POLITECNICO
MILANO 1863

SYSTEMS AND METHODS FOR BIG AND UNSTRUCTURED DATA

SPARK

Marco Brambilla

marco.brambilla@polimi.it

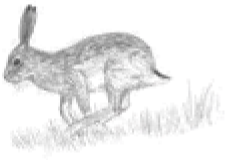
@marcobrambi

What is Big Data used For?

- Reports, e.g.,
  - Track business processes, transactions
- Diagnosis, e.g.,
  - Why is user engagement dropping?
  - Why is the system slow?
  - Detect spam, worms, viruses, DDoS attacks
- Decisions, e.g.,
  - Personalized medical treatment
  - Decide what feature to add to a product
  - Decide what ads to show

Data is only as useful as the decisions it enables

Data Processing Goals



Low latency (interactive) queries on historical data: enable faster decisions

- E.g., identify why a site is slow and fix it

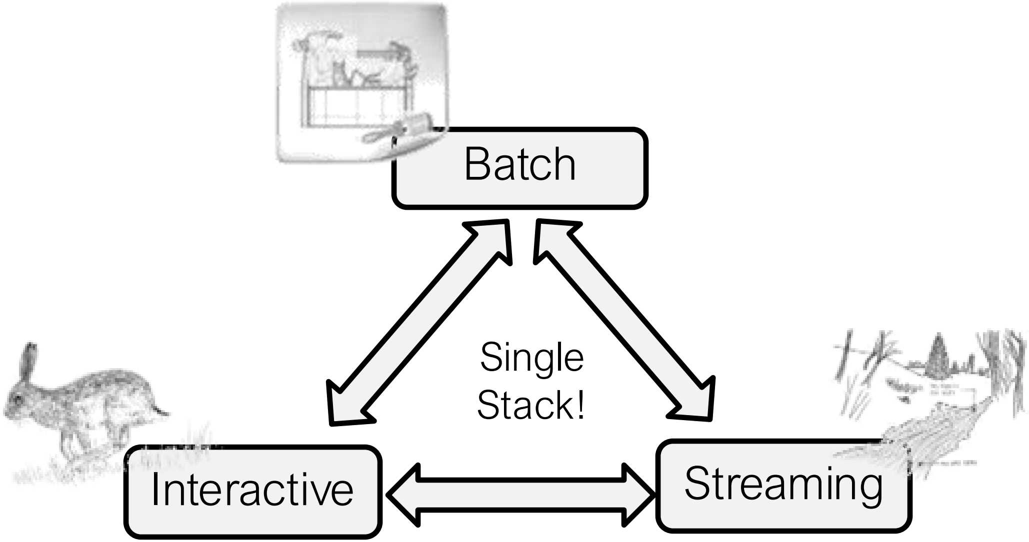
Low latency queries on live data (streaming): enable decisions on real-time data

- E.g., detect & block worms in real-time (a worm may infect **1mil** hosts in **1.3sec**)

Sophisticated data processing: enable "better" decisions

- E.g., anomaly detection, trend analysis

Goal



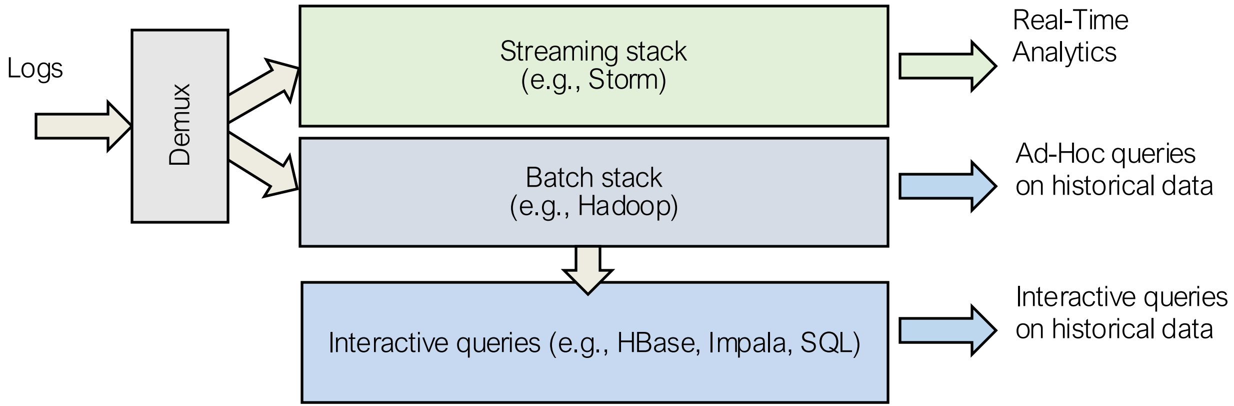
Support *batch*, *streaming*, and *interactive* computations…

… and make it easy to compose them

Easy to develop *sophisticated* algorithms (e.g., graph, ML algos)

The Need for Unification (1/2)

Today's state-of-art analytics stack



Challenges:

» Need to maintain three separate stacks
  - Expensive and complex
  - Hard to compute consistent metrics across stacks

» Hard and slow to share data across stacks

The Need for Unification (2/2)

Make real-time decisions
- Detect DDoS, fraud, etc

E.g.,: what's needed to detect a DDoS attack?
1. Detect attack pattern in real time → streaming
2. Is traffic surge expected? → interactive queries
3. Making queries fast → pre-computation (batch)

And need to implement complex algos (e.g., ML)!

Data Processing Stack

Data Processing Layer

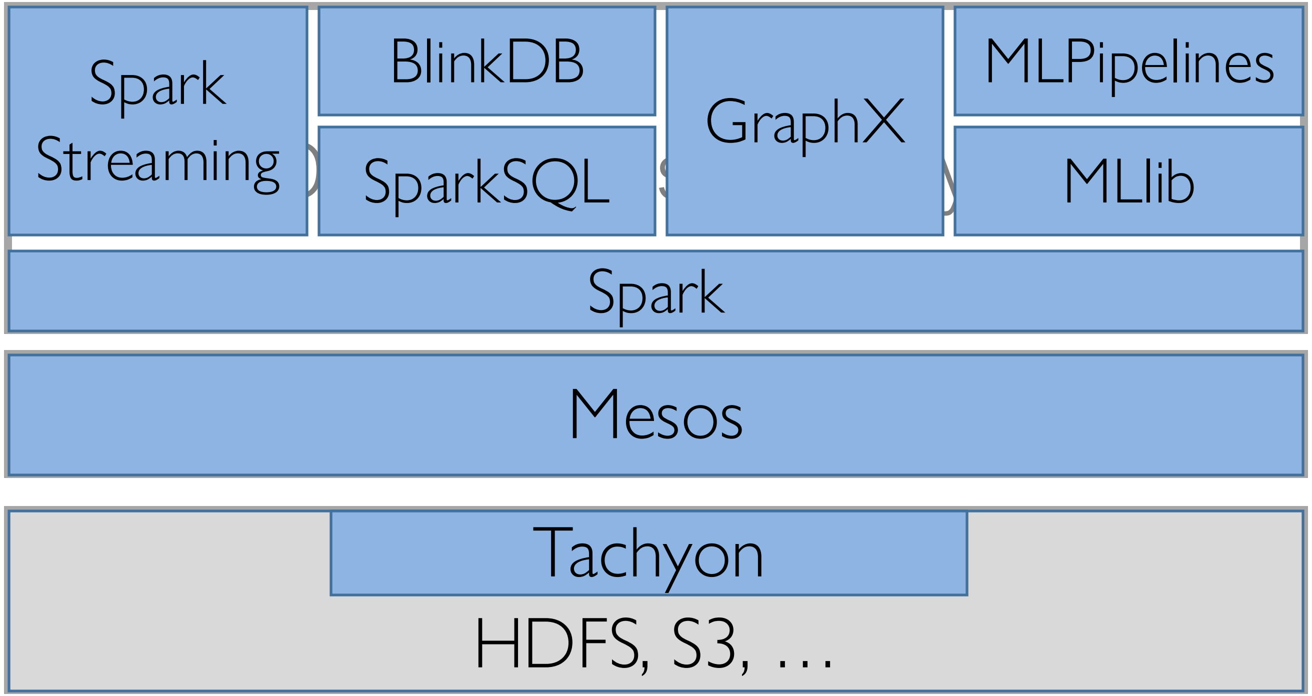
Resource Management Layer

Storage Layer

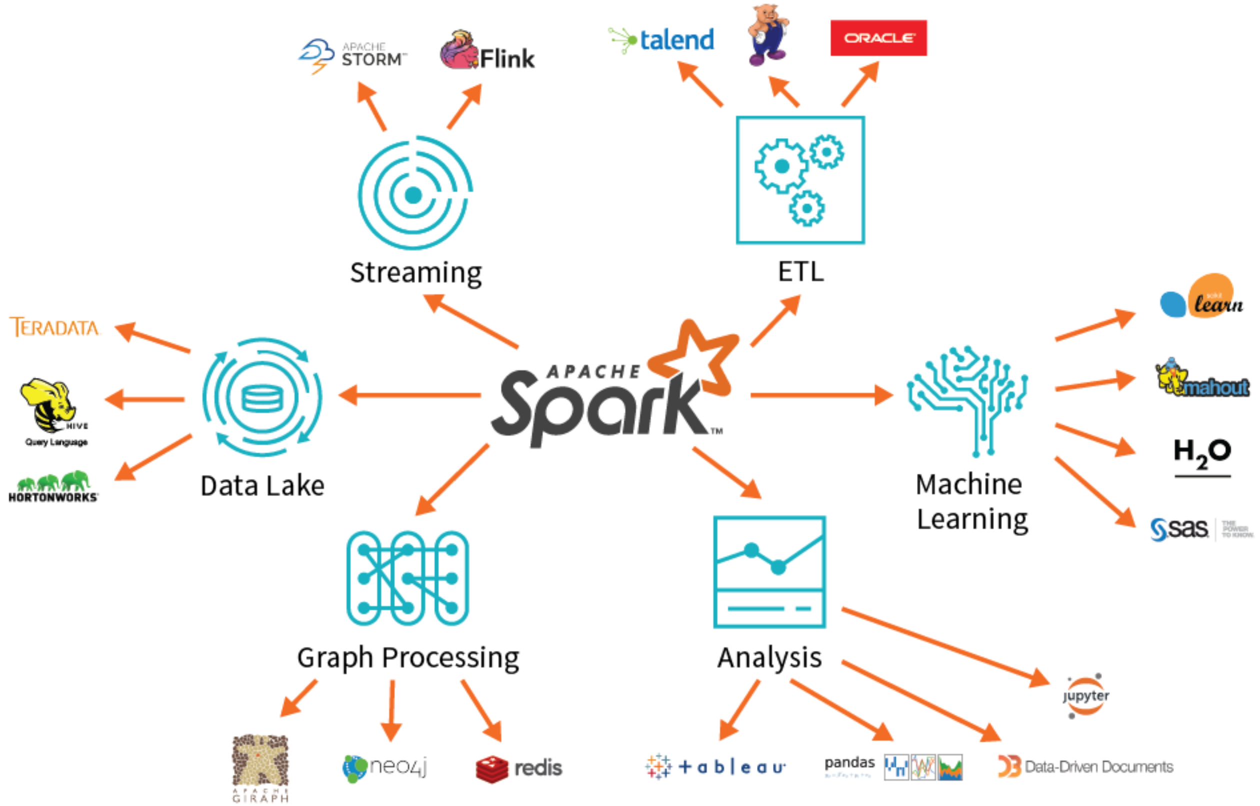
Spark

- An open source project on Apache
- First released in February 2013 and has exploded in popularity due to it’s ease of use and speed
- It was created at the AMPLab at UC Berkeley

Berkeley Data Analytics Stack



Spark



APACHE STORM
Flink
talend
ORACLE
Streaming
ETL
TERADATA
HIVE
Query Language
HORTONWORKS
Data Lake
APACHE Spark
learn
mahout
H_2O
sas
Machine Learning
Graph Processing
Analysis
jupyter
APACHE GIRAPH
neo4j
redis
+ a b l e a u
pandas
Data-Driven Documents

