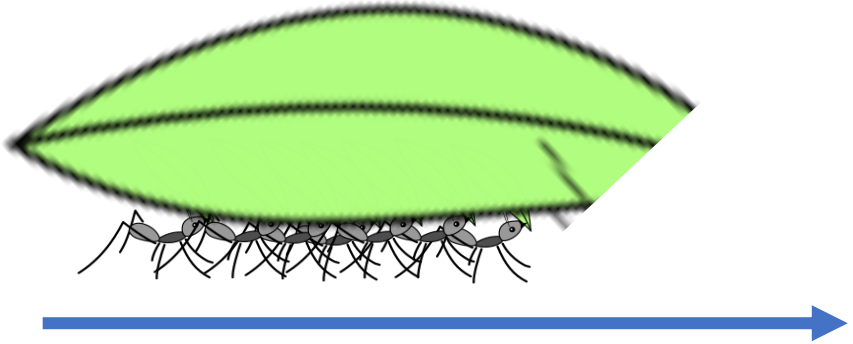


Easy to work in parallel

Many small tasks

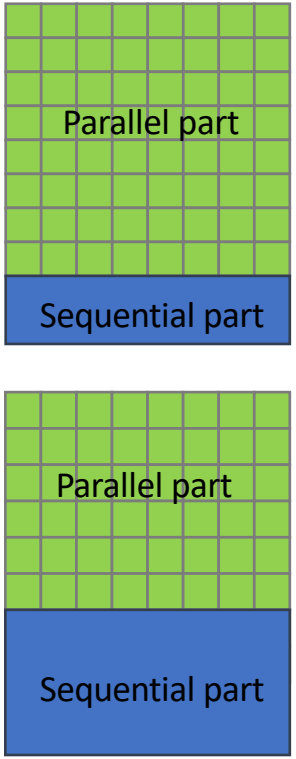
Easy to work in parallel

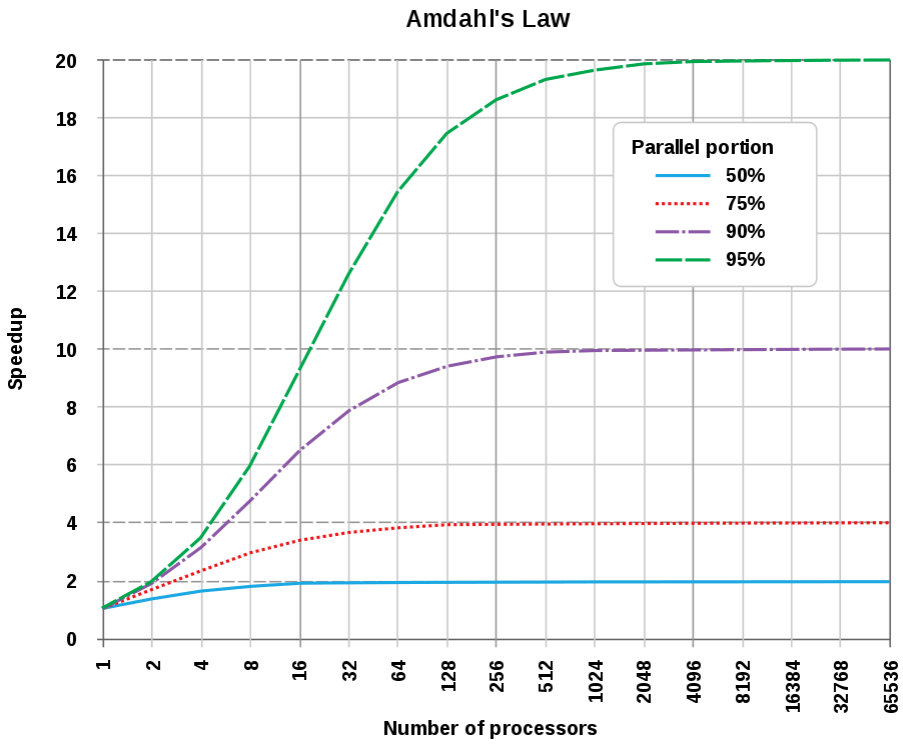
Big tasks



Not faster!

Amdahl’s Law


Parallel part
Sequential part
Parallel part
Sequential part


Amdahl's Law
Speedup
Number of processors
Parallel portion
50%
75%
90%
95%
0
2
4
6
8
10
12
14
16
18
20
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536

Map-Reduce

MapReduce

- Jeff Dean, Sanjay Ghemawat, Google 2004
- Two simple operations:
  - Map: `{a, b, c} -> {m(a), m(b), m(c)}`
  - Reduce: `{a, b, c} -> r(a, b, c)`
- Example:
  - Map: `x -> 2 * x`
    `{1, 2, 3} -> {2, 4, 6}`
  - Reduce: `sum: {x, y} -> x + y`
    `{2, 4, 6} -> 12`

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.

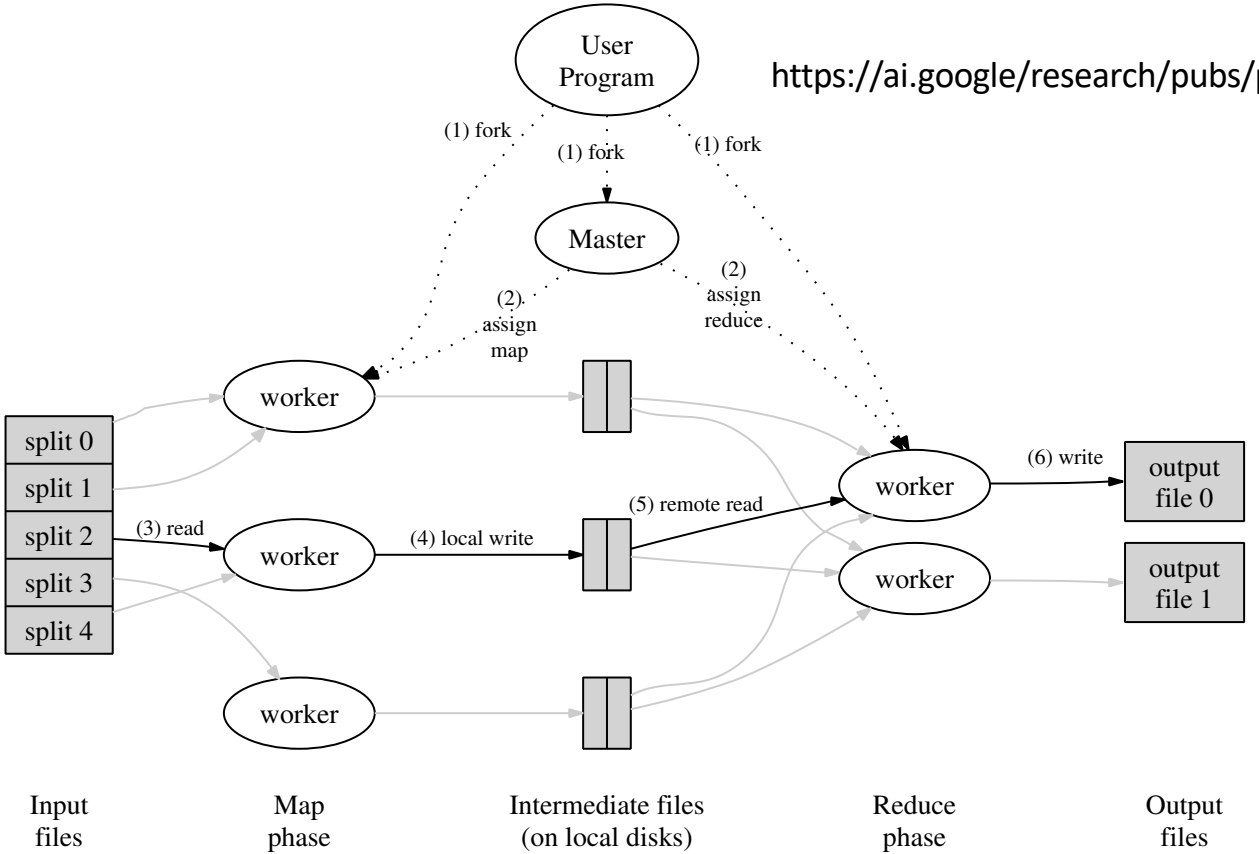
The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

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

Parallel execution of MapReduce on clusters

https://ai.google/research/pubs/pub62

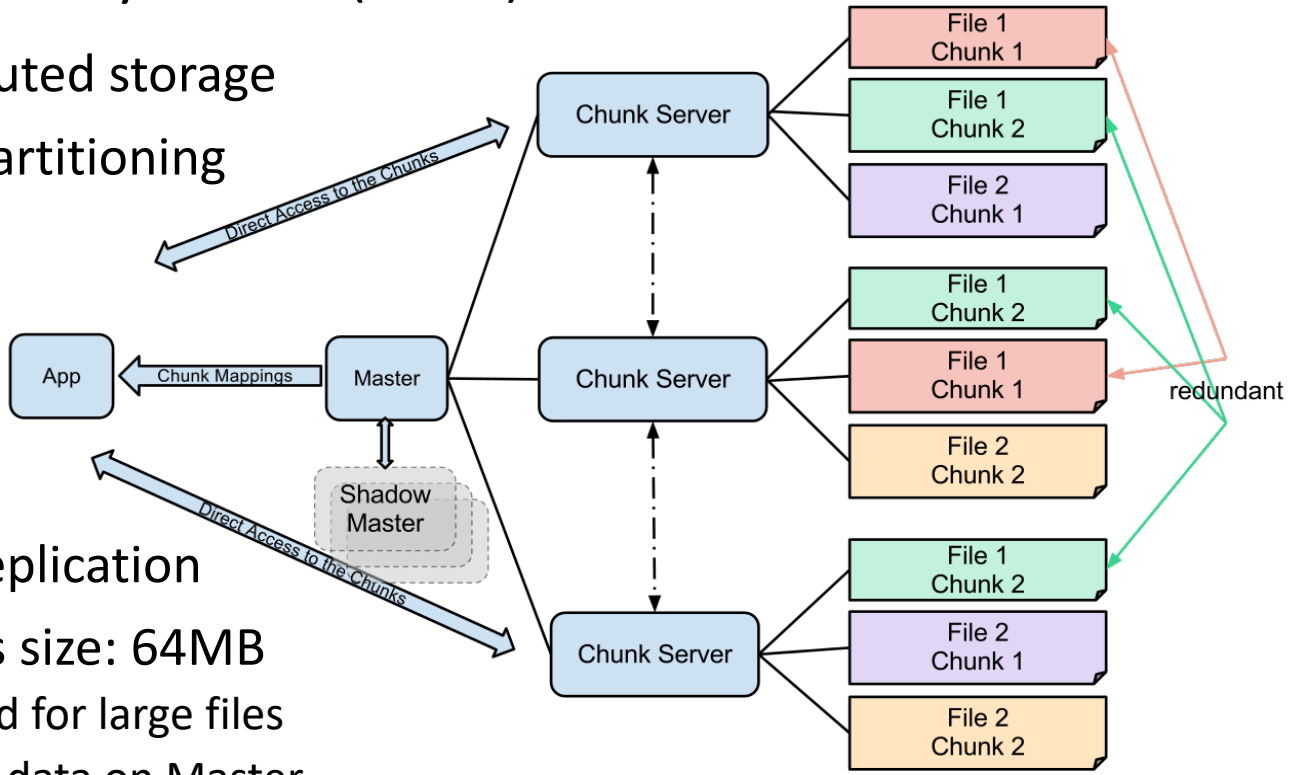


MapReduce examples

- **Word Count**: count occurrences of words in the text
- **Distributed Grep**: match lines containing a string
- **Count of URL Access Frequency**: processing logs
- **Reverse Web-Link Graph**: build a list of {target, list (source)} pairs
- **Term-Vector per Host**: vector of ⟨word, frequency⟩ per Host
- **Inverted Index**: list of ⟨word, list(document ID)⟩ pairs
- **Distributed Sort**: use of partitioning and ordering

Google File System (GFS)