HDFS

HDFS Overview

Responsible for storing data on the cluster

Data files are split into blocks and distributed across the nodes in the cluster

Each block is replicated multiple times

HDFS Basic Concepts

HDFS is a file system written in Java based on the Google's GFS

Provides redundant storage for massive amounts of data

HDFS Basic Concepts

HDFS works best with a smaller number of large files

- Millions as opposed to billions of files
- Typically 100MB or more per file

Files in HDFS are write once

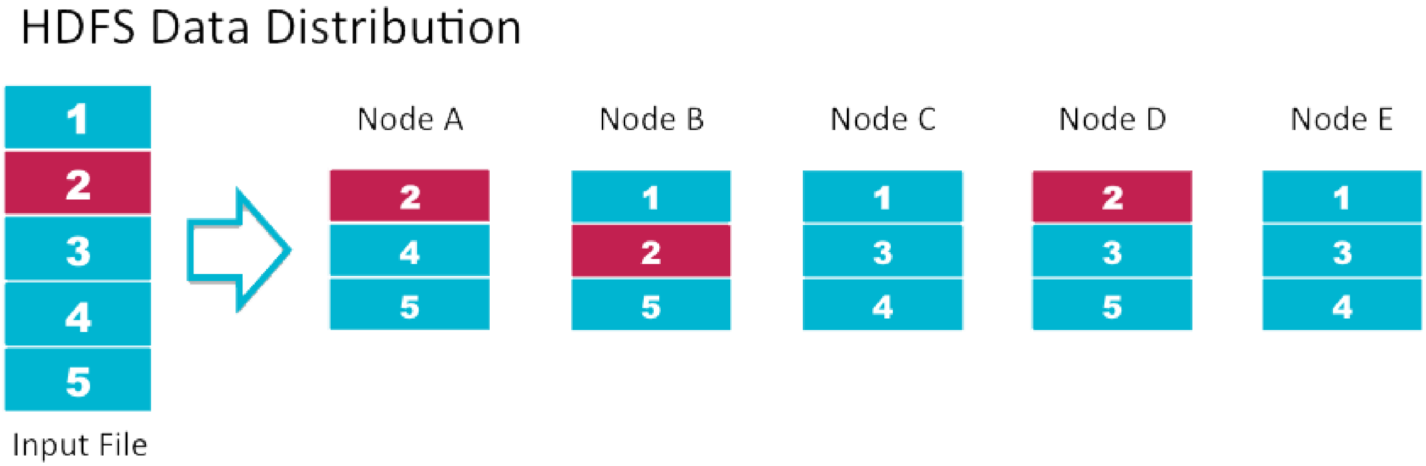
Optimized for streaming reads of large files and not random reads

How are Files Stored

- Files are split into blocks
- Blocks are split across many machines at load time
  - Different blocks from the same file will be stored on different machines
- Blocks are replicated across multiple machines
- The NameNode keeps track of which blocks make up a file and where they are stored

Data Replication

Default replication is 3-fold



Goals of HDFS

Very Large Distributed File System

- 10K nodes, 100 million files, 10PB

Assumes Commodity Hardware

- Files are replicated to handle hardware failure
- Detect failures and recover from them

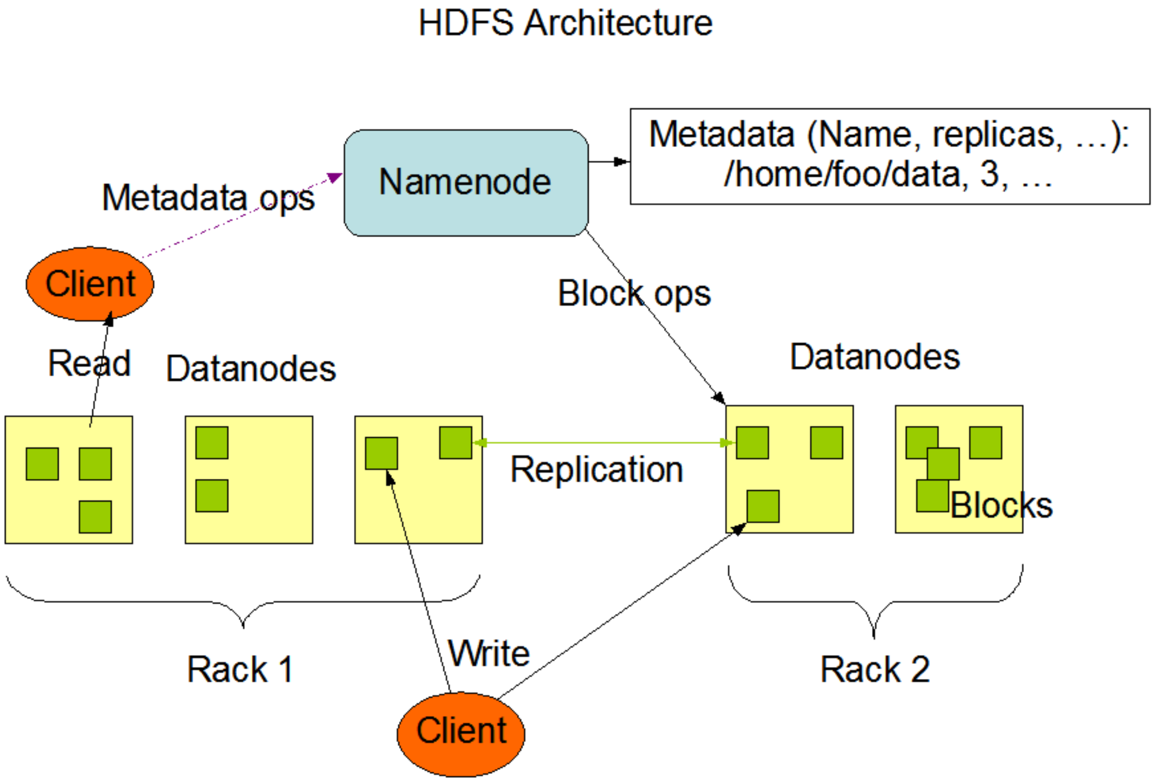
Optimized for Batch Processing

- Data locations exposed so that computations can move to where data resides
- Provides very high aggregate bandwidth

Distributed File System

- **Single Namespace** for entire cluster
- Data Coherency
  - **Write-once-read-many** access model
  - Client can only append to existing files
- Files are broken up into blocks
  - Typically 64MB block size
  - Each block replicated on multiple DataNodes
- Intelligent Client
  - Client can find location of blocks
  - Client accesses data directly from DataNode

HDFS Architecture



Marco Brambilla.

Functions of a NameNode

- Manages File System Namespace
  - Maps a file name to a set of blocks
  - Maps a block to the DataNodes where it resides
- Cluster Configuration Management
- Replication Engine for Blocks

NameNode Metadata

- Metadata in Memory
  - The entire metadata is in main memory
  - No demand paging of metadata
- Types of metadata
  - List of files
  - List of Blocks for each file
  - List of DataNodes for each block
  - File attributes, e.g. creation time, replication factor
- A Transaction Log
  - Records file creations, file deletions etc

DataNode

- A Block Server
  - Stores data in the local file system (e.g. ext3)
  - Stores metadata of a block (e.g. CRC)
  - Serves data and metadata to Clients
- Block Report
  - Periodically sends a report of all existing blocks to the NameNode
- Facilitates Pipelining of Data
  - Forwards data to other specified DataNodes

Block Placement

Strategy

- One replica on local node
- Second replica on a remote rack
- Third replica on same remote rack
- Additional replicas are randomly placed

Clients read from nearest replicas

Heartbeats

- DataNodes send hearbeat to the NameNode
  - Once every 3 seconds
- NameNode uses heartbeats to detect DataNode failure

Replication Engine

NameNode detects DataNode failures

- Chooses new DataNodes for new replicas
- Balances disk usage
- Balances communication traffic to DataNodes

Data Correctness

- Use Checksums to validate data
  - Use CRC32
- File Creation
  - Client computes checksum per 512 bytes
  - DataNode stores the checksum
- File access
  - Client retrieves the data and checksum from DataNode
  - If Validation fails, Client tries other replicas

NameNode Failure

- A single point of failure in HDFS 1
- Transaction Log stored in multiple directories
  - A directory on the local file system
  - A directory on a remote file system (NFS/CIFS)

Data Pipelining

Client retrieves a list of DataNodes on which to place replicas of a block

Client writes block to the first DataNode

The first DataNode forwards the data to the next node in the Pipeline

When all replicas are written, the Client moves on to write the next block in file

Rebalancer

Goal: % disk full on DataNodes should be similar

- Usually run when new DataNodes are added
- Cluster is online when Rebalancer is active
- Rebalancer is throttled to avoid network congestion

Secondary NameNode

Copies FsImage and Transaction Log from Namenode to a temporary directory

Merges FSImage and Transaction Log into a new FSImage in temporary directory

Uploads new FSImage to the NameNode

- Transaction Log on NameNode is purged

User Interface

Commads for HDFS User:

hadoop dfs -mkdir /foodir
hadoop dfs -cat /foodir/myfile.txt
hadoop dfs -rm /foodir/myfile.txt

Commands for HDFS Administrator

hadoop dfsadmin -report
hadoop dfsadmin -decommision datanodename

Web Interface

http://host:port/dfshealth.jsp

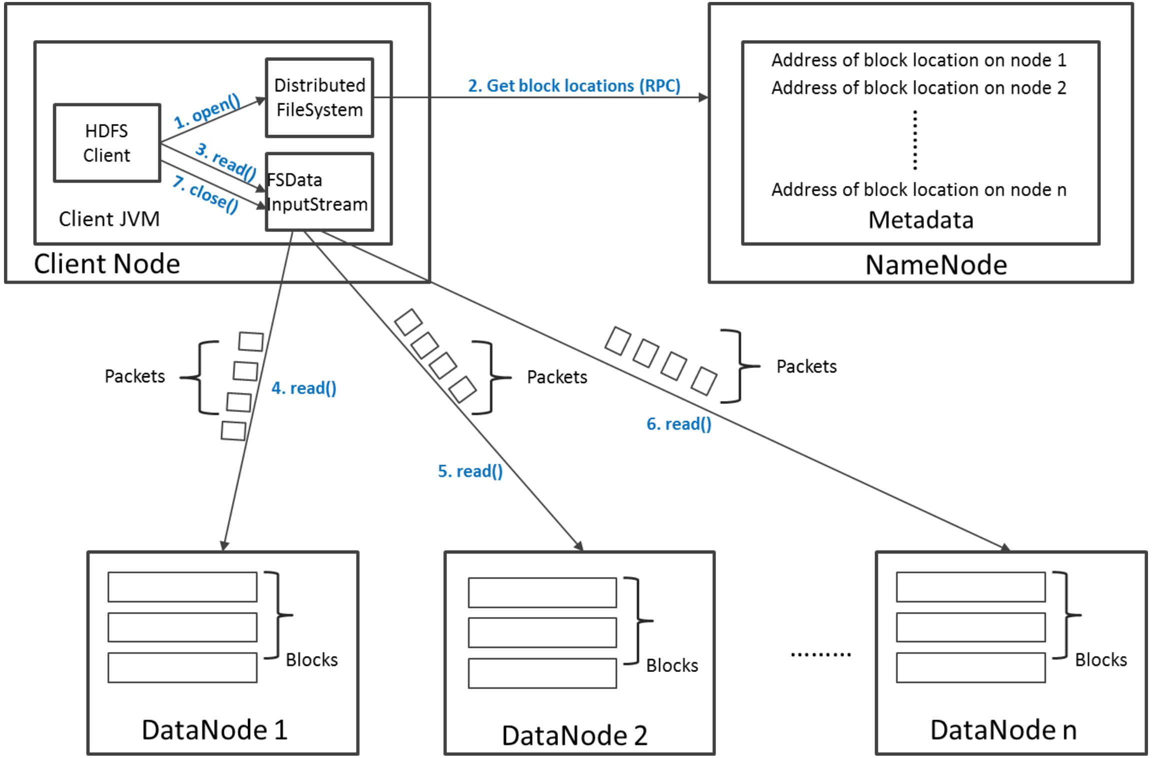
Data Retrieval

When a client wants to retrieve data

- Communicates with the NameNode to determine which blocks make up a file and on which data nodes those blocks are stored

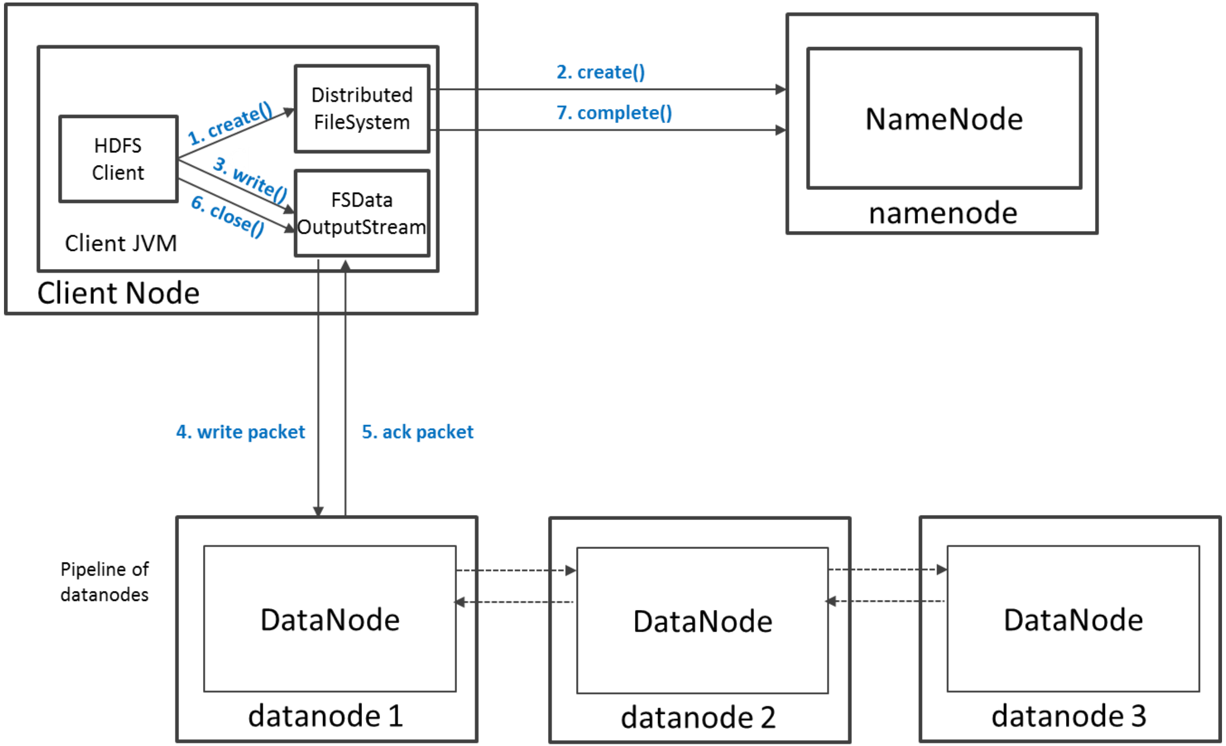
- Then communicated directly with the data nodes to read the data

Read Operation in HDFS



Marco Brambilla.

Write Operation in HDFS



Marco Brambilla.

HDFS Security

- Authentication to Hadoop
  - Simple – insecure way of using OS username to determine hadoop identity
  - Kerberos – authentication using kerberos ticket
  - Set by `hadoop.security.authentication=simple|kerberos`
- File and Directory permissions are same like in POSIX
  - read (r), write (w), and execute (x) permissions
  - also has an owner, group and mode
  - enabled by default (`dfs.permissions.enabled=true`)
- ACLs are used for implemention permissions that differ from natural hierarchy of users and groups
  - enabled by `dfs.namenode.acls.enabled=true`

HDFS Configuration

HDFS Defaults

Block Size – 64 MB

Replication Factor – 3

Web UI Port – 50070

HDFS conf file - `/etc/hadoop/conf/hdfs-site.xml`

Interfaces to HDFS

Java API `(DistributedFileSystem)`

C wrapper `(libhdfs)`

HTTP protocol

WebDAV protocol

Shell Commands

However the command line is one of the simplest and most familiar

HDFS – Shell Commands

There are two types of shell commands

User Commands

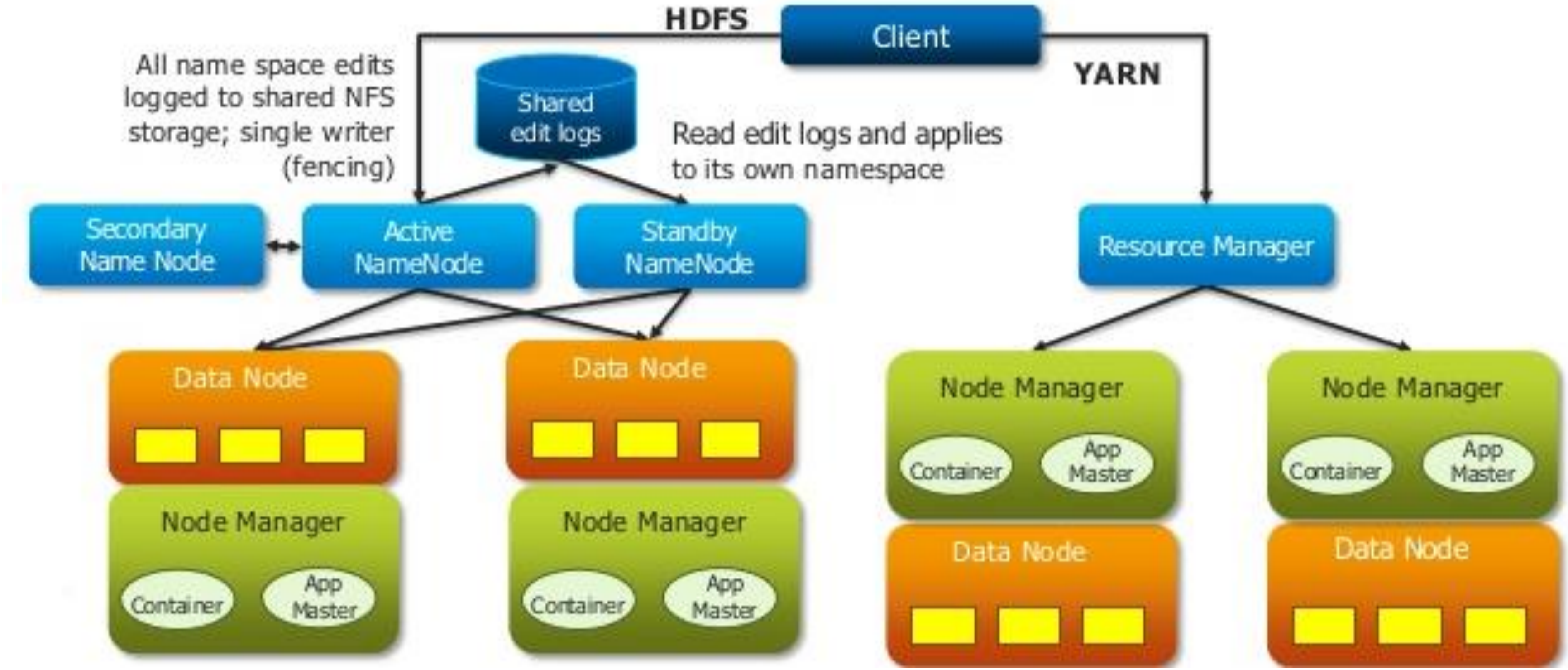
- `hdfs dfs` – runs filesystem commands on the HDFS
- `hdfs fsck` – runs a HDFS filesystem checking command

Administration Commands

- `hdfs dfsadmin` – runs HDFS administration commands

Hadoop

Hadoop 2.0 Architecture - YARN



Spark

History

Apache Spark started as a research project at the University of California AMPLab, in 2009 by Matei Zaharia.

In 2013

- donated to the Apache Software Foundation
- open sourced, adopted the Apache 2.0 license

In February 2014, Spark became a Top-Level Apache Project.

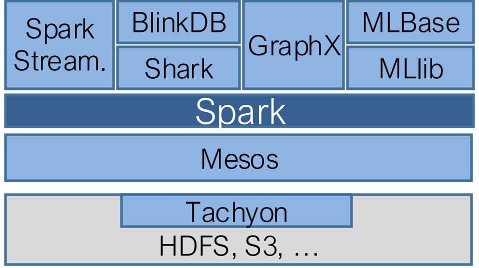
In November 2014, Spark founder Matei_Zaharia's company Databricks set a new world record in large scale sorting using Spark.

Latest stable release: CLICK-HERE

600,000+ lines of code (75% Scala)

Built by 1,000+ developers from more than 250+ organizations

Apache Spark



Distributed Execution Engine
- Fault-tolerant, efficient in-memory storage (RDDs)
- Powerful programming model and APIs (Scala, Python, Java)

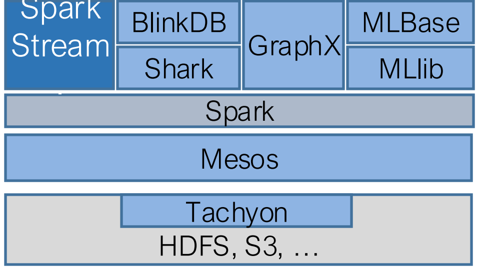
Fast: up to 100x faster than Hadoop

Easy to use: 5-10x less code than Hadoop

General: support interactive & iterative apps

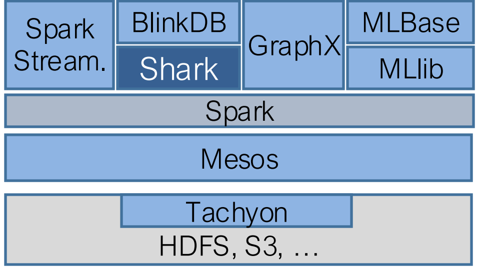
Two major releases since last AMPCamp

Spark Streaming



- Large scale streaming computation
- Implement streaming as a sequence of <1s jobs
  - Fault tolerant
  - Handle stragglers
  - Ensure exactly one semantics
- Integrated with Spark: unifies batch, interactive, and batch computations

SparkSQL



Hive over Spark: full support for HQL and UDFs

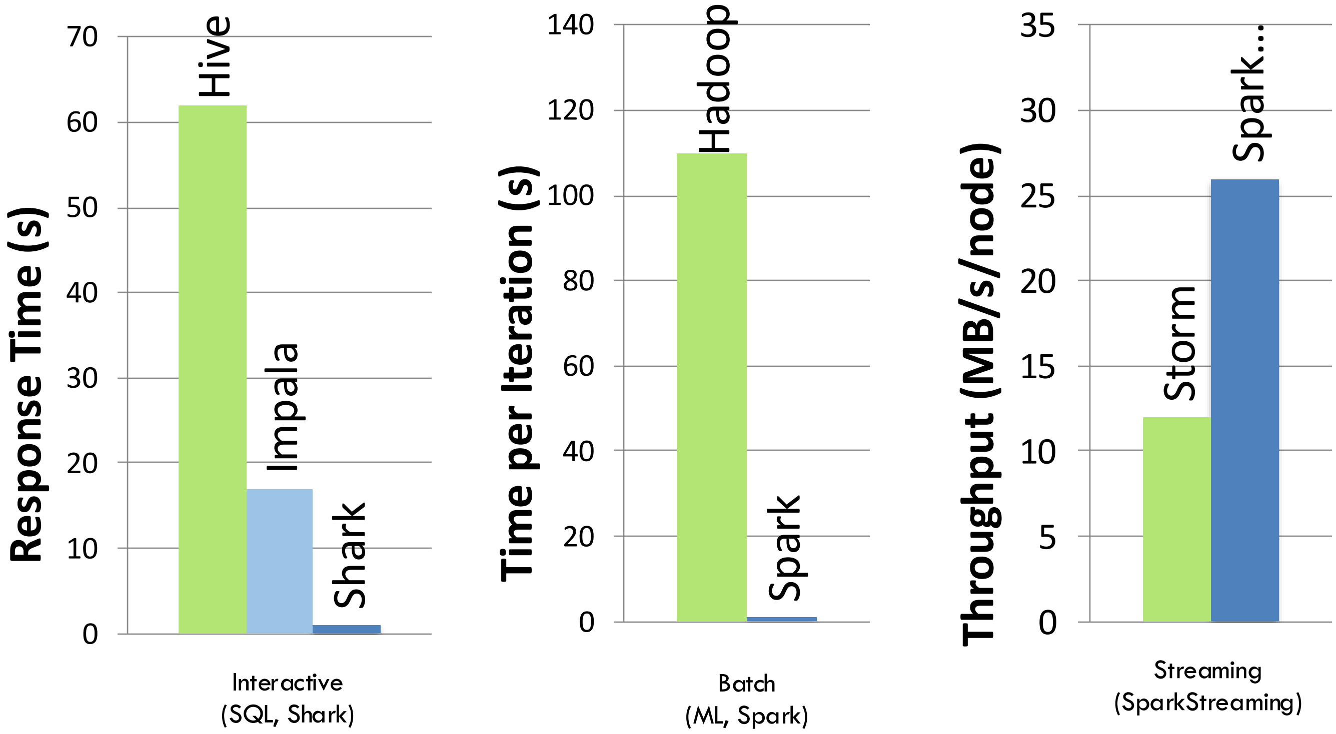
Up to 100x when input is in memory

Up to 5-10x when input is on disk

Running on hundreds of nodes at Yahoo!