- Distributed storage
- Data partitioning

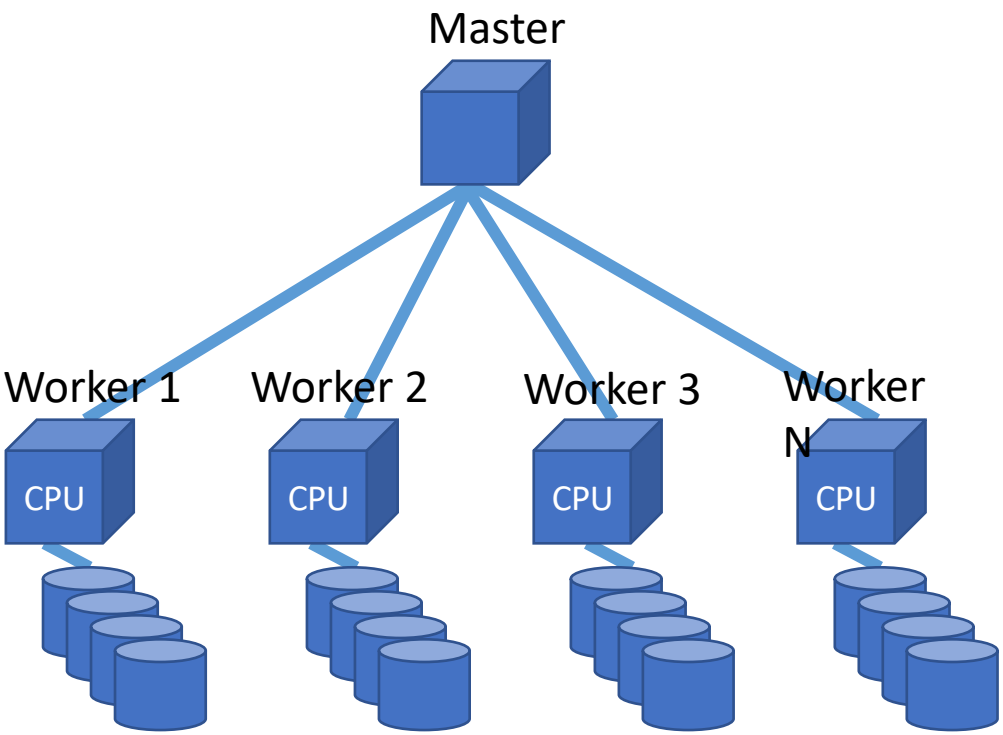


- Data replication
- Chunks size: 64MB
  - Good for large files
  - Less data on Master

https://ai.google/research/pubs/pub51

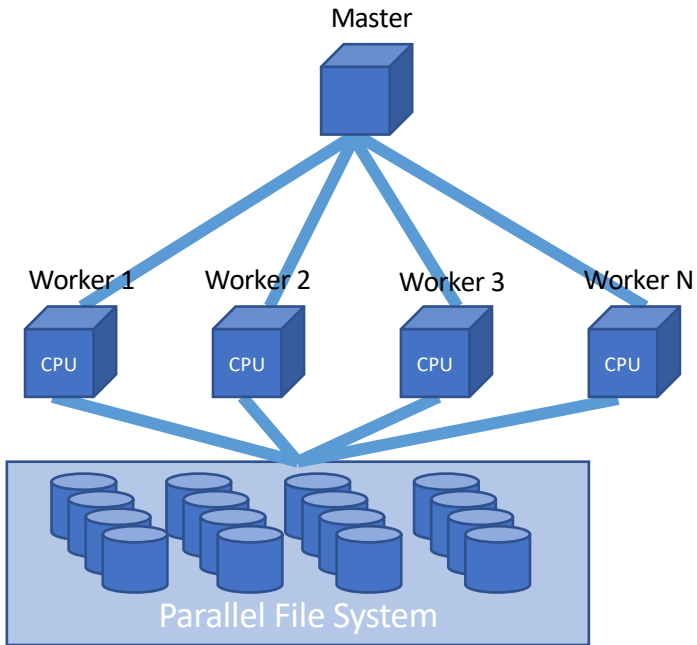
Architecture of MapReduce cluster

- Just a Bunch of Disks (JBOD)
- Commodity hardware
- Distributed storage
- Storage local to CPU
- Move computing to data

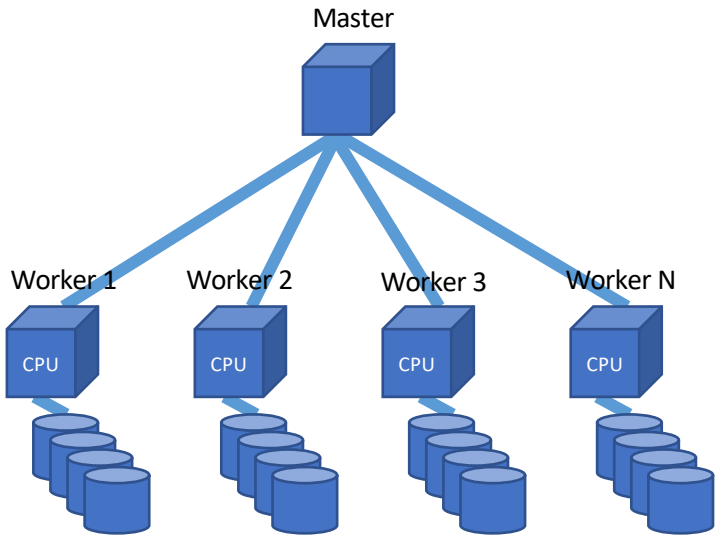


HPC vs Big Data cluster

HPC cluster



Big Data Cluster



Apache Hadoop

- Java implementation of MapReduce by Yahoo
- Open Source project since 2006
- Google File System -> Hadoop File System (HDFS)
- YARN "Yet Another Resource Negotiator” – Resource manager
- Other components on top:
  - Apache Pig – high level language
  - Apache Hive – SQL-like query syntax
  - Apache Hbase – NoSQL database using HDFS, based on Google Big Table concept

Google Big Table

- Original paper in 2006
- Internally used at Google
- Based on GFS
- HBase – open source implementation in Hadoop
- Now available in Google cloud
- Compatible with HBase API
- https://ai.google/research/pubs/pub27898

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

Abstract

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this paper we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

1 Introduction

Over the last two and a half years we have designed, implemented, and deployed a distributed storage system for managing structured data at Google called Bigtable. Bigtable is designed to reliably scale to petabytes of data and thousands of machines. Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability. Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth. These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

In many ways, Bigtable resembles a database: it shares many implementation strategies with databases. Parallel databases [14] and main-memory databases [13] have achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

Section 2 describes the data model in more detail, and Section 3 provides an overview of the client API. Section 4 briefly describes the underlying Google infrastructure on which Bigtable depends. Section 5 describes the fundamentals of the Bigtable implementation, and Section 6 describes some of the refinements that we made to improve Bigtable's performance. Section 7 provides measurements of Bigtable's performance. We describe several examples of how Bigtable is used at Google in Section 8, and discuss some lessons we learned in designing and supporting Bigtable in Section 9. Finally, Section 10 describes related work, and Section 11 presents our conclusions.

2 Data Model

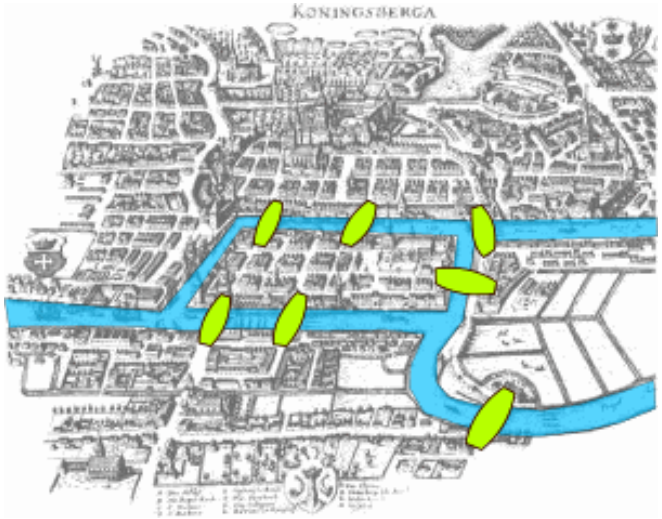
A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

(row:string, column:string, time:int64) → string

To appear in OSDI 2006

Pregel – graph processing

- Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N. and Czajkowski, G., 2010, June. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (pp. 135-146). ACM.
- Pregel (Pregoła): Euler 1736



Pregel – Bulk Synchronous Processing (BSP)

- Similar to SIMD – single operation on multiple data items (here: graph nodes)
- Example: PageRank
  - PR(A) = (1-d) + d (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))
  - d = 0.85
  - C() – neighbors count
- PageRank Algorithm:
  1. PR=1
  2. At each step: p sends PR(p)/C(p) to neighbors
  3. New value: PR = 0.15 + 0.85 * Sum(contribs)

Scalability! But at what COST?

- Configuration that Outperforms a Single Thread
- Many systems have high COST of 100s of cores

Frank McSherry; Michael Isard; Derek G. Murray, Usenix 2015

Scalability! But at what COST?

Frank McSherry Unaffiliated
Michael Isard Unaffiliated*
Derek G. Murray Unaffiliated†

Abstract

We offer a new metric for big data platforms, COST, or the Configuration that Outperforms a Single Thread. The COST of a given platform for a given problem is the hardware configuration required before the platform outperforms a competent single-threaded implementation. COST weighs a system's scalability against the overheads introduced by the system, and indicates the actual performance gains of the system, without rewarding systems that bring substantial but parallelizable overheads.

We survey measurements of data-parallel systems recently reported in SOSP and OSDI, and find that many systems have either a surprisingly large COST, often hundreds of cores, or simply underperform one thread for all of their reported configurations.

1 Introduction

> "You can have a second computer once you've shown you know how to use the first one."
>
> –Paul Barham

The published work on big data systems has fetishized scalability as the most important feature of a distributed data processing platform. While nearly all such publications detail their system's impressive scalability, few directly evaluate their absolute performance against reasonable benchmarks. To what degree are these systems truly improving performance, as opposed to parallelizing overheads that they themselves introduce?

Contrary to the common wisdom that effective scaling is evidence of solid systems building, any system can scale arbitrarily well with a sufficient *lack* of care in its implementation. The two scaling curves in Figure 1 present the scaling of a Naiad computation before (system A) and after (system B) a performance optimization is applied. The optimization, which removes parallelizable overheads, damages the apparent scalability despite resulting in improved performance in all configurations.

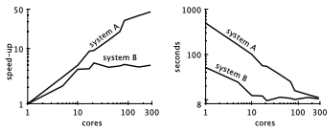


Figure 1: Scaling and performance measurements for a data-parallel algorithm, before (system A) and after (system B) a simple performance optimization. The unoptimized implementation "scales" far better, despite (or rather, because of) its poor performance.

While this may appear to be a contrived example, we will argue that many published big data systems more closely resemble system A than they resemble system B.

1.1 Methodology

In this paper we take several recent graph processing papers from the systems literature and compare their reported performance against simple, single-threaded implementations on the same datasets using a high-end 2014 laptop. Perhaps surprisingly, many published systems have *unbounded* COST—i.e., no configuration outperforms the best single-threaded implementation—for all of the problems to which they have been applied.

The comparisons are neither perfect nor always fair, but the conclusions are sufficiently dramatic that some concern must be raised. In some cases the single-threaded implementations are more than an order of magnitude faster than published results for systems using hundreds of cores. We identify reasons for these gaps: some are intrinsic to the domain, some are entirely avoidable, and others are good subjects for further research.

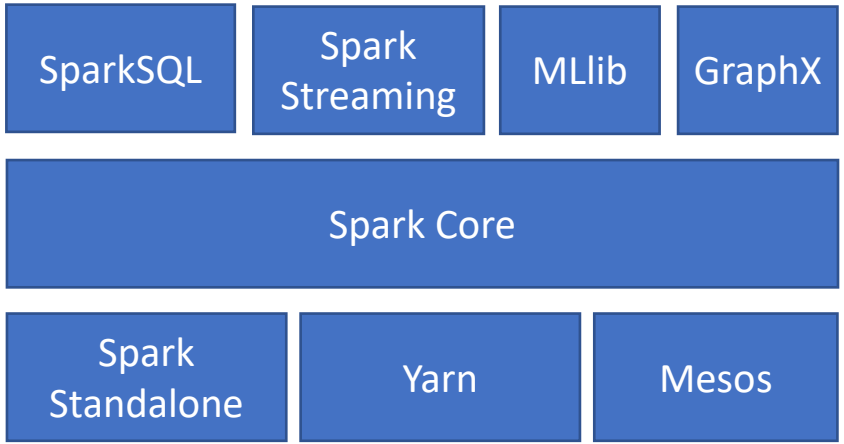
We stress that these problems lie not necessarily with the systems themselves, which may be improved with time, but rather with the measurements that the authors provide and the standard that reviewers and readers demand. Our hope is to shed light on this issue so that future research is directed toward distributed systems whose scalability comes from advances in system design rather than poor baselines and low expectations.

*Michael Isard was employed by Microsoft Research at the time of his involvement, but is now unaffiliated.

†Derek G. Murray was unaffiliated at the time of his involvement, but is now employed by Google Inc.

Apache Spark

Apache Spark



- Open Source project since 2014
- Started at University of California Berkeley
- In-memory processing
- Resilient Distributed Dataset (RDD) abstraction
- SparkSQL – high-level programming API
- Spark Streaming – stream processing on RDD
- MLlib – machine learning library
- GraphX – distributed graph-processing framework based on Pregel

Spark paper

- Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S. and Stoica, I., 2012, April. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (pp. 2-2). USENIX Association.