Two major releases along Spark

Performance and Generality (Unified Computation Models)



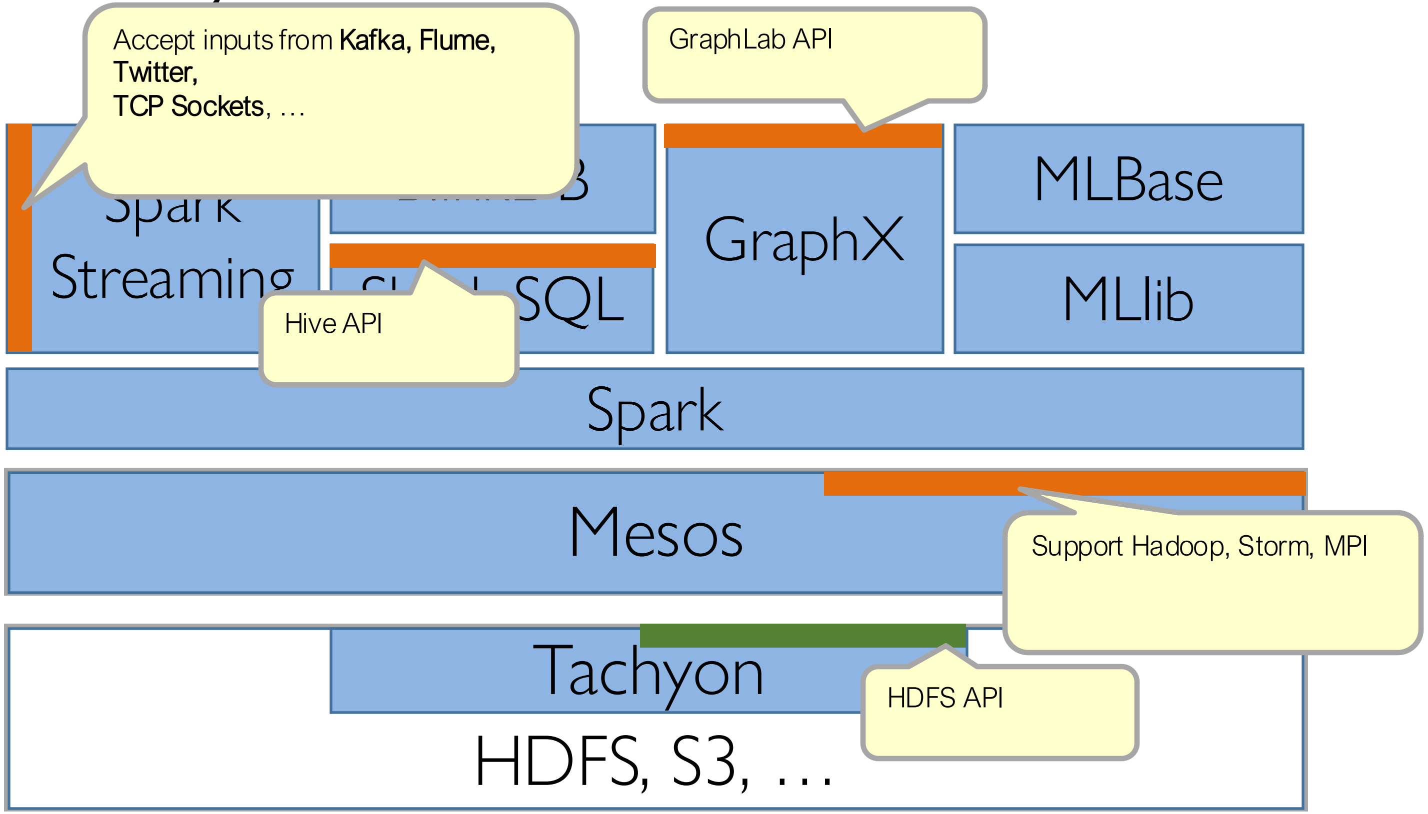
Unified Programming Models

Unified system for SQL, graph processing, machine learning

All share the same set of workers and caches

```
def logRegress(points: RDD[Point]): Vector {
  var w = Vector(D, _ => 2 * rand.nextDouble - 1)
  for (i <- 1 to ITERATIONS) {
    val gradient = points.map { p =>
      val denom = 1 + exp(-p.y * (w dot p.x))
      (1 / denom - 1) * p.y * p.x
    }.reduce(_ + _)
    w -= gradient
  }
  w
}

val users = sql2rdd("SELECT * FROM user u
   JOIN comment c ON c.uid=u.uid")

val features = users.mapRows { row =>
  new Vector(extractFeature1(row.getInt("age")),
             extractFeature2(row.getStr("country")),
             ...)}
val trainedVector = logRegress(features.cache())
```

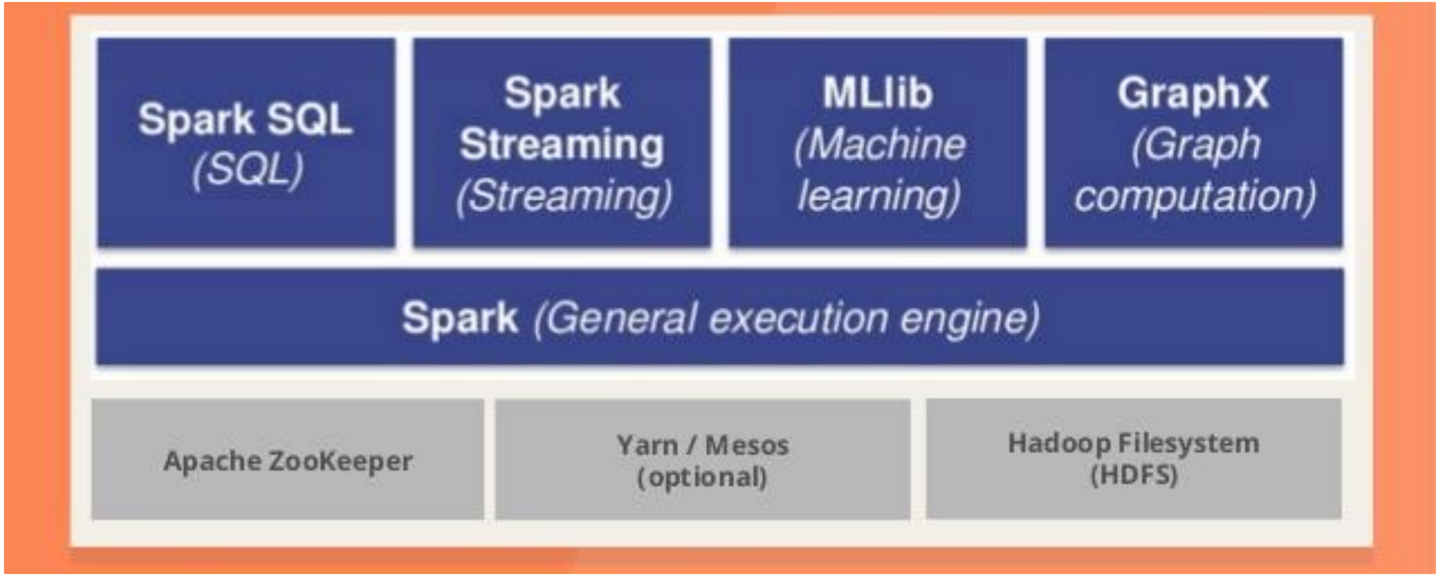
Compatibility to Existing Ecosystem



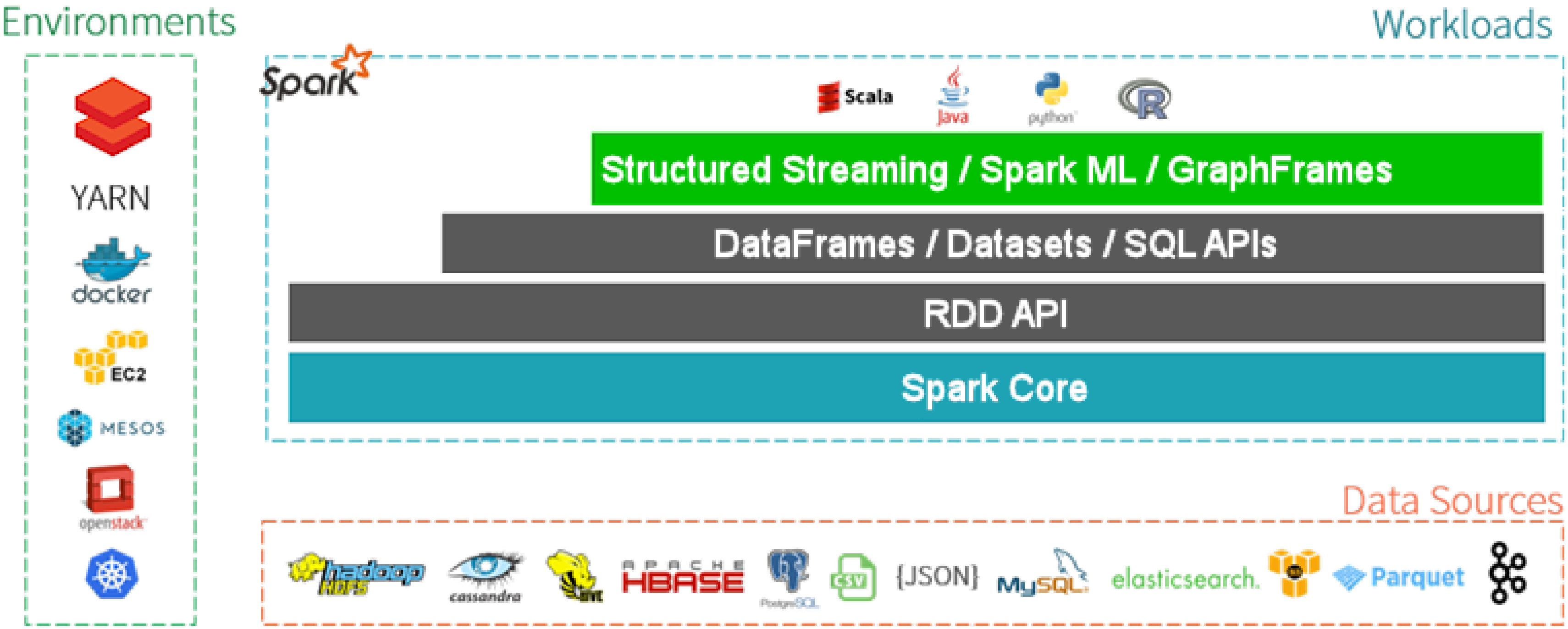
Spark

- You can think of Spark as **scalable computation platform**
- Spark can use data stored in a variety of formats
  - Cassandra
  - AWS S3
  - HDFS
  - And more

SPARK in context



Spark Components



Marco Brambilla.

History of Spark APIs

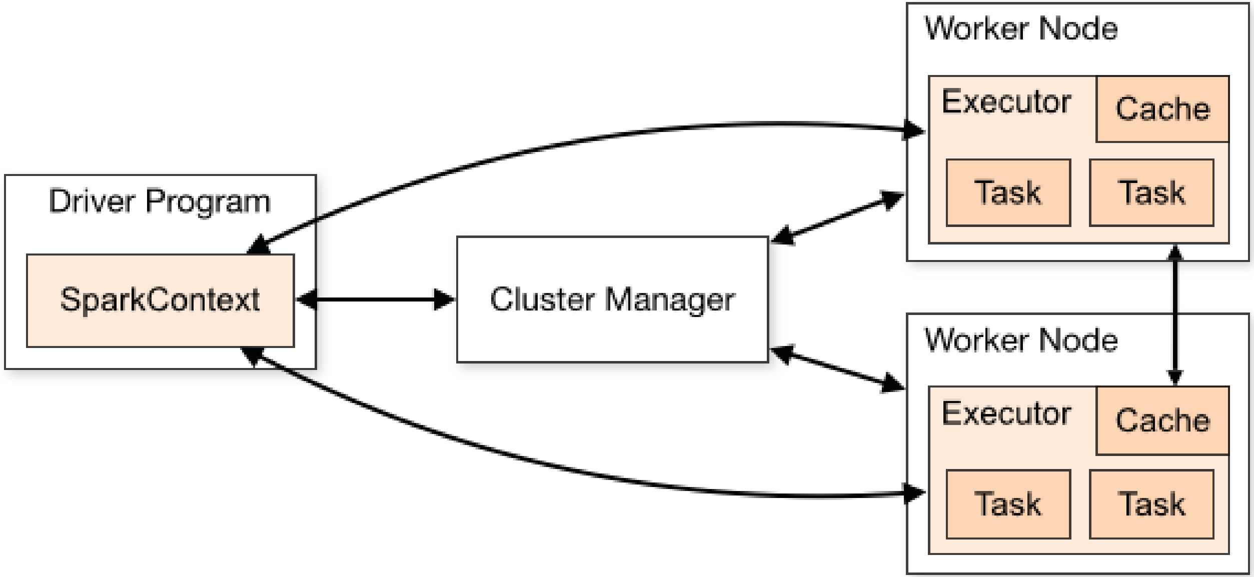
RDD (2011)	DataFrame (2013)	DataSet (2015)
Distribute collection of JVM objects	Distribute collection of Row objects	Internally rows, externally JVM objects
Functional Operators (map, filter, etc.)	Expression-based operations and UDFs	Almost the “Best of both worlds”: type safe + fast
	Logical plans and optimizer	But slower than DF Not as good for interactive analysis, especially Python
	Fast/efficient internal representations	

databricks

Spark RDDs

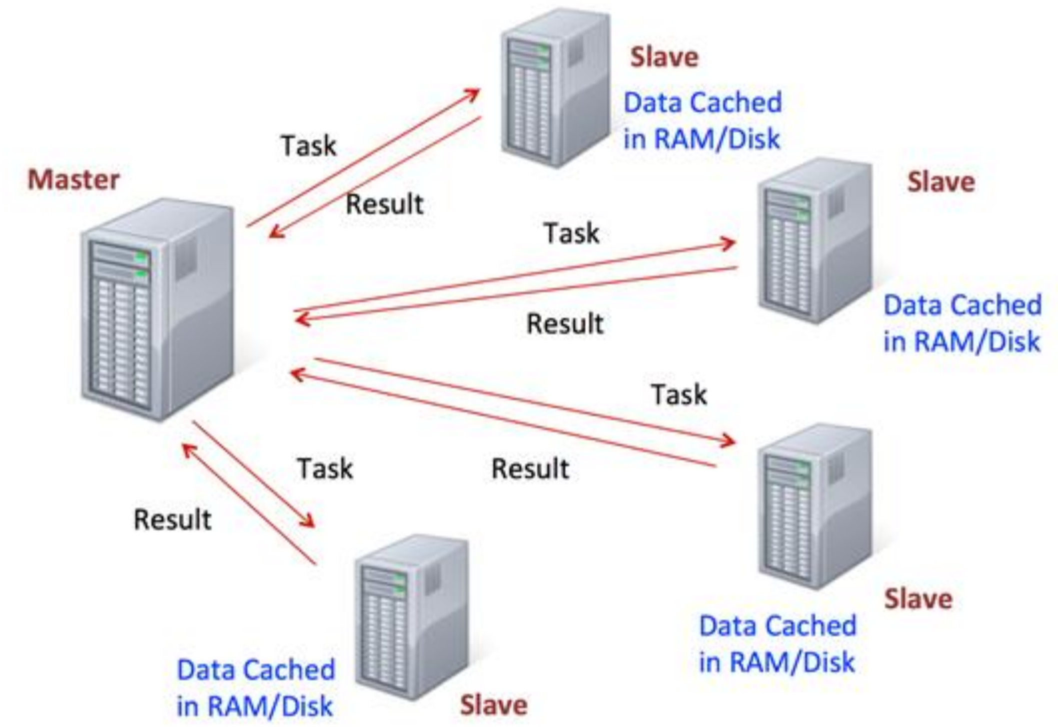
- At the core of Spark is the idea of a Resilient Distributed Dataset (RDD)
- Resilient Distributed Dataset (RDD) has 4 main features:
  - Distributed Collection of Data
  - Fault-tolerant
  - Parallel operation - partioned
  - Ability to use many data sources

Spark RDDs



Marco Brambilla.

Spark RDDs



Slave
Data Cached
in RAM/Disk
Task
Master
Result
Slave
Task
Data Cached
in RAM/Disk
Result
Task
Task
Result
Result
Data Cached
in RAM/Disk
Slave
Slave
Data Cached
in RAM/Disk

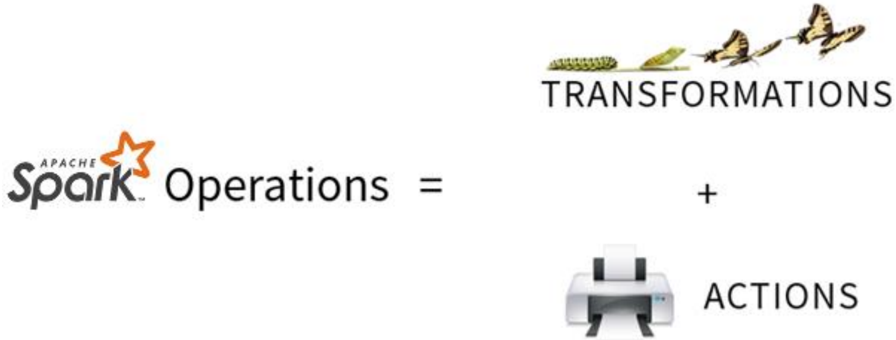
Spark RDDs

RDDs are:

- immutable,
- lazily evaluated,
- cacheable.

Spark Operations

- There are two types of Spark operations:
    - Transformations
    - Actions
- **Transformations** are a recipe to follow.
- **Actions** perform what the recipe says to do and returns something back.



Spark Operations

- This carries over to the syntax when coding.
- you write a method call, but won't see anything as a result until you call the action.
- Why?

With a large dataset, you don't want to calculate all the transformations until you are sure you want to perform them!

RDD operations

transformations to build RDDs through deterministic operations on other RDDs

- transformations include *map*, *filter*, *join*
- lazy operation

actions to return value or export data

- actions include *count*, *collect*, *save*
- triggers execution

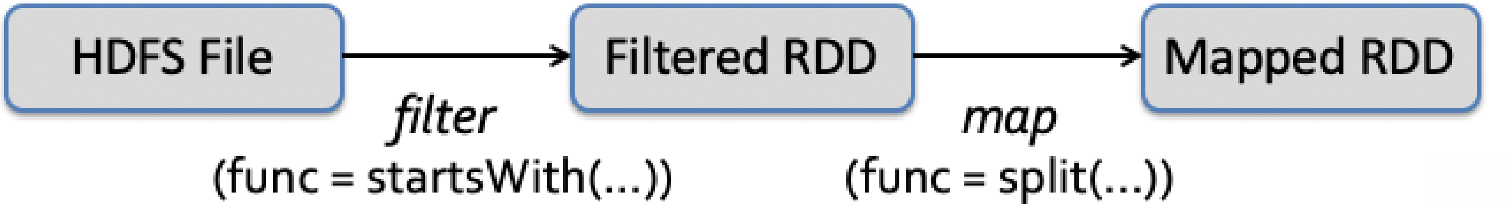
Operations

Transformations *(lazy)*	Actions
`select`	`show`
`distinct`	`count`
`groupBy`	`collect`
`sum`	`save`
`orderBy`	
`filter`	
`limit`	

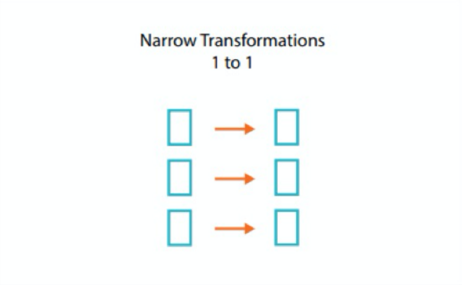
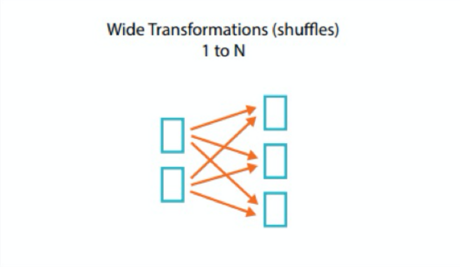
Transformations

RDD are immutable, transformations create new RDD

msgs = textFile.filter(lambda s: s.startsWith("ERROR")) .map(lambda s: s.split("\t")[2])



Narrow vs. Wide

Narrow Transformations	Wide Transformations
The data required to compute the records in a single partition reside in at most one partition of the parent RDD.	The data required to compute the records in a single partition may reside in many partitions of the parent RDD.
	
* `filter(..)`	* `distinct()`
* `drop(..)`	* `groupBy(..).sum()`
* `coalesce()`	* `repartition(n)`

Spark RDDs and DataFrames

- With Spark you can find: RDD syntax versus DataFrame syntax discussions. Why?
- With Spark 2.0, Spark is moving towards a DataFrame based syntax

- But the way files are being distributed can still be thought of as RDDs, it is just the typed out syntax that is changing

DataFrame

Definition

- Immutable Data with named columns (built on RDDs)

Characteristics

- User-friendly API
- Uniform APIs across languages (Scala, Java, Python, R, and SQL)
- Improved performance via optimizations (Tungsten and Catalyst)

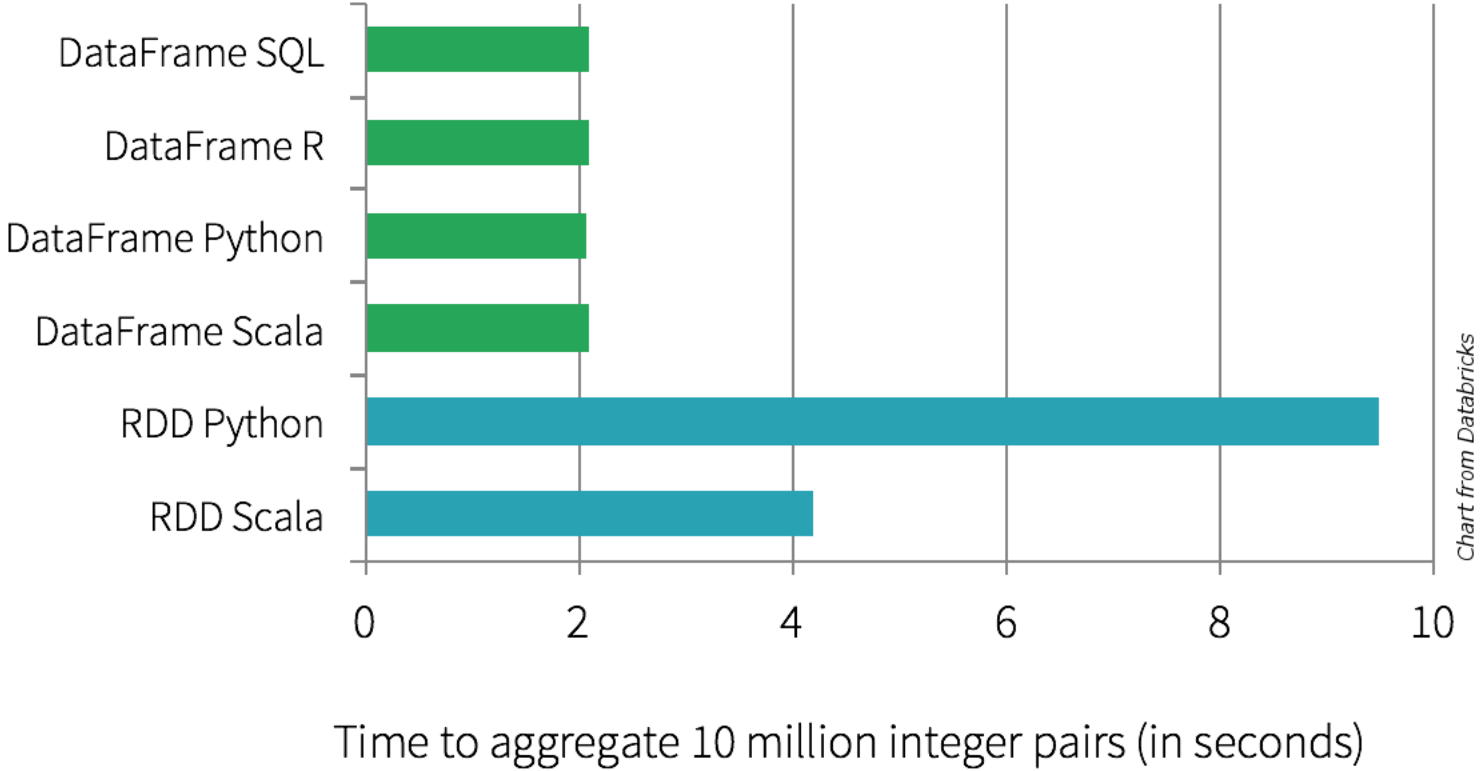
Code example

an Apache Spark code snippet using SQL and DataFrames to query and join different data sources

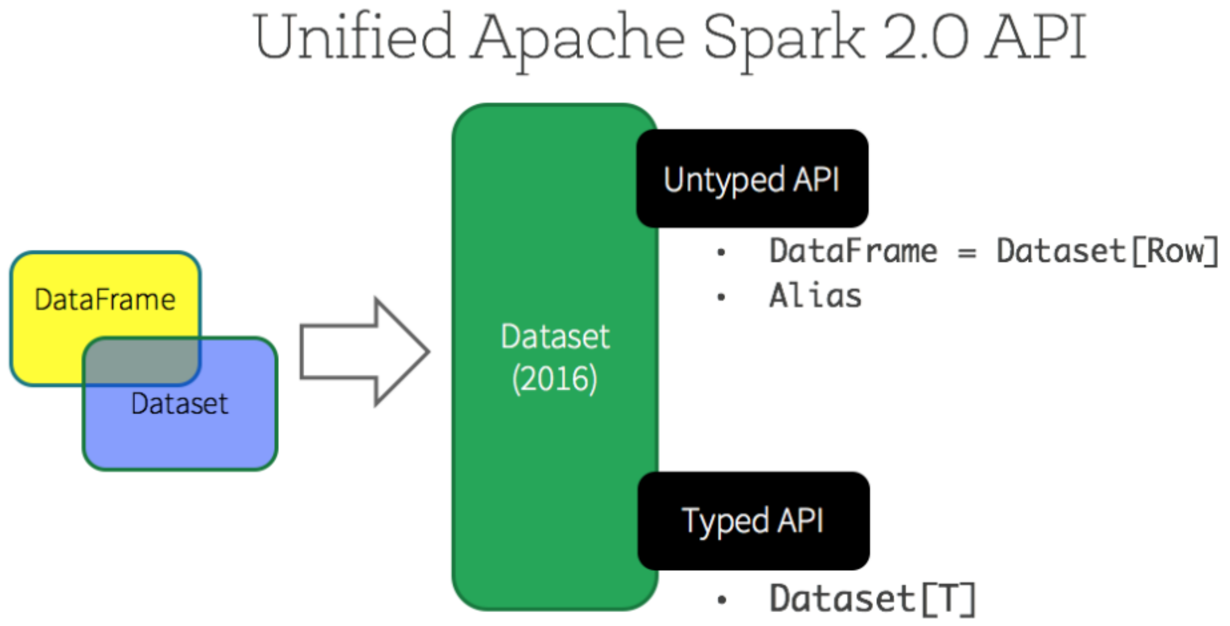
```
# Read JSON file and register temp view

jsonDf = context.jsonFile("s3n://…").createOrReplaceTempView("json")

# Execute SQL query or ...

results = context.sql("""SELECT * FROM people JOIN json … WHERE …""")
```

```
# ... or Use DataFrame APIs

results = peopleDf.join(jsonDf, peopleDf.id == jsonDf.id).filter(...)
```