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, in-

tion, which can dominate application execution times.

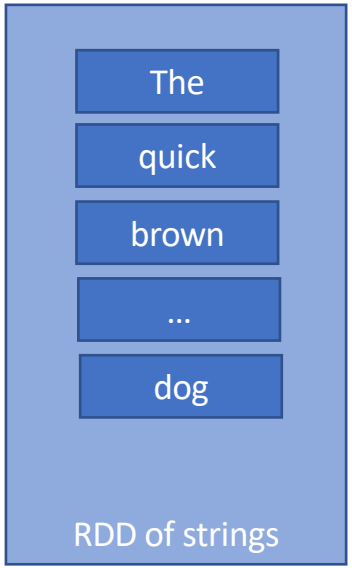
Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.,* looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.,* to let a user load several datasets into memory and run ad-hoc queries across them.

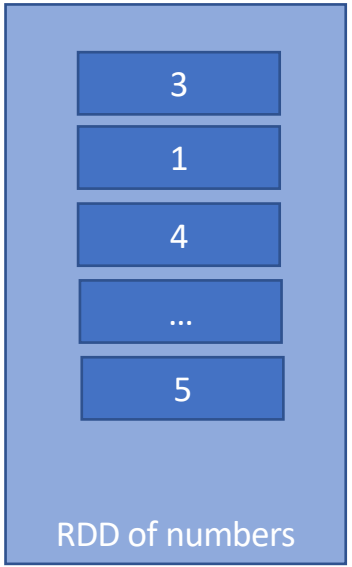
In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are

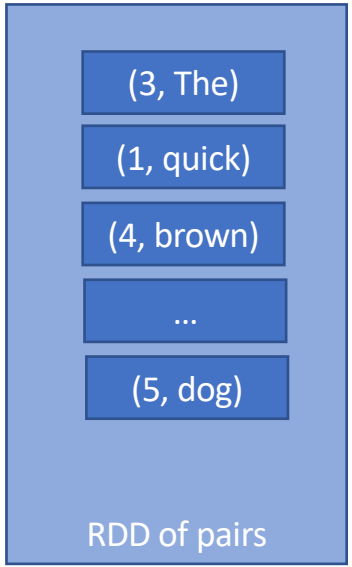
RDD Abstraction

- RDD: read-only, partitioned collection of records
- RDDs can only be created through deterministic operations on
  - data in stable storage, or
  - other RDDs
- Operations = transformations. Examples:
  - map,
  - filter,
  - join
- Do not need to be materialized at all times.
- RDD has enough information about how it was derived from other datasets (its lineage „rodowód”) to compute its partitions from data in stable storage.
- Powerful property: a program cannot reference an RDD that it cannot reconstruct after a failure
- Users can control two other aspects of RDDs:
  - persistence which RDDs to reuse, storage strategy for them (e.g., in-memory storage
  - Partitioning across machines based on a key in each record
- Lazy evaluated

RDD Examples

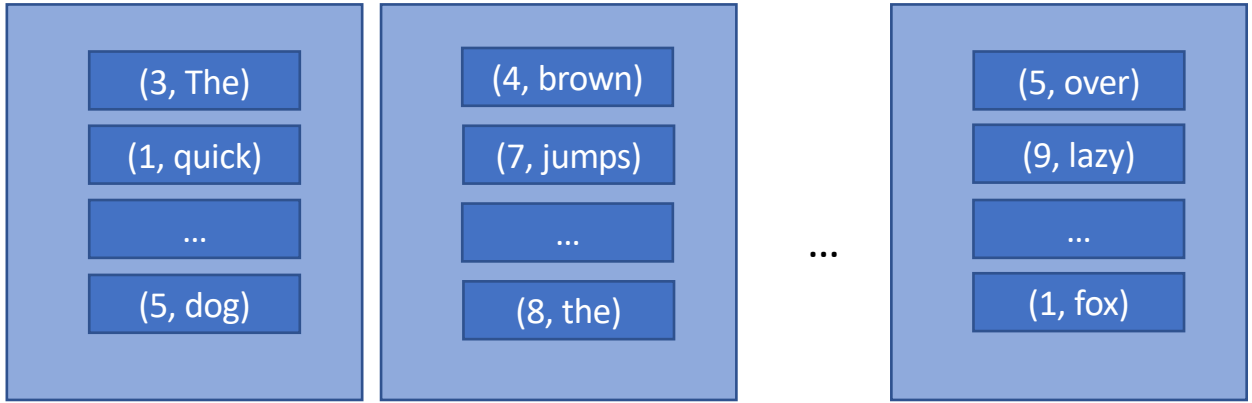






- https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-rdd.html

Distribution and partitioning



- No replication!
- In the case of failure, RDD can be
  - reloaded from disk, or
  - recomputed from other RDD

RDD Transformations Example

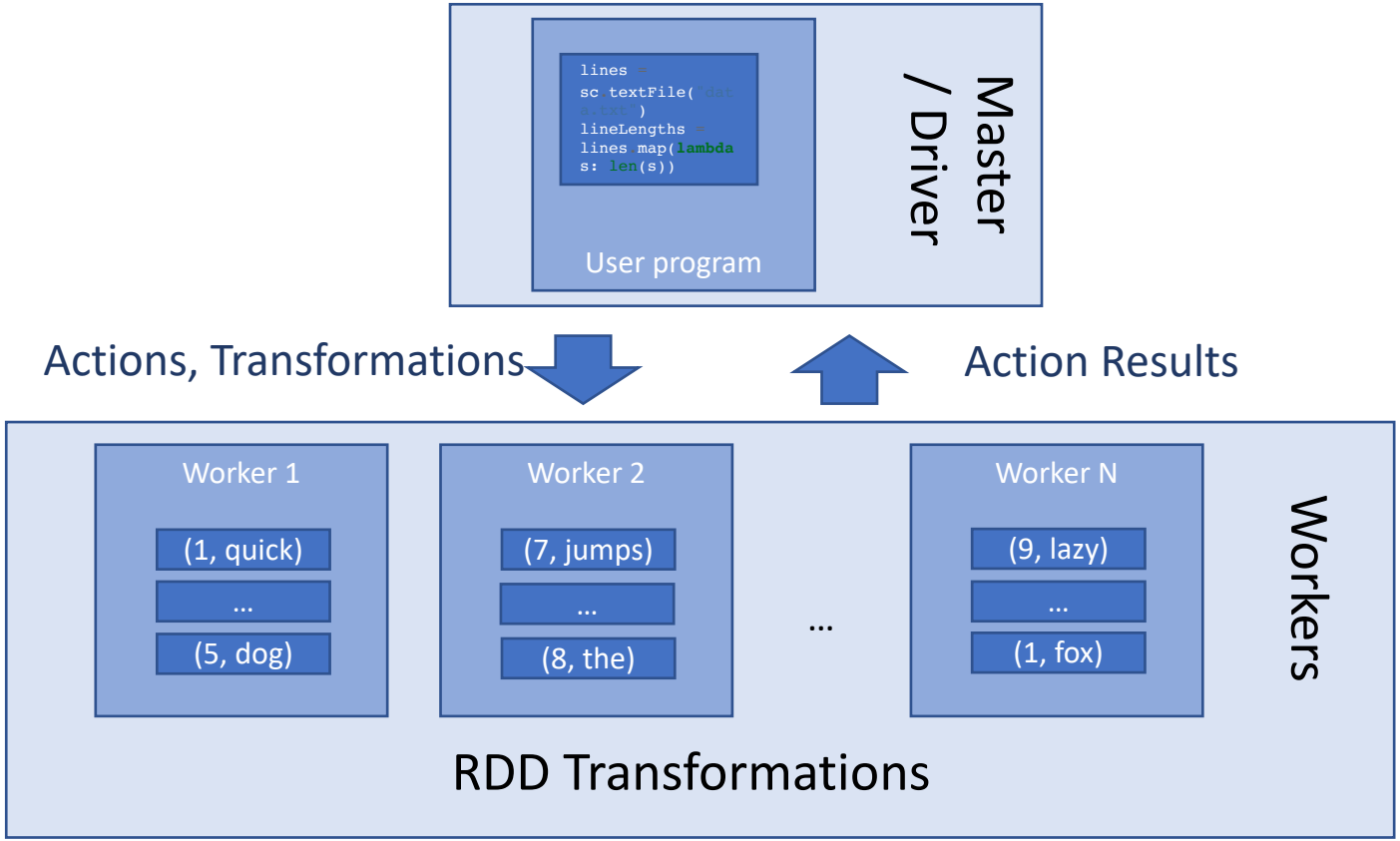
- Transformations:
  - create a new dataset from an existing one
  - lazy evaluation

```
lines = sc.textFile("data.txt")
lineLengths = lines.map(lambda s: len(s))
```

- Actions
  - run a computation on the dataset
  - return a value to the driver program

```
totalLength = lineLengths.reduce(lambda a, b: a + b)
```

Spark architecture



Datasets and DataFrames

Dataset

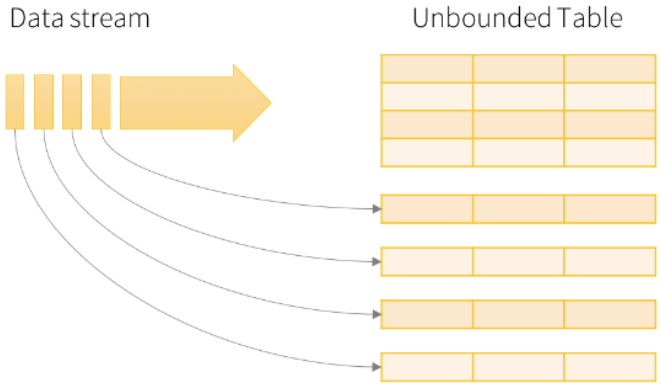
- distributed collection of data, added in Spark 1.6
- benefits of RDDs (strong typing, ability to use powerful lambda functions)
- benefits of Spark SQL's optimized execution engine.
- can be constructed from JVM objects
- manipulated using functional transformations (map, flatMap, filter, etc.).
- Dataset API is available in Scala and Java.
- Python does not have the support for the Dataset API.

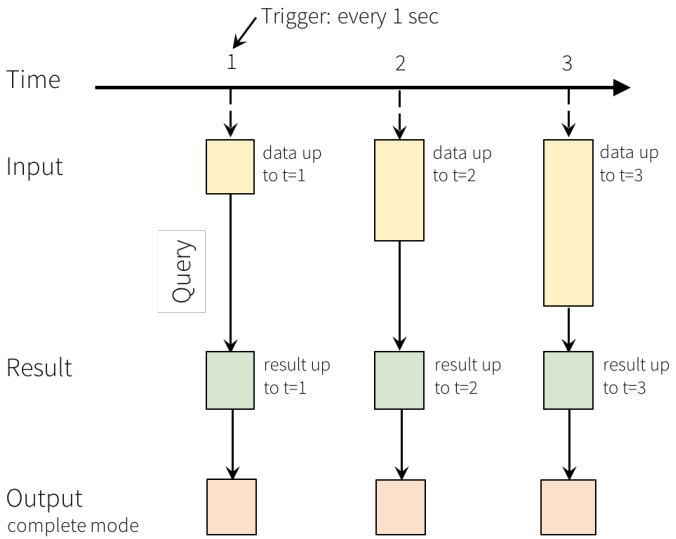
DataFrame

- Dataset organized into named columns.
- conceptually equivalent to a table in a relational database or a data frame in R/Python,
- DataFrames can be constructed from:
  - structured data files,
  - tables in Hive,
  - external databases,
  - existing RDDs.
- The DataFrame API is available in
  - Scala,
  - Java,
  - Python,
  - R

Spark Streaming

- API on top of RDD
- Structured stream
- Query on a stream
- Combining of streaming data with static datasets and interactive queries
- Native integration with advanced processing libraries (SQL, machine learning, graph processing)





Programming Model for Structured Streaming

Spark Machine Learning Library – MLlib

- ML Algorithms
  - classification, regression, clustering, and collaborative filtering
- Featurization
  - feature extraction, transformation, dimensionality reduction, and selection
- Pipelines
  - Transformers: DF to DF
  - Estimator: DF to Transformer (fit data to produce a trained model)
- Persistence
  - saving and load algorithms, models, and Pipelines
- Utilities
  - linear algebra, statistics, data handling, etc.

Spark Alternatives

- H2O.ai – in memory, distributed machine learning, export models as POJO (enterprise integration)
- Storm – stream processing, on-line computing, real-time analytics
- Flink – stream processing

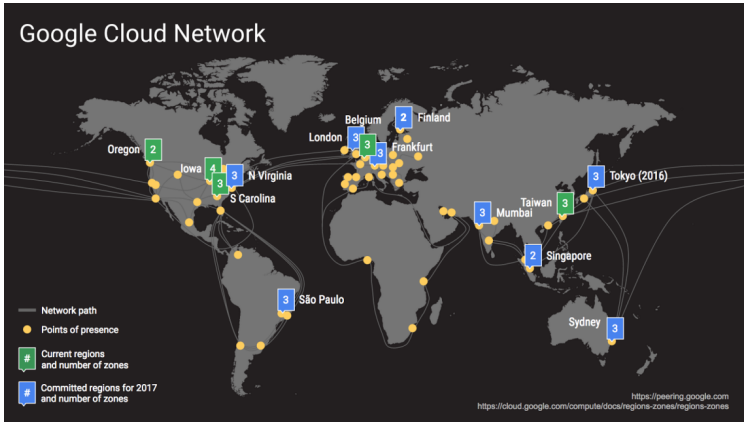
Cloud For Big Data

Infrastructure – from clusters to clouds

- Traditional HPC clusters in computing centers
  - Job scheduling systems
    - #jobs > #resources
    - Queueing systems
  - Local (parallel) high performance storage
  - Free for researchers
    - Allocation based on computing grants

Infrastructure – from clusters to clouds

- Infrastructure as a service
  - Amazon AWS
  - Google Cloud
  - Microsoft Azure
- Globally distributed
- Virtual machines (VMs)
- On-demand
- Cost in $$ per time unit
  - 2008: $0.1 per hour
  - 2018: per second!



Cloud computing characteristics

- Elasticity
  - Capability to get a lot of resources for a certain period of time quickly
- Scalability
  - Capability to meet growing client’s demands for resources
  - Enabled by elasticity but it’s not the same!
- Pay-per-use model

Cloud Elasticity: NYT example

- New York Times provides PDF versions of articles
- Previously generated each time on-demand
  - Large load on NYT servers
- Decided to convert 11M archive articles once and store as PDFs
  - Used 100 instances of Amazon EC2
  - Conversion took 24 hours

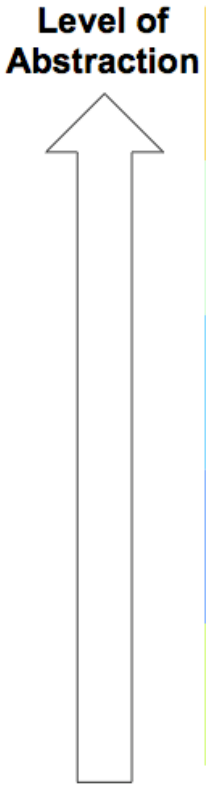
Clouds as alternative hosting model

- Instead of paying (annually) for servers in hosting companies …
- … or buying and maintaining servers yourself …
- … rent cycles in the cloud!
- Advantages:
  - Elastic, scalable, reliable infrastructure / platform
  - No need to maintain large computing infrastructure

SaaS, PaaS, IaaS

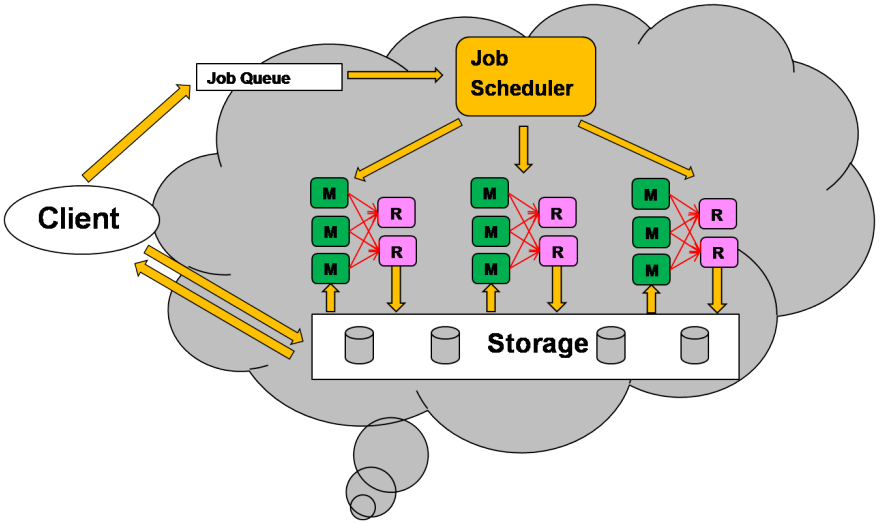
Cloud Market Types	Types of Offerings	Examples
Software-as-a-Service	• Rich Internet application web sites • Application as Web Sites • Collaboration and email • Office Productivity • Client apps that connect to services in the cloud	• Flikr • Myspace.com • Cisco WebEx office • Gmail • IBM Bluehouse
App-components -as-a-Service	• APIs for specific service access for integration • Web-based software service than can combine to create new services, as in a mashup	• Amazon Flexible Payments Service and DevPay • Salesforce.com's AppExchange • Yahoo! Maps API • Google Calendar API • zembly
Platform-as-a-Service	• Development-platform-as-a-service • Database • Message Queue • App Servicer • Blob or object data stores	• Google App Engine and BigTable • Microsoft SQL Server Data Services • Engine Yard • Salesforce.com's Force.com
Infrastructure-as-a-Service	• Virtual servers • Logical disks • VLAN networks • Systems Management	• Akamai • Amazon EC2 • CohesiveFT • Mosso (from Rackspace) • Joyent Accelerators • Nirvanix Storage Delivery Network
Physical Infrastructure	• Managed Hosting • Collocation • Internet Service Provider • Unmanaged hosting	• GoDaddy.com • Rackspace • Savvis

Adapted from Forrester Research Taxonomy



Clouds for batch processing

- Batch processing: statistics computation, report generation, machine learning, analytics ...
- batch mode - economical to use the "pay as you go" model
- high tolerance in latency, independent tasks
- Map-reduce on the cloud



Clouds for storage

- Amazon S3 – Simple Storage Service
- Amazon Relational Database Service (RDS) – access to mysql databases
- NoSQL storage
  - GoogleBigTable, HBase, Hypertable
  - AmazonDynamo, Voldemort, Cassendra, Riak
  - Redis
  - CouchDB, MongoDB

Amazon Web Services

- Elastice Compute Cloud (EC2) – computing
- Simple Storage Service (S3) – storage
- SimpleDB – simple database
- Simple Queuing Service (SQS) – scalable, reliable message queue
- Others … http://aws.amazon.com

Amazon EC2 pricing over 45 types of instances

	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
General Purpose - Current Generation					
t2.nano	1	Variable	0.5	EBS Only	$0.0065 per Hour
t2.micro	1	Variable	1	EBS Only	$0.013 per Hour
t2.small					
t2.medium					
t2.large					
m4.large					
m4.xlarge					
m4.2xlarge					
m4.4xlarge					
m4.10xlarg					
m4.16xlarg					

Compute Optimized - Current Generation					
c4.large	2	8	3.75	EBS Only	$0.105 per Hour
c4.xlarge	4	16	7.5	EBS Only	$0.209 per Hour
c4.2xlarge					
c4.4xlarge					
c4.8xlarge					
c3.large					
c3.xlarge					
c3.2xlarge					
c3.4xlarge					
c3.8xlarge					

Storage Optimized - Current Generation					
i2.xlarge	4	14	30.5	1 x 800 SSD	$0.853 per Hour
i2.2xlarge	8	27	61	2 x 800 SSD	$1.705 per Hour
i2.4xlarge	16	53	122	4 x 800 SSD	$3.41 per Hour
i2.8xlarge	32	104	244	8 x 800 SSD	$6.82 per Hour
d2.xlarge	4	14	30.5	3 x 2000 HDD	$0.69 per Hour
d2.2xlarge	8	28	61	6 x 2000 HDD	1.38 per Hour
d2.4xlarge	16	56	122	12 x 2000 HDD	$2 per Hour
d2.8xlarge	36	116	244	24 x 2000 HDD	$5.52 per Hour

850 x (!!!)

Amazon EC2 quick start

EC2 Quick start

1. Create account:
   - Get student credits from AWS Educate: https://aws.amazon.com/education/awseducate
2. Log-in to AWS Console:
   - https://eu-west-1.console.aws.amazon.com/ec2/v2/home?region=eu-west-1#

AWS Console -> EC2

https://eu-west-1.console.aws.amazon.com/ec2/v2/home?region=eu-west-1#

aws | Services | Resource Groups | Maciej Malawski | Ireland | Support

- EC2 Dashboard
- Events
- Tags
- Reports
- Limits

INSTANCES
- Instances
- Launch Templates
- Spot Requests
- Reserved Instances
- Dedicated Hosts
- Scheduled Instances
- Capacity Reservations

IMAGES
- AMIs
- Bundle Tasks

Resources

You are using the following Amazon EC2 resources in the EU West (Ireland) region:

2 Running Instances	1 Elastic IPs
0 Dedicated Hosts	24 Snapshots
2 Volumes	0 Load Balancers
601 Key Pairs	36 Security Groups
0 Placement Groups	

Create Instance

To start using Amazon EC2 you will want to launch a virtual server, known as an Amazon EC2 instance.

Launch Instance

Note: Your instances will launch in the EU West (Ireland) region

Service Health

Scheduled Events

Account Attributes

Supported Platforms
- EC2
- VPC

Resource ID length management

Console experiments

Additional Information

- Getting Started Guide
- Documentation
- All EC2 Resources
- Forums
- Pricing
- Contact Us

Launch instance

https://eu-west-1.console.aws.amazon.com/ec2/v2/home?region=eu-west-1#LaunchInstanceWizard:

aws Services Resource Groups Maciej Malawski Ireland Support

1. Choose AMI | 2. Choose Instance Type | 3. Configure Instance | 4. Add Storage | 5. Add Tags | 6. Configure Security Group | 7. Review

Step 1: Choose an Amazon Machine Image (AMI)

Cancel and Exit

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. You can select an AMI provided by AWS, our user community, or the AWS Marketplace; or you can select one of your own AMIs.

ami-0d80f9df685ad98f9

Quick Start (0)

My AMIs (1)

AWS Marketplace (3007)

Community AMIs (1)

1 to 1 of 1 AMIs

spark-data-science-v1 - ami-0d80f9df685ad98f9

Select

Root device type: ebs Virtualization type: hvm Owner: 524374011118 ENA Enabled: Yes 64-bit (x86)

Instance type: t2.micro

aws Services Resource Groups Maciej Malawski Ireland Support

1. Choose AMI | 2. Choose Instance Type | 3. Configure Instance | 4. Add Storage | 5. Add Tags | 6. Configure Security Group | 7. Review

Step 2: Choose an Instance Type

	Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance	IPv6 Support
☐	General purpose	t2.nano	1	0.5	EBS only	-	Low to Moderate	Yes
☑	General purpose	t2.micro Free tier eligible	1	1	EBS only	-	Low to Moderate	Yes
☐	General purpose	t2.small	1	2	EBS only	-	Low to Moderate	Yes
☐	General purpose	t2.medium	2	4	EBS only	-	Low to Moderate	Yes
☐	General purpose	t2.large	2	8	EBS only	-	Low to Moderate	Yes
☐	General purpose	t2.xlarge	4	16	EBS only	-	Moderate	Yes

Cancel | Previous | Review and Launch | Next: Configure Instance Details

Make sure to open port 80 (HTTP)

aws Services Resource Groups Maciej Malawski Ireland Support

1. Choose AMI 2. Choose Instance Type 3. Configure Instance 4. Add Storage 5. Add Tags 6. Configure Security Group 7. Review

Step 6: Configure Security Group

A security group is a set of firewall rules that control the traffic for your instance. On this page, you can add rules to allow specific traffic to reach your instance. For example, if you want to set up a web server and allow Internet traffic to reach your instance, add rules that allow unrestricted access to the HTTP and HTTPS ports. You can create a new security group or select from an existing one below. Learn more about Amazon EC2 security groups.