Performance



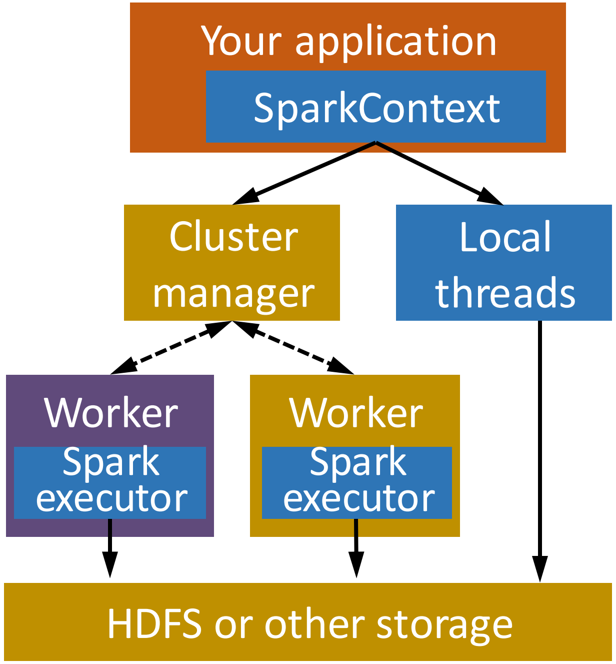
Dataset



Unified Apache Spark 2.0 API
DataFrame
Dataset
Dataset
(2016)
Untyped API
• DataFrame = Dataset[Row]
• Alias
Typed API
• Dataset[T]

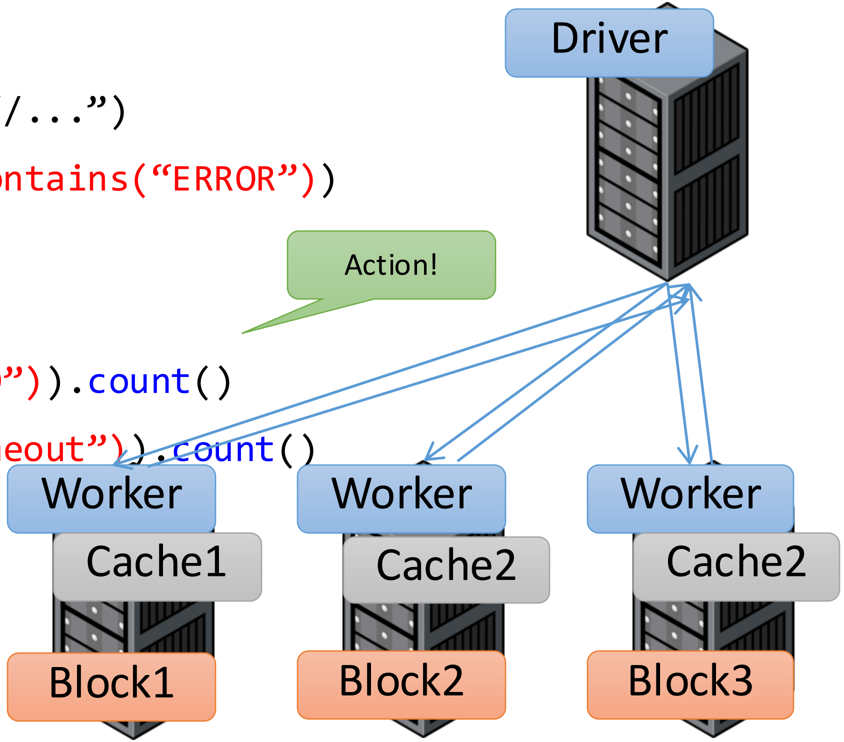
Software Components

- Spark runs as a library in your program (1 instance per app)
- Runs tasks locally or on cluster
  - Mesos, YARN or standalone mode
- Accesses storage systems via Hadoop InputFormat API
  - Can use HBase, HDFS, S3, …



Job example

```
val log = sc.textFile("hdfs://...")
val errors = file.filter(_.contains("ERROR"))
errors.cache()

errors.filter(_.contains("I/O")).count()
errors.filter(_.contains("timeout")).count()
```



Marco Brambilla.

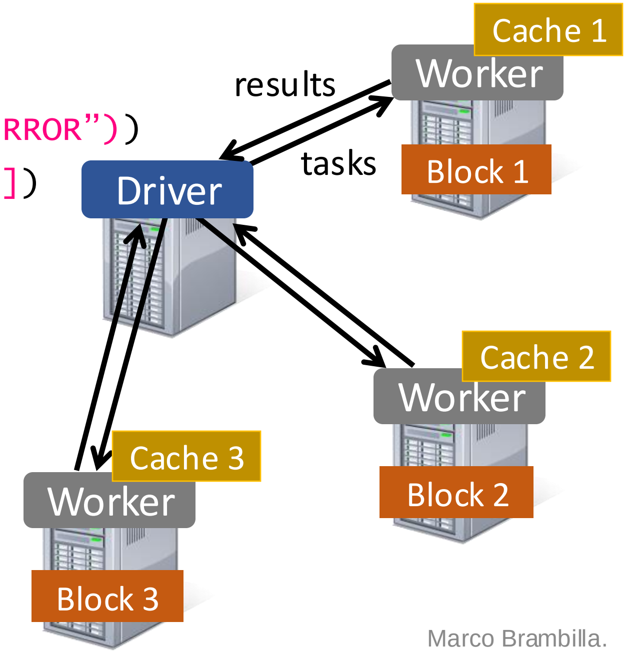
Job Example

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
. . .
```

Full-text search of Wikipedia

- 60GB on 20 EC2 machines
- 0.5 sec vs. 20s for on-disk

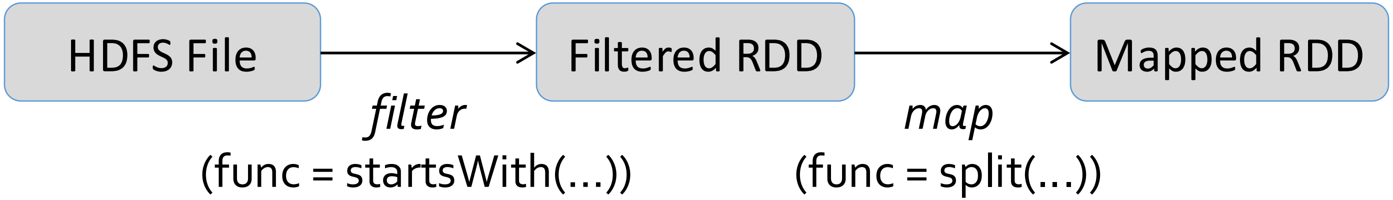


Marco Brambilla.

Fault Recovery

RDDs track *lineage* information that can be used to efficiently recompute lost data

```
msgs = textFile.filter(lambda s: s.startsWith(“ERROR”))
               .map(lambda s: s.split(“\t”)[2])
```

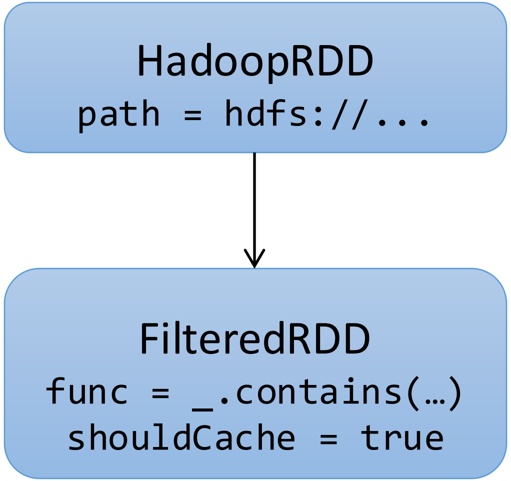


RDD partition-level view

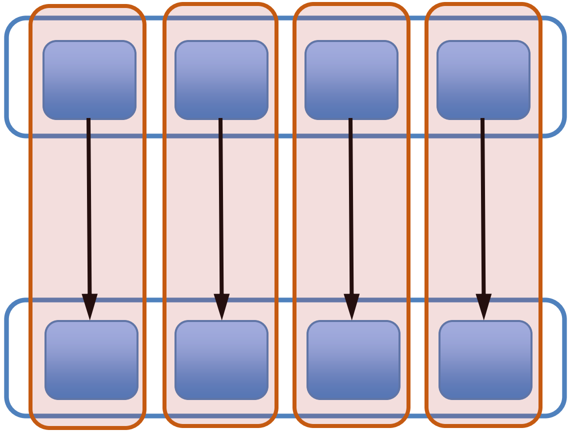
Dataset-level view:

log:

errors:



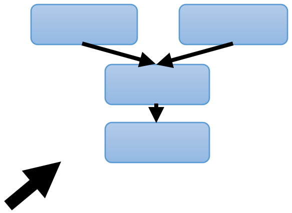
Partition-level view:



Task 1 Task 2 …

Job scheduling

RDD Objects

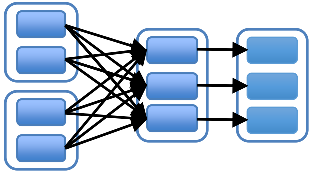


```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
```

build operator DAG

DAG

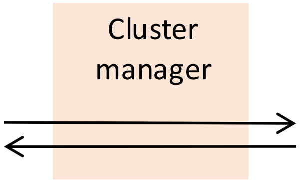
DAGScheduler



split graph into *stages* of tasks
submit each stage as ready

TaskSet

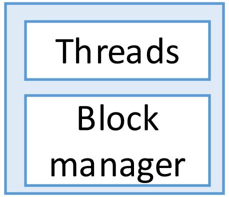
TaskScheduler



launch tasks via cluster manager
retry failed or straggling tasks

Task

Worker



execute tasks

store and serve blocks

Available APIs

You can write in Java, Scala or Python

interactive interpreter: Scala & Python only

standalone applications: any

performance: Java & Scala are faster thanks to static typing

Importing Data

Importers for most data formats and from most database technology

Spark Data Sources

- Cassandra
- Couchbase
- ElasticSearch
- Importing HIVE Tables
- MongoDB
- Neo4j
- Oracle
- Reading Avro Files
- Reading CSV Files
- Reading JSON Files
- Reading LZO Compressed Files
- Reading Parquet Files
- Redis
- Riak Time Series
- Connecting to SQL Databases using JDBC
- Zip Files
- Amazon Redshift
- Amazon S3 with Apache Spark
- Azure storage services
- Azure Cosmos DB
- SQL Data Warehouse

SparkContext

Main entry point to Spark functionality

Available in shell as variable `sc`

In standalone programs, you'd make your own (see later for details)

Creating RDDs

```
# Turn a Python collection into an RDD
>sc.parallelize([1, 2, 3])

# Load text file from local FS, HDFS, or S3
>sc.textFile("file.txt")
>sc.textFile("directory/*.txt")
>sc.textFile("hdfs://namenode:9000/path/file")

# Use existing Hadoop InputFormat (Java/Scala only)
>sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```

Basic Transformations

```
>nums = sc.parallelize([1, 2, 3])

# Pass each element through a function
>squares = nums.map(lambda x: x*x)   // {1, 4, 9}

# Keep elements passing a predicate
>even = squares.filter(lambda x: x % 2 == 0) // {4}

# Map each element to zero or more others
>nums.flatMap(lambda x: => range(x))
    ># => {0, 0, 1, 0, 1, 2}
```

Range object (sequence of numbers 0, 1, …, x-1)