Assign a security group: Create a **new** security group

Select an **existing** security group

Security group name: launch-wizard-17

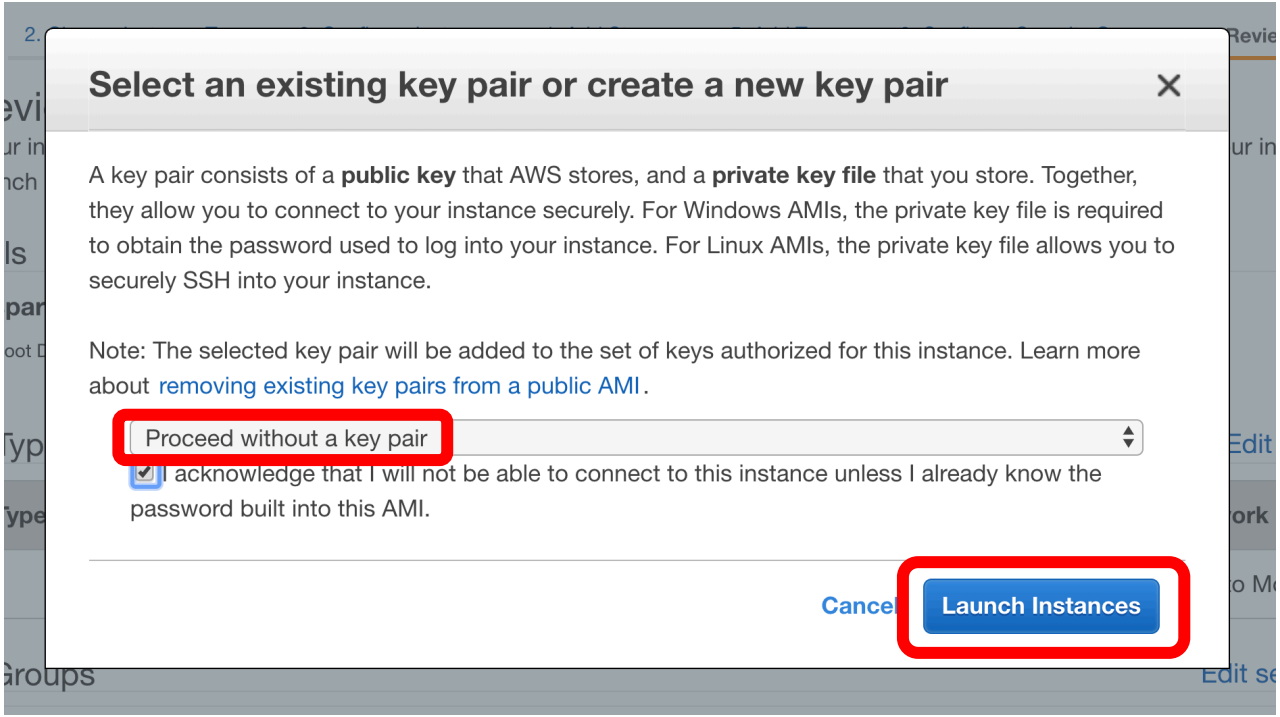
Description: launch-wizard-17 created 2018-12-08T15:22:14.279+01:00

Type	Protocol	Port Range	Source	Description
HTTP	TCP	80	Custom 0.0.0.0/0	e.g. SSH for Admin Deskto

Add Rule

Cancel Previous Review and Launch

Proceed without a key pair



Copy Public IP Address

Launch Instance | Connect | Actions

search : i-02b30c963f756493d | Add filter | 1 to 1 of 1

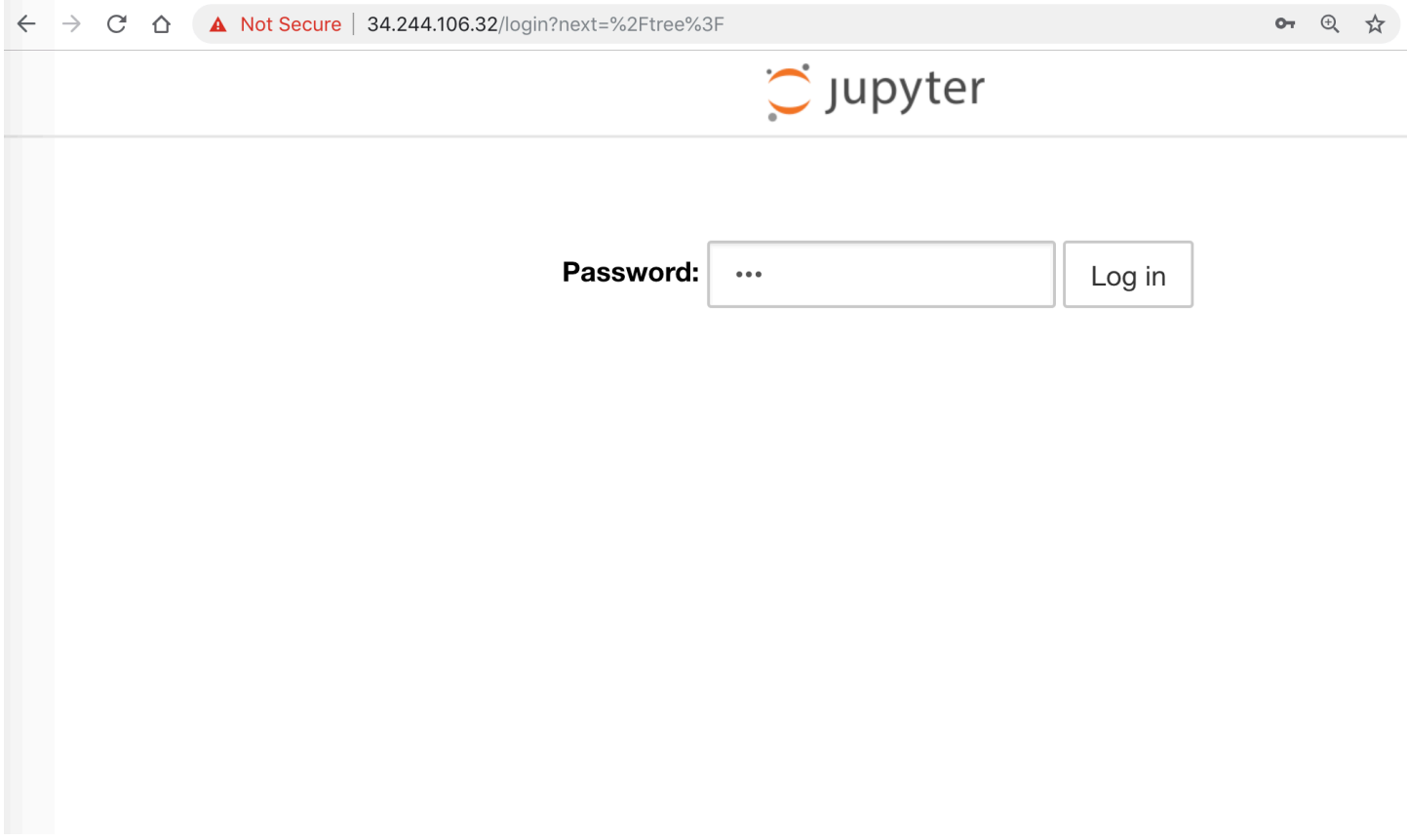
Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checl
	i-02b30c963f756493d	t2.micro	eu-west-1a	running	Initializir

Instance: i-02b30c963f756493d Public IP: 34.244.106.32

Description | Status Checks | Monitoring | Tags

Instance ID	i-02b30c963f756493d	Public DNS (IPv4)	-
Instance state	running	IPv4 Public IP	34.244.106.32
Instance type	t2.micro	IPv6 IPs	-
Elastic IPs		Private DNS	ip-172-30-0-150.eu-west-

Log in to Jupyter. Password: agh



Bonus: Spark UI at port 4040

Not Secure | 52.214.254.80:4040/jobs/

Spark 2.4.0 | Jobs | Stages | Storage | Environment | Executors | SQL

Spark Jobs (?)

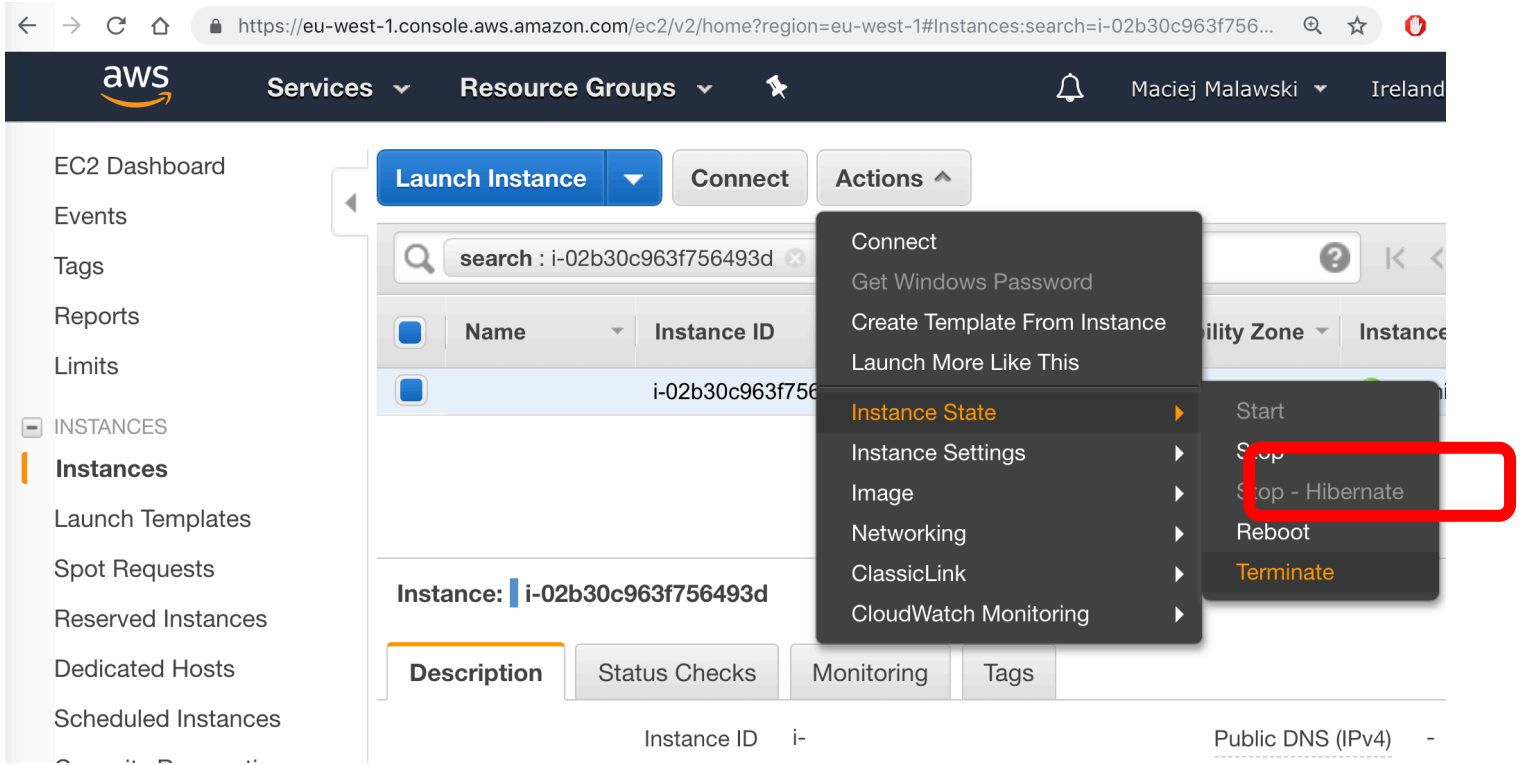
User: ubuntu
Total Uptime: 1.7 min
Scheduling Mode: FIFO
Completed Jobs: 23

Event Timeline

Completed Jobs (23)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stag
22	toPandas at <ipython-input-49-d9239ee24c79>:1 toPandas at <ipython-input-49-d9239ee24c79>:1	2018/12/08 20:03:51	63 ms	1/1	
21	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2018/12/08 20:03:50	0.3 s	1/1	
20	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2018/12/08 20:03:50	0.1 s	1/1	
19	toPandas at <ipython-input-38-0a74a532b560>:1 toPandas at <ipython-input-38-0a74a532b560>:1	2018/12/08 20:03:47	1 s	2/2	

Terminate instance when not using!



Recent Example from CERN

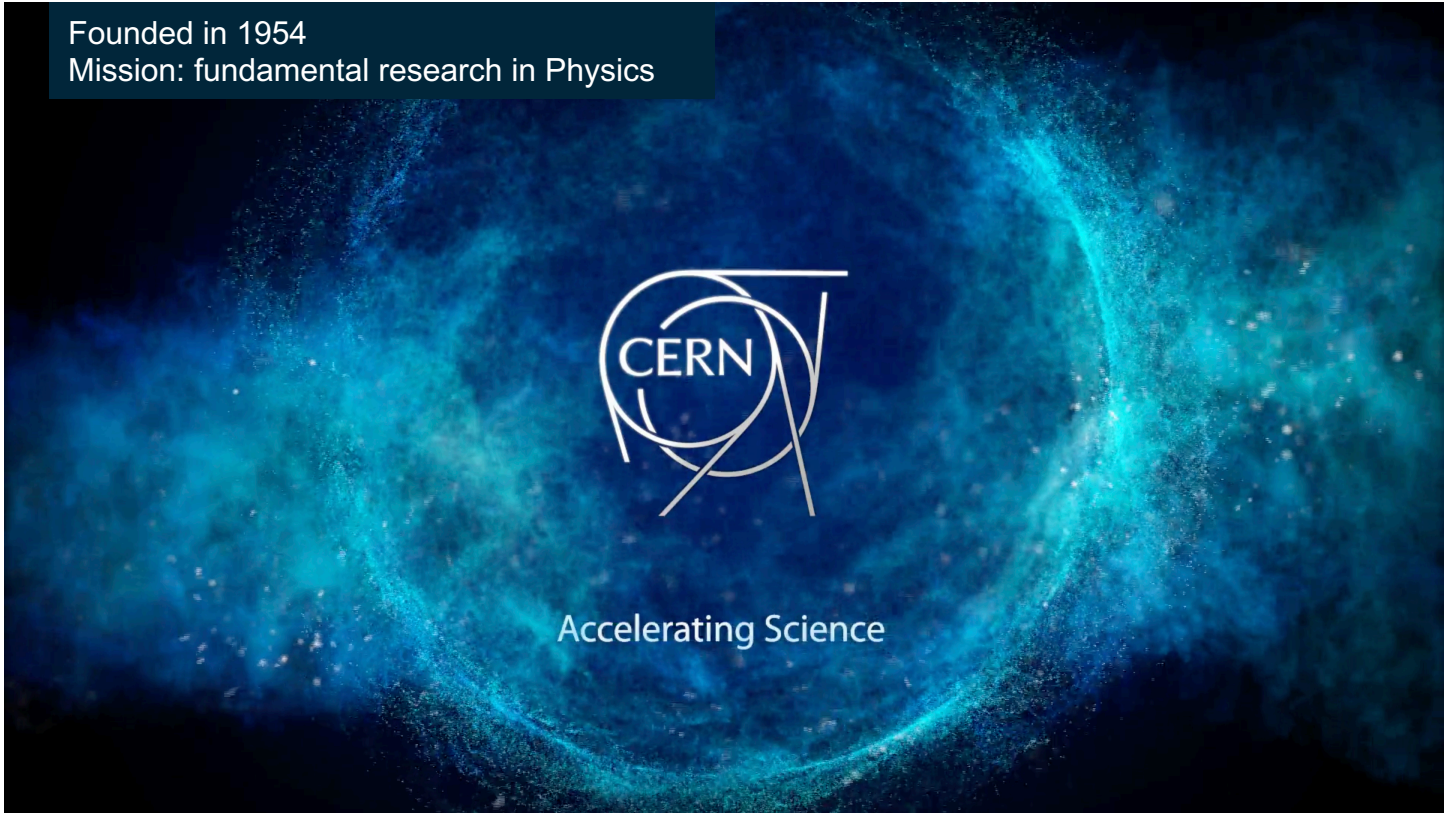
Declarative Big Data Analysis for High-Energy Physics: TOTEM Use Case

Valentina Avati[†], Milosz Blaszkiewicz[†], Enrico Bocchi[*], Luca Canali[*], Diogo Castro[*], Javier Cervantes[*], Leszek Grzanka[†], Enrico Guiraud[*], Jan Kaspar[*], Prasanth Kothuri[*], Massimo Lamanna[*], Maciej Malawski[†], Aleksandra Mnich[†], Jakub Moscicki[*], Shravan Murali[*], Danilo Piparo[*], Enric Tejedor[*]

[†]AGH Universtity of Science and Technology, Krakow, Poland, Email: {grzanka,malawski}@agh.edu.pl

[*] CERN, CH-1211 Geneva 23, Switzerland, Email: {first.last}@cern.ch

Founded in 1954
Mission: fundamental research in Physics



Large Hadron Collider, CERN, Geneva

http://cdn.zmescience.com/wp-content/uploads/2015/05/cern-lhc-aerial.jpg

27 km circumference
6.5 TeV per proton

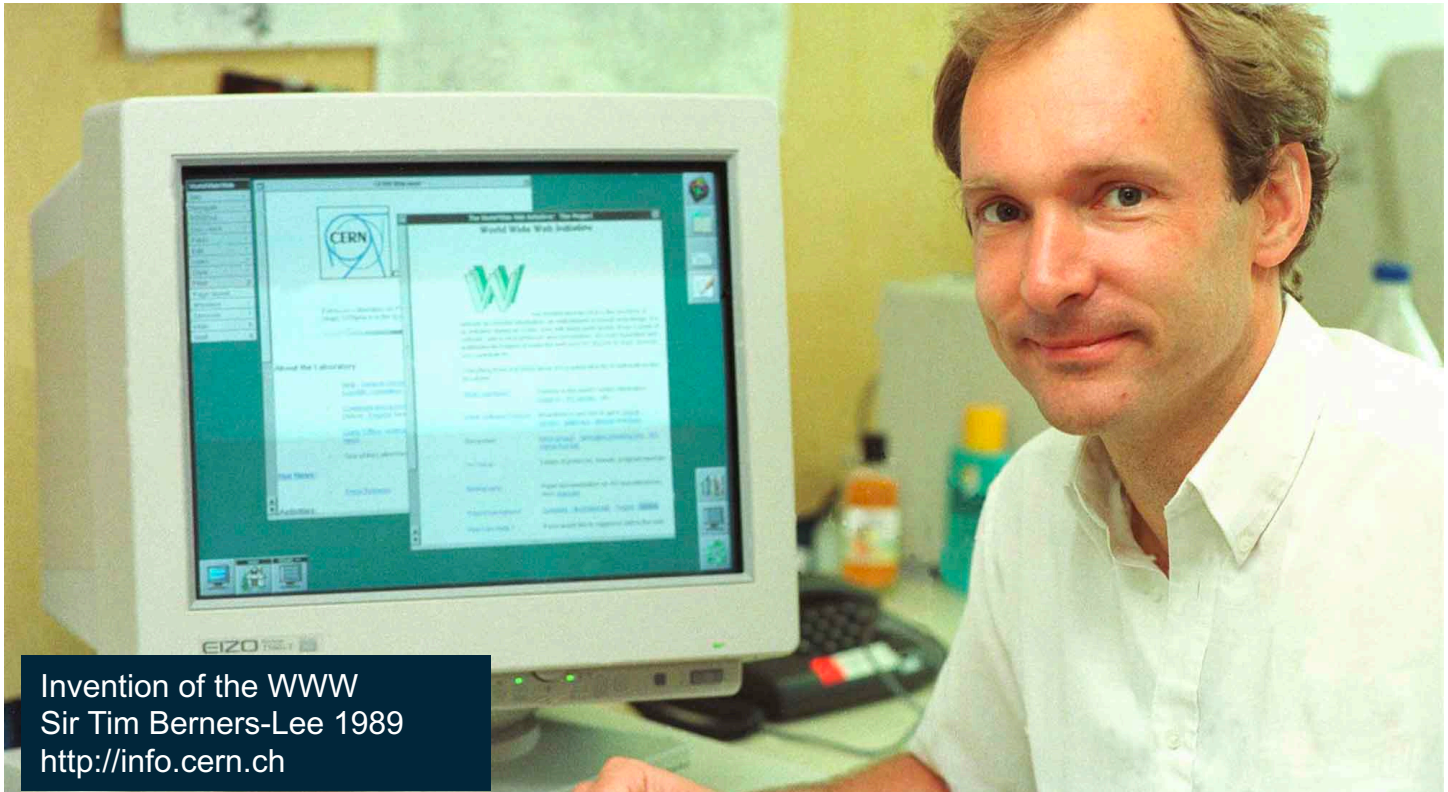
The largest scientific experiment to date

600 million events per second

1 PB / second of raw physics data

https://www.bnl.gov/magnets/images/LHC/dual-aperture-magnet-720px.jpg

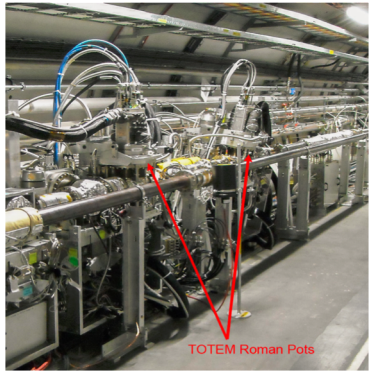
Discovery of the Higgs boson
ATLAS & CMS 2012

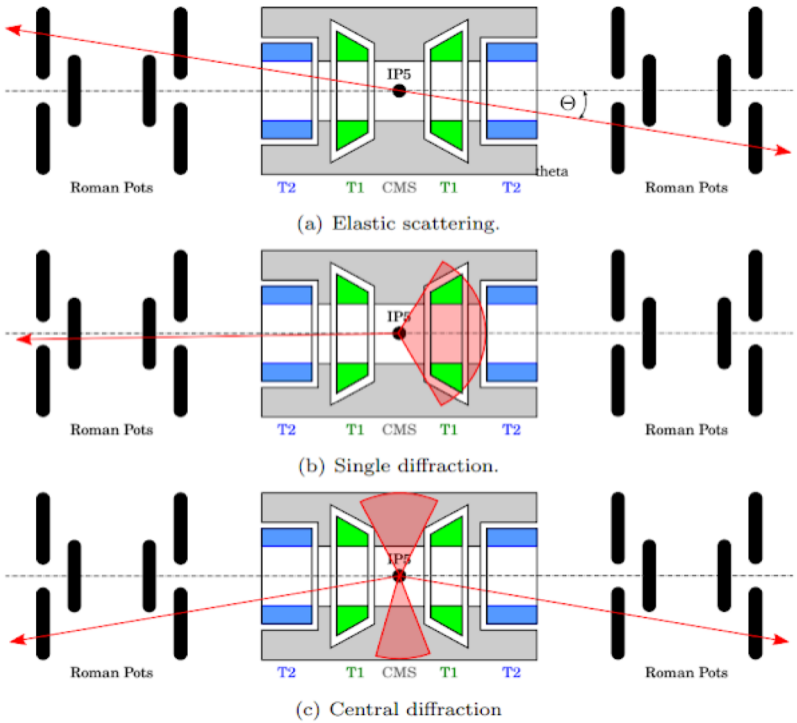


Invention of the WWW
Sir Tim Berners-Lee 1989
http://info.cern.ch

Less known experiment - TOTEM

Detectors located ~200m from collision point, approaching beam to ~mm.
Measurements of scattered protons.





Both protons survive collision
Scattering angle ~ few mrad
no signal in CMS detector

One proton survive collision
Signal in CMS and forward detector

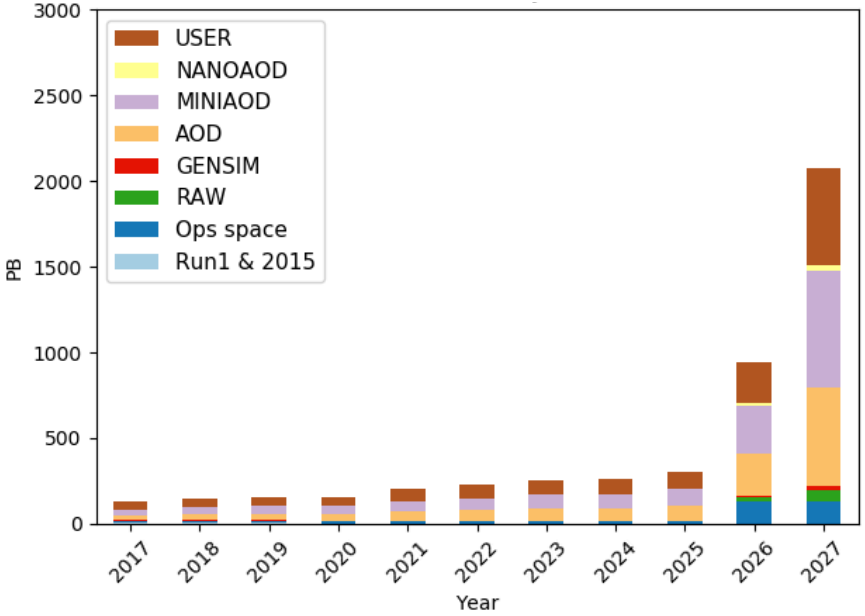
Both protons survive collision
signal in CMS and forward detector

Motivation: projected increase of data volume in LHC

- HL-LHC needs 10x faster software
- 30 times more data
- Produced by ATLAS / CMS

  → Need for new better tools

CMS estimated disk space required into the HL-LHC era by tier

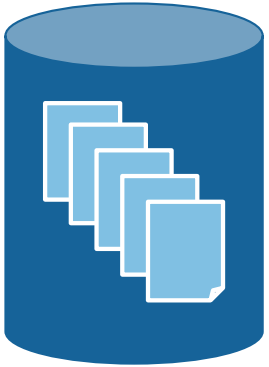


https://twiki.cern.ch/twiki/bin/view/CMSPublic/CMSOfflineComputingResults

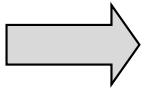
ROOT – current programming model

https://root.cern/

- The data analysis framework for high-energy physics (HEP)
- Data processing, statistical analysis, visualization and storage
- ~1 EB stored in ROOT format
  - Binary
  - Compressed
  - Columnar
- Standard usage: a loop over the events (C++)