Basic Actions

```
>nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection
>nums.collect() # => [1, 2, 3]

# Return first K elements
>nums.take(2)    # => [1, 2]

# Count number of elements
>nums.count()    # => 3

# Merge elements with an associative function
>nums.reduce(lambda x, y: x + y)  # => 6

# Write elements to a text file
>nums.saveAsTextFile("hdfs://file.txt")
```

Working with Key-Value Pairs

Spark's "distributed reduce" transformations operate on RDDs of key-value pairs

Python:

```python
pair = (a, b)
           pair[0] # => a
           pair[1] # => b
```

Scala:

```scala
val pair = (a, b)
           pair._1 // => a
           pair._2 // => b
```

Java:

```java
Tuple2 pair = new Tuple2(a, b);
pair._1 // => a
pair._2 // => b
```

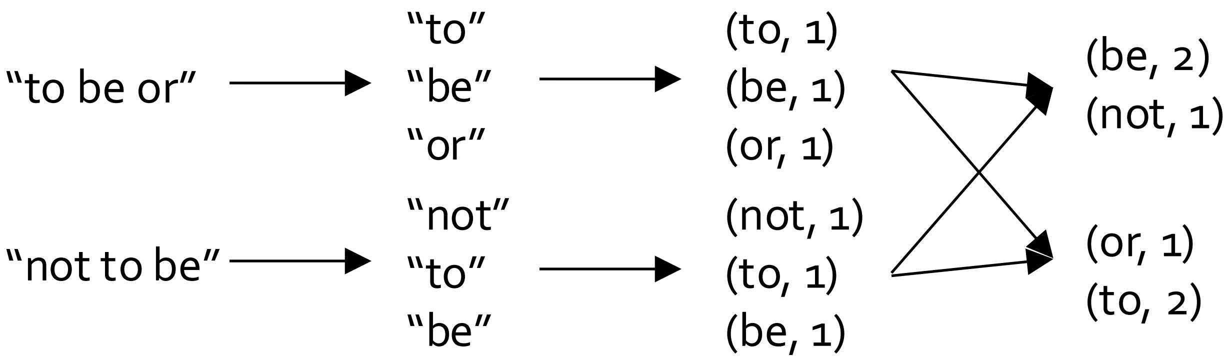
Some Key-Value Operations

```
>pets = sc.parallelize(
  [(“cat”, 1), (“dog”, 1), (“cat”, 2)])
>pets.reduceByKey(lambda x, y: x + y)
                 # => {(cat, 3), (dog, 1)}
>pets.groupByKey() # => {(cat, [1, 2]), (dog, [1])}
>pets.sortByKey()  # => {(cat, 1), (cat, 2), (dog, 1)}
```

`reduceByKey` also automatically implements combiners on the map side

Example: Word Count

```
>lines = sc.textFile("hamlet.txt")

>counts = lines.flatMap(lambda line: line.split(" "))
               .map(lambda word: (word, 1))
               .reduceByKey(lambda x, y: x + y)
```



Other Key-Value Operations

```
>visits = sc.parallelize([ ("index.html", "1.2.3.4"),
                           ("about.html", "3.4.5.6"),
                           ("index.html", "1.3.3.1") ])

>pageNames = sc.parallelize([ ("index.html", "Home"),
                              ("about.html", "About") ])

>visits.join(pageNames)
# ("index.html", ("1.2.3.4", "Home"))
# ("index.html", ("1.3.3.1", "Home"))
# ("about.html", ("3.4.5.6", "About"))

>visits.cogroup(pageNames)
# ("index.html", (["1.2.3.4", "1.3.3.1"], ["Home"]))
# ("about.html", (["3.4.5.6"], ["About"]))
```

Setting the Level of Parallelism

All the pair RDD operations take an optional second parameter for number of tasks

```
>words.reduceByKey(lambda x, y: x + y, 5)
>words.groupByKey(5)
>visits.join(pageViews, 5)
```

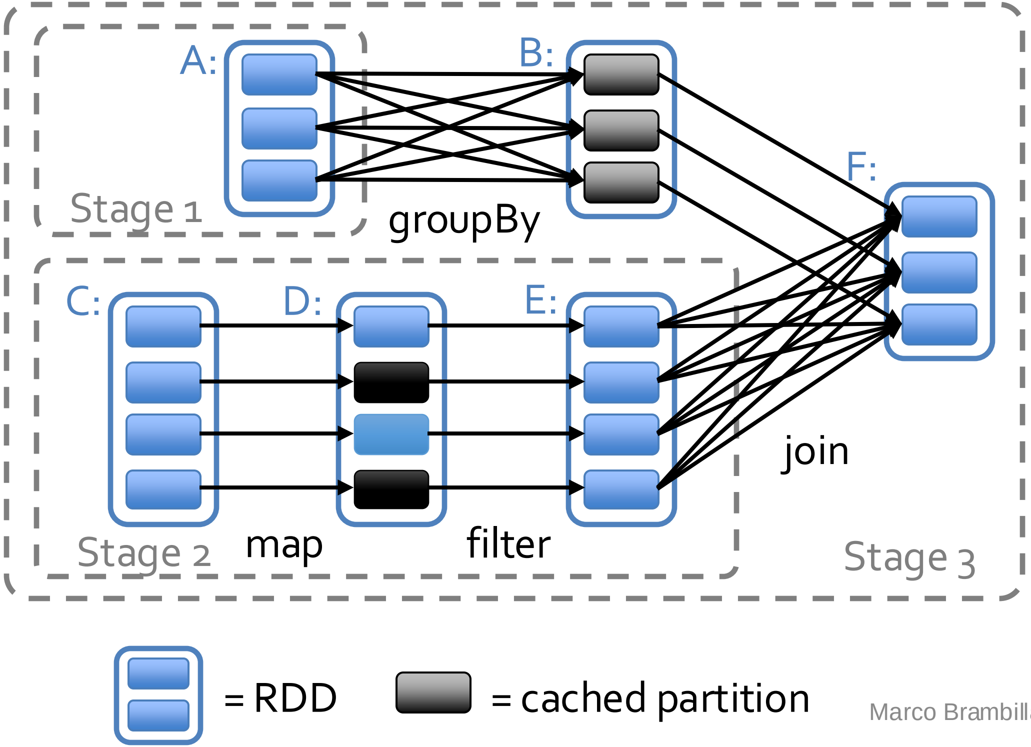
Under The Hood: DAG Scheduler

General task graphs

Automatically pipelines functions

Data locality aware

Partitioning aware to avoid shuffles



Marco Brambilla.

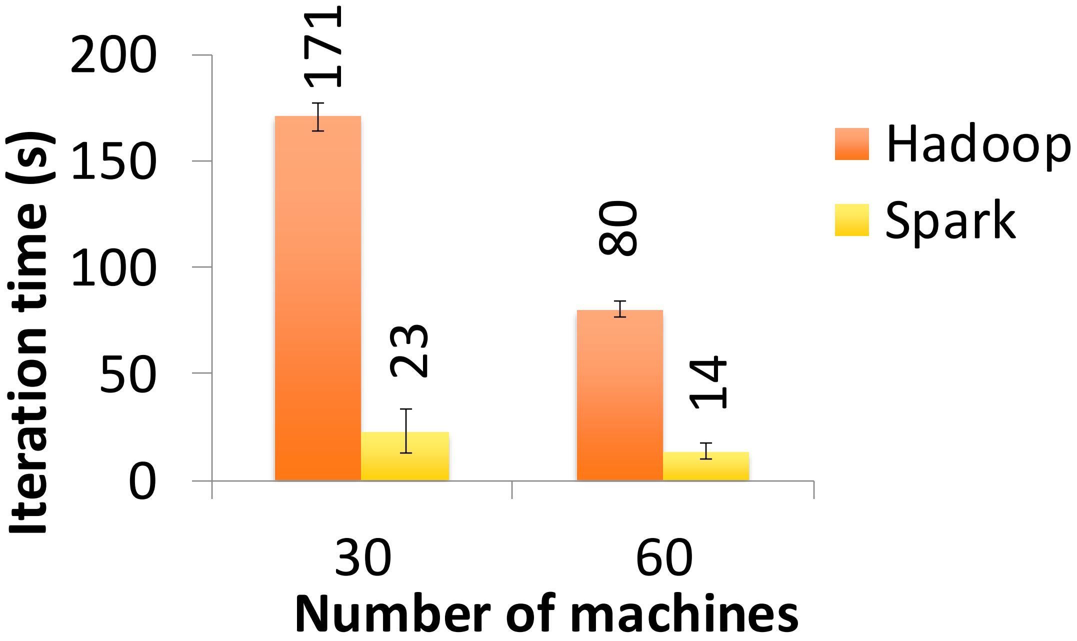
More RDD Operators

map
filter
groupBy
sort
union
join
leftOuterJoin
rightOuterJoin

reduce
count
fold
reduceByKey
groupByKey
cogroup
cross
zip

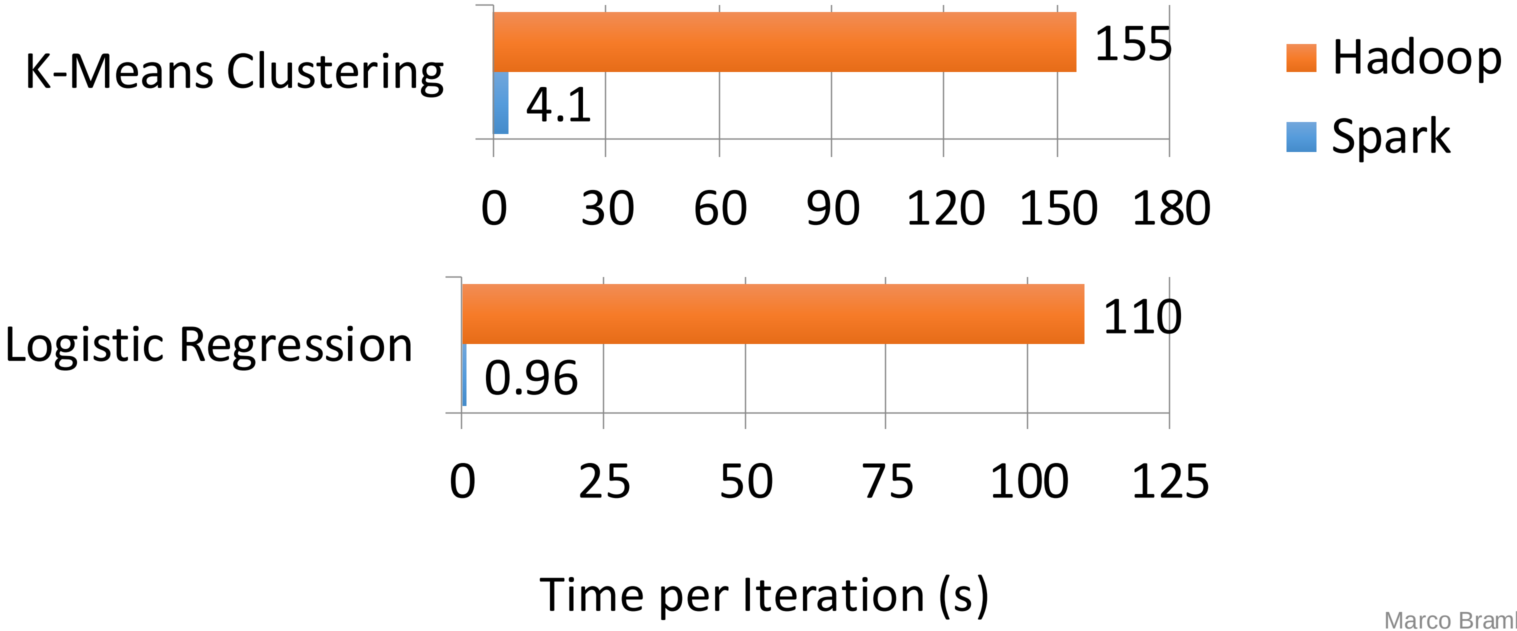
sample
take
first
partitionBy
mapWith
pipe
save ...

PageRank Performance



Marco Brambilla.

Other Iterative Algorithms



Marco Brambilla.