ROOT dataset stored in one or multiple files

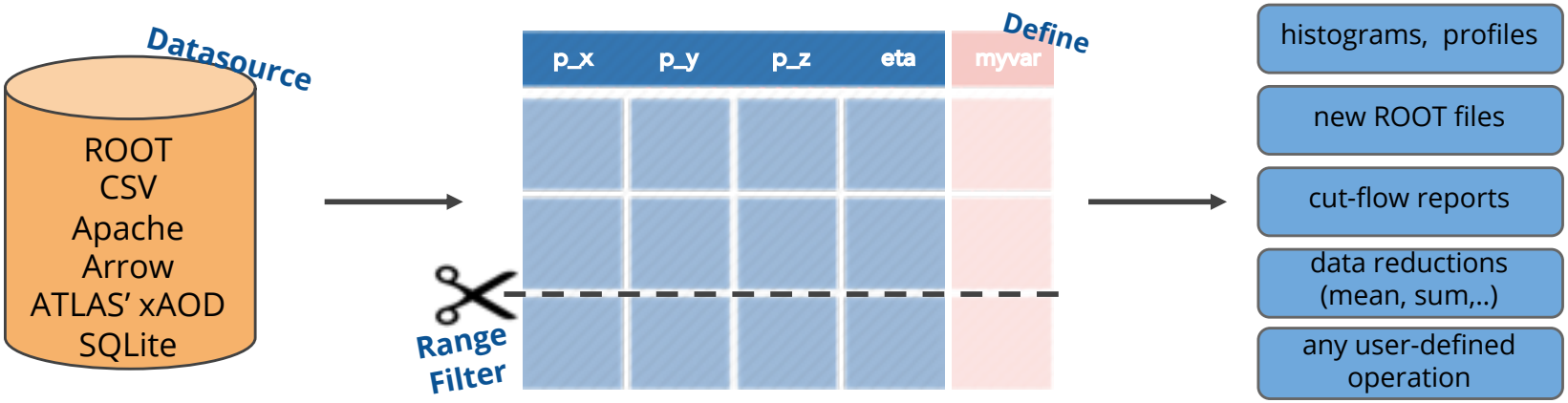


Column: **physics entity**

px	py	pz	eta

Row: **collision event**

RDataFrame - New ROOT Interface

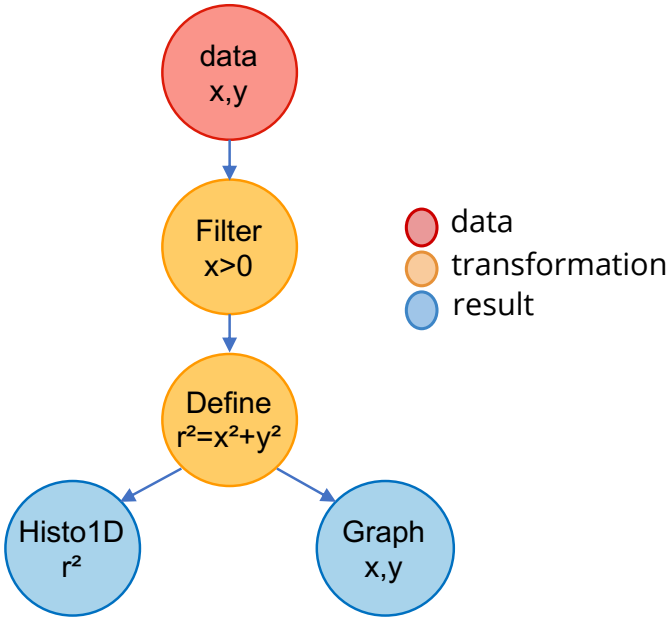


- Transparent low-level optimizations: multi-threading, lazy I/O, lazy evaluation
- Full support for and consistent interfaces in both Python and C++

RDataFrame – declarative programming model

- RDataFrame
  - High level interfaces
  - Implemented in C++, interfaced also to Python
  - Tailored for ROOT and HEP

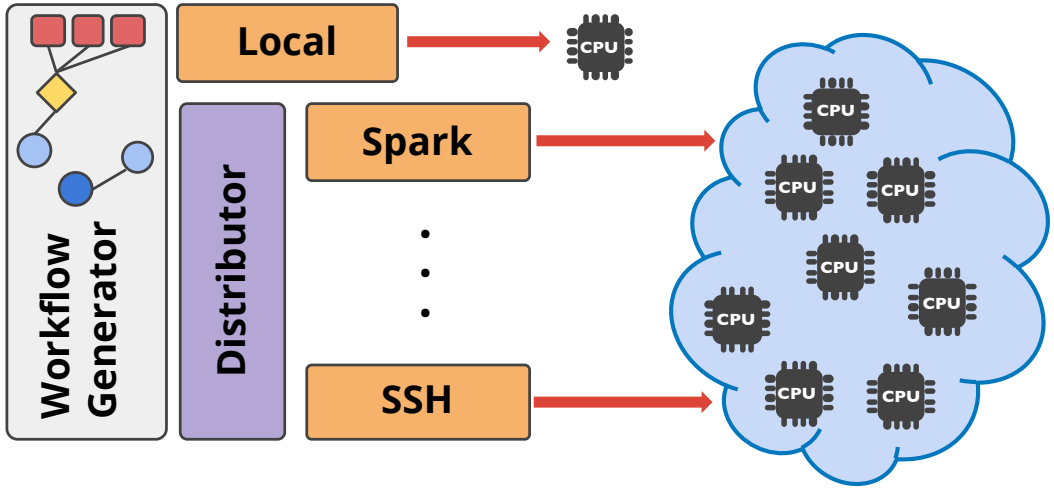
```
import ROOT

df = ROOT.ROOT.RDataFrame(dataset)

df2 = df.Filter('x > 0')
        .Define('r2', 'x*x + y*y')

h = df2.Histo1D('r2')
g = df2.Graph('x', 'y')
```



Implicit parallelization

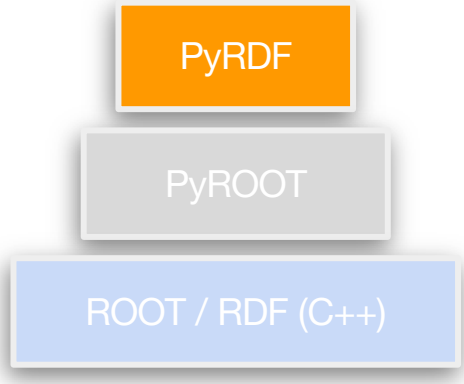
- Two levels
  - Multicore (on single node)
  - Distributed (using Spark)

```
d = RDataFrame("t", dataset)
f = d.Define(...)
     .Define(...)
     .Filter(...)

h1 = f.Histo1D(...)
h2 = f.Histo1D(...)
h3 = f.Histo1D(...)
```



High-level programming model

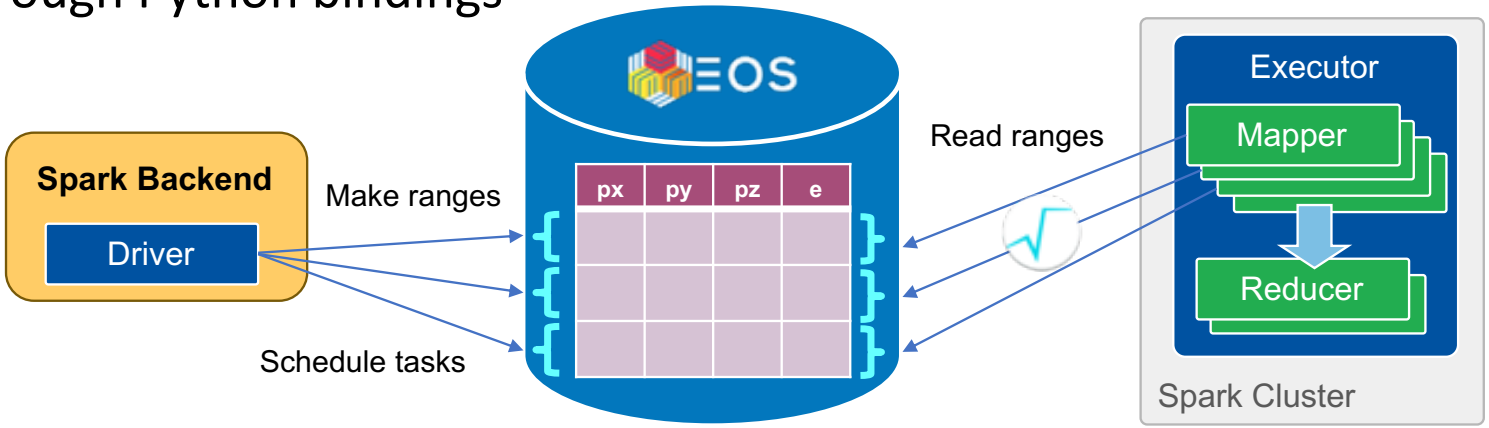
- Adding distribution with minimal changes to the code




```
# Select Spark backend
PyRDF.use("spark")

# Initialize RDataFrame object
df = PyRDF.RDataFrame(dataset)

# Define operations
df2 = df.Filter("x > 0")
        .Define("r2", "x*x + y*y")

rHist = df2.Histo1D("r2")

# Display histogram
rHist.Draw()
```

Spark backend

- Map-reduce workflow where every mapper runs the RDataFrame computation graph on a range of collision events
- Run analysis in C++ with Spark
- Through Python bindings



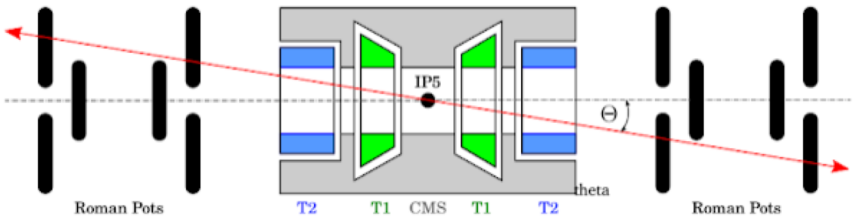
Access to C++ code from Python and Spark

- Include C++ header files
- Send local C++ libraries to distributed nodes
- PyROOT bindings make them available from Python

- Automatically import libraries
- Use Just-in-time (JIT) compiled code

```
import PyRDF

# Select Spark backend
PyRDF.use("spark")

# Add analysis headers
PyRDF.include("GetR2.h")

# Initialize RDataFrame object
df = PyRDF.RDataFrame(dataset)

# Operations run in Spark
df2 = df.Define("r2", "GetR2(x,y)")
```

TOTEM Analysis



- Proton-proton elastic scattering
- Data gathered by TOTEM experiment in 2015
  - Special LHC fill with optics parameter β^* adjusted to 90m
- Total of 4.7 TB data in ROOT Ntuple format
  - 1153 files
  - 2.8 Billion events
- Original analysis written using ROOT C++
- 2 stages:
  - Data reduction,
  - Filtering based on physics cuts

Conversion of TOTEM analysis to RDataFrame

- Original code: imperative C++ event processing loop

```
RPRootDumpTrackInfo rp_L_1_F = new RPRootDumpTrackInfo();
ch->SetBranchStatus("track_rp_5", 1);
ch->SetBranchAddress("track_rp_5", &rp_L_1_F);

TFile f_out = TFile::Open(fn_out.c_str(), "recreate");

EventRed ev;
TTree outT = new TTree("distilled", "treename")

outT->Branch("v_L_1_F", &ev.h.L_1_F.v);
outT->Branch("y_L_1_F", &ev.h.L_1_F.x);
…
for (; evi > ch->GetEntries(); evi++) {
  ch->GetEvent(evi);
  ev.h.L_1_F.v = rp_L_1_F->valid;
  ev.h.L_1_F.x = rp_L_1_F->x;

  bool skip = filterFunc(ev);
  if (skip)
    continue;

  outT->Fill()
}
```

Define data structure, mark branch to be read and bind the memory address with the variable

Define output file, output tree and Event Structure

Define branches of the output tree (outT) and bind them to the memory address with the value

Event loop starts

Read event number evi and automatically value of rp_L_1_F gets updated to element of next event

Apply filter to a given event

Fills tree with all current variable values pointing
to data on disk

Conversion to RDataFrame in Python

- Declarative applying of filters on the Data Frame structure
- Plus: we can do both steps at once:
  - Reduction
  - Producing histograms

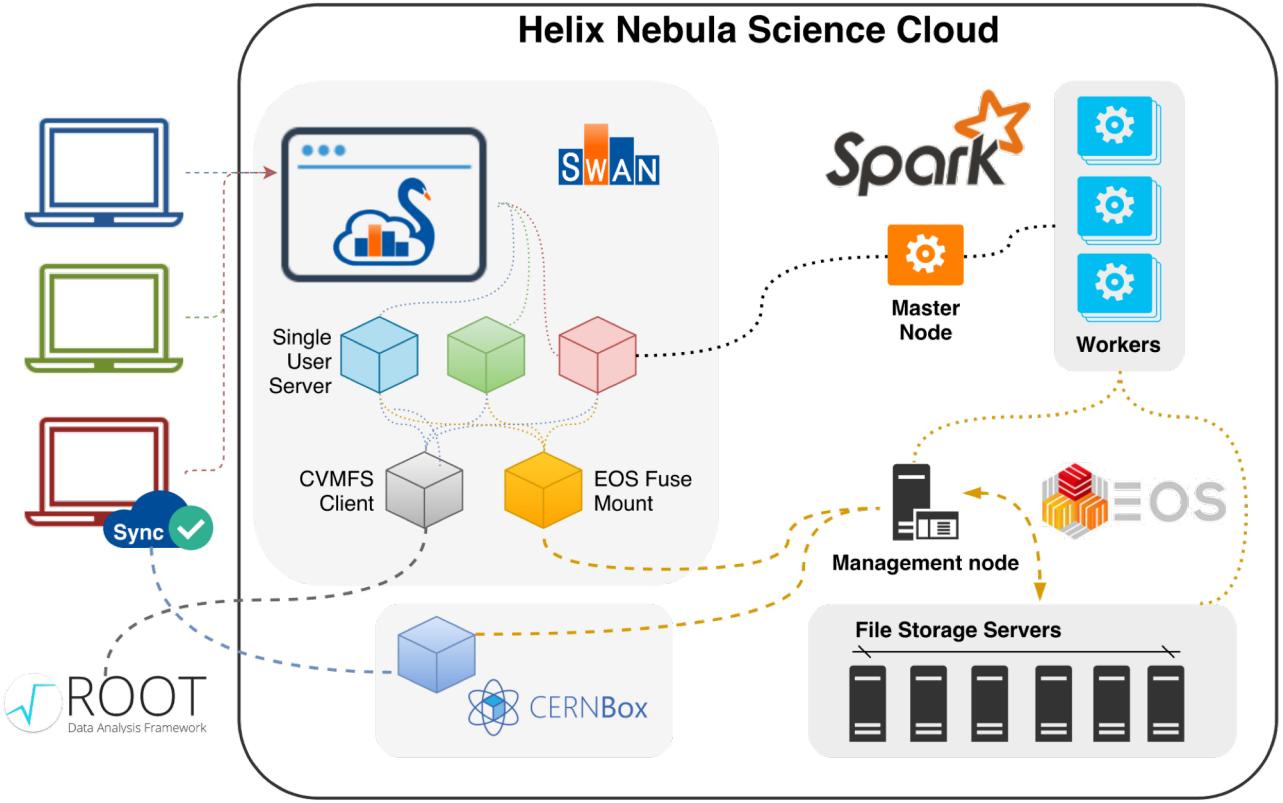
```
rdf = PyRDF.RDataFrame(treename, input_files)

branchList = ["v_L_1_F", "x_L_1_F", "y_L_1_F", ... ]
...
r = rdf.Filter(filter_code_string) \
       .Define("v_L_1_F", track_rp_5.valid)
       .Define("x_L_1_F", track_rp_5.x)
       .Define("y_L_1_F", track_rp_5.y)
            ...
       .Snapshot(outTreeName, outFileName, branchList)
```

Components used in evaluation

- ***SWAN:*** web-based platform for interactive data analysis at CERN, based on Jupyter notebook interface and integrated with the ROOT analysis framework with the dedicated ROOT C++ kernel
- ***EOS:*** a distributed storage system used to host all physics data at CERN. A dedicated EOS instance storing a subset of the TOTEM experiment data is deployed on the HNSciCloud
- ***Spark cluster:*** deployed on the HNSciCloud and integrated with SWAN
- ***CERNBox:*** to manage and share files stored on EOS and SWAN notebooks between users; synchronization with personal laptops.

All tools available as a containerized setup, deployable on clouds and on premises

Testbed architecture



Spark Cluster:

57 nodes

- 32 cores
- 128 GiB RAM

TOTAL

- 1824 vCPUs
- 7296 GiB RAM

Interactive interface

Jupyter notebooks

DistillDistibution-AllDatasets

Last Checkpoint: 3 hours ago (unsaved changes)

FILE EDIT VIEW INSERT CELL KERNEL HELP Not Trusted Python 2

ROOT Data Analysis Framework

Distributing a TOTEM analysis with RDataFrame in Python

```
In [2]: # Branches clasified by diagonal
diagonals = {
    # return a tuple: ([left] verticals in 45, [right] verticals in 56))
    "d45b_56t"  : (["track_rp_5", "track_rp_21", "track_rp_25"],
                  ["track_rp_104", "track_rp_120", "track_rp_124"]),
    "ad45b_56b" : (["track_rp_5", "track_rp_21", "track_rp_25"],
                  ["track_rp_105", "track_rp_121", "track_rp_125"]),
    "d45t_56b"  : (["track_rp_4", "track_rp_20", "track_rp_24"],
                  ["track_rp_105", "track_rp_121", "track_rp_125"]),
    "ad45t_56t" : (["track_rp_4", "track_rp_20", "track_rp_24"],
                  ["track_rp_104", "track_rp_120", "track_rp_124"])
}

# Data source files
DS = {
    'DS1'   : '4495',
    'DS2'   : '4496',
    'DS3'   : '4499',
    'DS4'   : '4505',
    'DS5'   : '4509',
    'DS6'   : '4510',
    'DS7'   : '4511'
}

# Select branches
selected_diagonal = "d45b_56t"

rp_left, rp_right = diagonals[selected_diagonal]

# Columns per branch
attributes = ['valid', 'x', 'y']

full_branches = ["{}.{}".format(c,a) for a in attributes for c in rp_left+rp_right ]
```

Spark Monitoring

```
In [*]: %%time

# Name of the ROOT Tree
input_ntuple_name = "TotemNtuple"


selected_ds = ['DS1', 'DS2', 'DS3', 'DS4', 'DS5', 'DS6', 'DS7']

source_files = ["input_files_{}.txt".format(ds) for ds in selected_ds]

# Remote path in EOS
prefixes            = ["root://eostotem//eos/totem/data/cmstotem/2015/90m/Totem/Ntuple/version2/{}/".format(DS[ds]) for ds
input_files         = [prefixes[i] + line.rstrip('\r\n') for i in range(len(source_files)) for line in open(source_files[i])

# All datasets
dTree = DistTree(input_files,
                 treename = "TotemNtuple",
                 npartitions = 512)

histos = dTree.ProcessAndMerge(fillHist, mergeHist)
```

Apache Spark: 8 EXECUTORS 256 CORES Jobs: 1 RUNNING

Job ID	Job Name	Status	Stages	Tasks	Submission Time	Duration
0	treeReduce	RUNNING	0/2 (1 active)	111 + 256 / 554	2 minutes ago	-

Stage Id	Stage Name	Status	Tasks	Submission Time	Duration
0	treeReduce	RUNNING	111 + 256 / 512	2 minutes ago	-
1	treeReduce	PENDING	0 / 20	Unknown	-

Apache Spark: 8 EXECUTORS 256 CORES Jobs: 1 RUNNING

Event Timeline

Show task phases

5 December 12:06 | 5 December 12:07 | 5 December 12:08

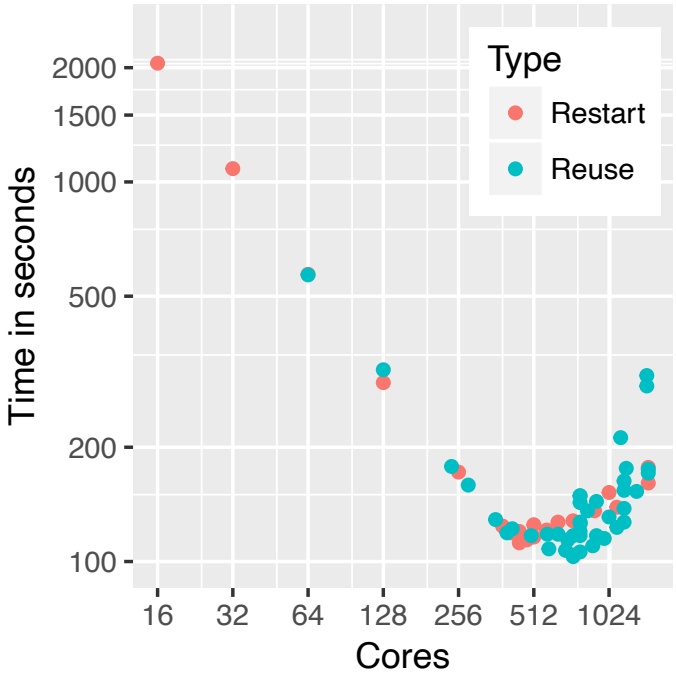
Jobs: 0:treeReduce

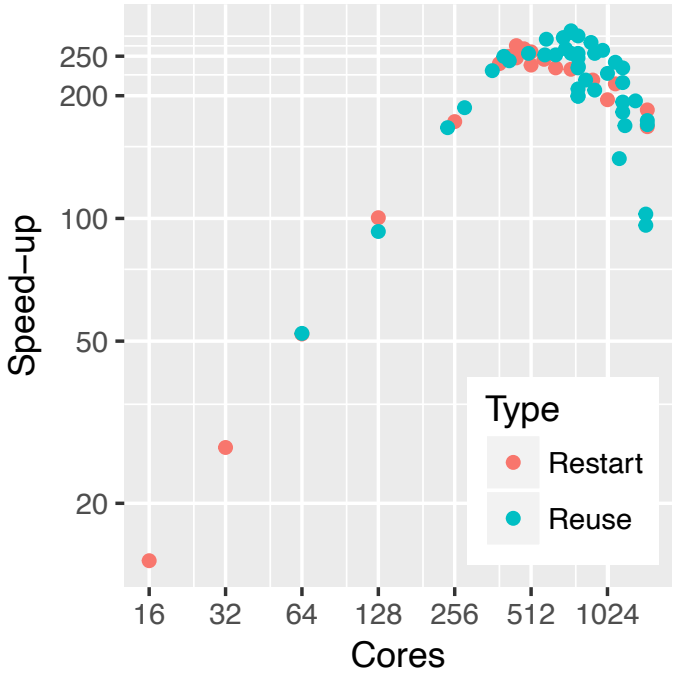
Stages: 0:treeReduce

Tasks: 5 spark-w-0039.novalo

0	314
8	372
16	365
24	377
32	385
40	364
48	362
56	350
64	374
72	352
80	337

Time and Speedup





Configurations tested: Spark executor *reuse* vs. *restart* policy

Experiment results

- Correctness
  - Automatic comparison with original results via scripts and histograms
  - Carefully checked to be identical with the original analysis
- Scalability
  - 8h15m20 at 1 core
  - 1m43s at 736 cores → speedup of 280x
- Interactivity
  - Use of interactive tools (SWAN)
  - Reduction of processing time
- Portability
  - Easy migration between HNSciCloud (external cloud) and Spark cluster at CERN

Future Work

- Tests in multi-user environment: to evaluate the performance and usability
- Tests on larger sample (TOTEM and CMS common data): to push scaling limits further
  - Currently 500+TB
  - More complex data structures from CMS Software
- Use of new cloud technologies:
  - Evaluation of public cloud providers
  - New cloud services (cloud storage, serverless computing)

Conclusions

- Cloud usage: Successful migration of CERN software stack (SWAN, ROOT, EOS, Spark) to external cloud service
- Data Analysis: new programming model supports real data analysis from TOTEM experiment
- Performance: new tools enable nearly interactive data analysis on a full dataset
- Results: Computing time reduced from 8h to 1m43s , speedup of 280x
- Great collaborative work of multiple teams

References

- TOTEM: http://totem.web.cern.ch
- SWAN: https://swan.web.cern.ch/
- ROOT: https://root.cern.ch/
- EOS: https://github.com/cern-eos/eos
- CERNBox: https://cernbox.web.cern.ch/
- Helix Nebula: http://www.helix-nebula.eu
- Science Box: https://sciencebox.web.cern.ch

This work was supported in part by the Polish Ministry of Science and Higher Education, grant DIR/WK/2018/13.

Books

- Matei Zaharia, Holden Karau, Andy Konwinski, Patrick Wendell: Learning Spark, Lightning-Fast Big Data Analysis, O'Reilly Media, 2015
- Matei Zaharia, Bill Chambers: Spark: The Definitive Guide, Big Data Processing Made Simple; O'Reilly Media, 2018