On-Disk Sort Record:
Time to sort 100TB

Record:
Hadoop

2100 machines

72 minutes

Record:
Spark

207 machines

23 minutes

Also sorted 1PB in 4 hours

Available File Formats

- -Text / CSV
- -JSON
- -SequenceFile
  - •binary key/value pair format
- -Avro
- -Parquet
- -Data Frames
- -ORC
  - optimized row columnar format

AVRO

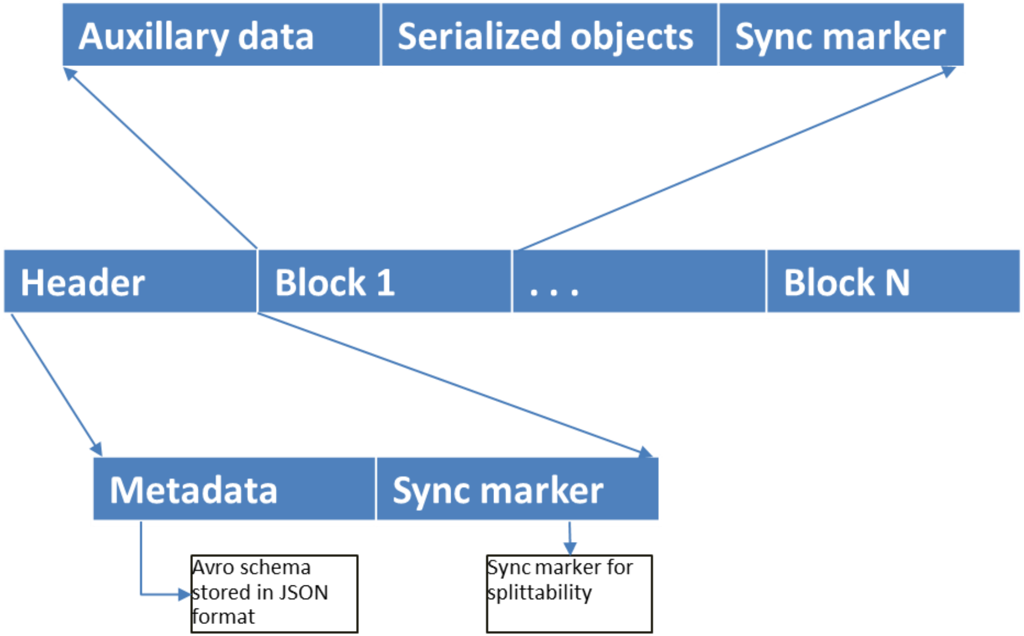
- Language neutral data serialization system
  - Write a file in python and read it in C
- AVRO data is described using language independent schema
- AVRO schemas are usually written in JSON and data is encoded in binary format
- Supports schema evolution
  - producers and consumers at different versions of schema
- Supports compression and are splittable

Avro – File structure and example

Sample AVRO schema in JSON format

```
{
  "type" : "record",
  "name" : "tweets",
  "fields" : [ {
    "name" : "username",
    "type" : "string",
}, {
    "name" : "tweet",
    "type" : "string",
}, {
    "name" : "timestamp",
    "type" : "long",
} ],
  "doc:" : "schema for storing tweets"
}
```

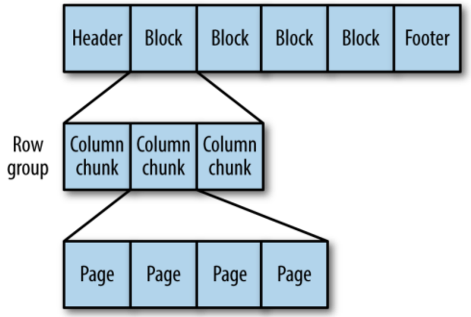
Avro file structure



Marco Brambilla.

Parquet file structure & Configuration

Internal structure of parquet file



Configurable parquet parameters

Property name	Default value	Description
parquet.block.size	128 MB	The size in bytes of a block (row group).
parquet.page.size	1MB	The size in bytes of a page.
parquet.dictionary.page.size	1MB	The maximum allowed size in bytes of a dictionary before falling back to plain encoding for a page.
parquet.enable.dictionary	true	Whether to use dictionary encoding.
parquet.compression	UNCOMPRESSED	The type of compression: UNCOMPRESSED, SNAPPY, GZIP & LZO

In summation, Parquet is state-of-the-art, open-source columnar format the supports *most* of processing frameworks and is optimized for high compression and high scan efficiency

DataFrame

noun – [dey-tuh-freym]

1. A distributed collection of rows organized into named columns.
2. An abstraction for selecting, filtering, aggregating and plotting structured data (*cf. R*, *Pandas*).
3. Archaic: Previously SchemaRDD (*cf. Spark* < *1.3*).

DataFrame

```
ctx = new HiveContext()
users = ctx.table("users")
young = users.where(users("age") < 21)
println(young.count())
```

- A distributed collection of rows with the same schema (RDDs suffer from type erasure)
- Can be constructed from external data sources or RDDs into essentially an RDD of Row objects (SchemaRDDs as of Spark < 1.3)
- Supports relational operators (e.g. *where*, *groupby*) as well as Spark operations.
- Evaluated lazily → unmaterialized *logical* plan

Data Model

- Nested data model
- Supports
  - primitive SQL types (boolean, integer, double, decimal, string, data, timestamp)
  - complex types (structs, arrays, maps, and unions)
  - also user defined types.
- First class support for complex data types

DataFrame Operations

- Relational operations (select, where, join, groupBy) via a DSL
- Operators take *expression* objects
- Operators build up an abstract syntax tree (AST), which is then optimized by *Catalyst*.

```
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

- Alternatively, register as temp SQL table and perform traditional SQL query strings

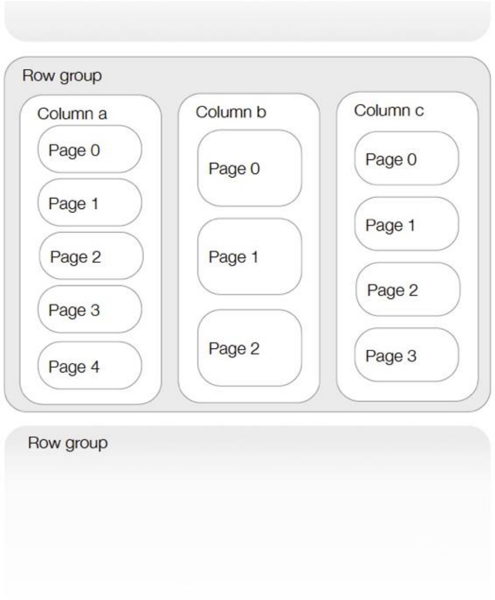
```
users.where(users("age") < 21)
     .registerTempTable("young")
ctx.sql("SELECT count(*), avg(age) FROM young")
```

Parquet

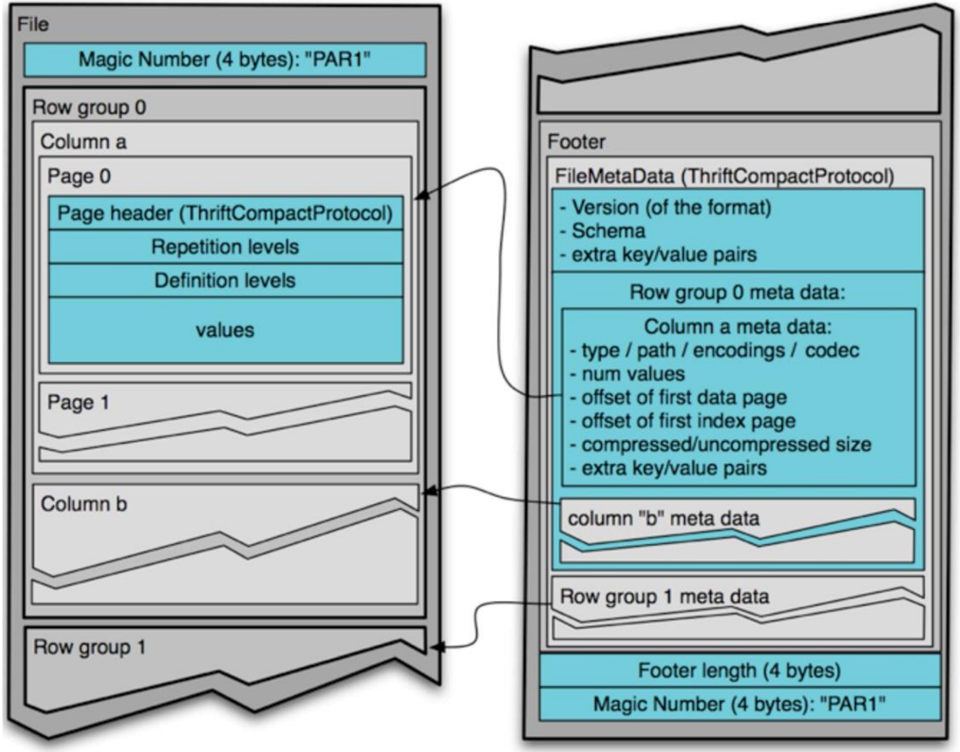
Apache Parquet is a **columnar storage** format available to any project in the Hadoop/Spark ecosystem, regardless of the choice of data processing framework, data model or programming language.

Parquet Format

- **Row group:** A group of rows in columnar format.
  - Max size buffered in memory while writing.
  - One (or more) per split while reading.
  - roughly: 50MB < row group < 1 GB
- **Column chunk:** The data for one column in a row group.
  - Column chunks can be read independently for efficient scans.
- **Page:** Unit of access in a column chunk.
  - Should be big enough for compression to be efficient.
  - Minimum size to read to access a single record (when index pages are available).
  - roughly: 8KB < page < 1MB



Parquet Format – details



- **Layout:**

  Row groups in columnar format. A footer contains column chunks offset and schema.

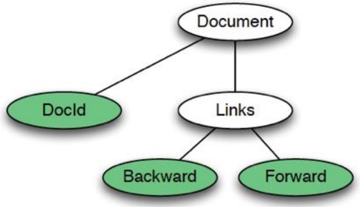
- **Language independent:**

  Well defined format. Hadoop

Nested Record Shredding & Assembly

Schema:

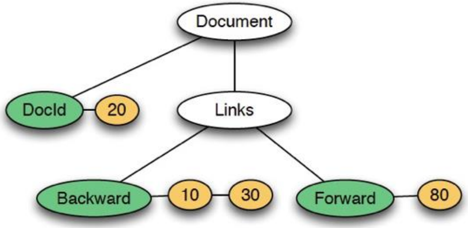
```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward;
  }
}
```



Columns	Max rep. level	Max def. level
DocId	0	0
Links.Backward	1	2
Links.Forward	1	2

Record:

```
DocId: 20
Links
  Backward: 10
  Backward: 30
  Forward:  80
```



Column	Value	R	D
DocId	20	0	0
Links.Backward	10	0	2
Links.Backward	30	1	2
Links.Forward	80	0	2

Parquet

columnar storage format

key strength is to store nested data in truly columnar format using definition and repetition levels[1]

Nested schema

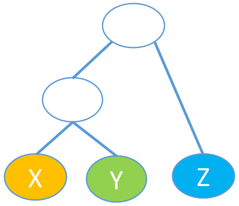
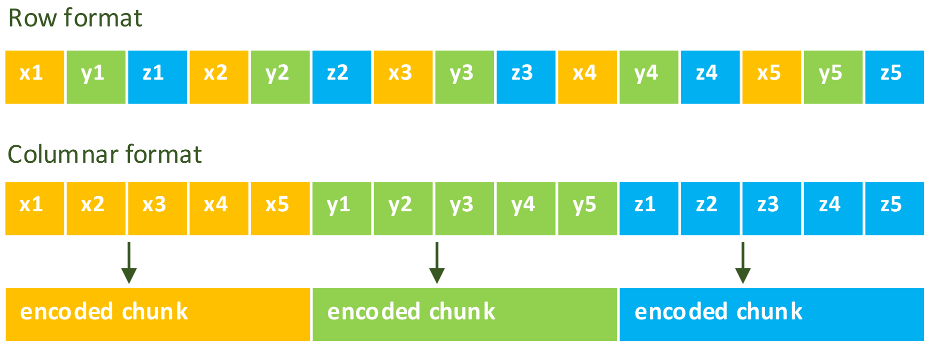


Table representation

X	Y	Z
x1	y1	z1
x2	y2	z2
x3	y3	z3
x4	y4	z4
x5	Y5	z5

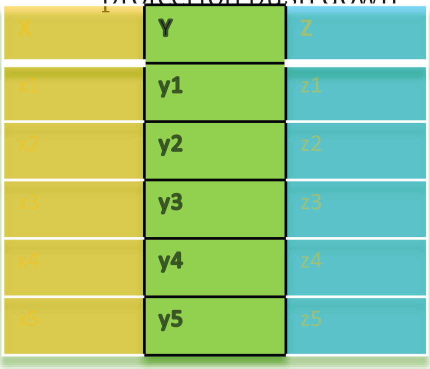


(1) Dremel made simple with parquet - https://blog.twitter.com/2013/dremel-made-simple-with-parquet

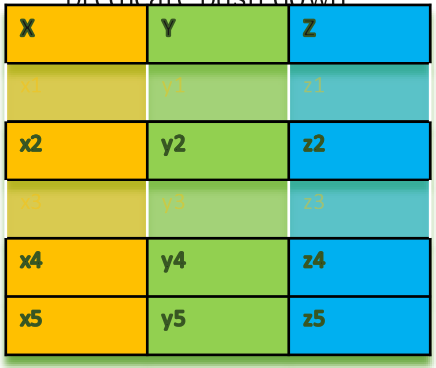
Optimizations – CPU and I/O

Statistics for filtering and query optimization

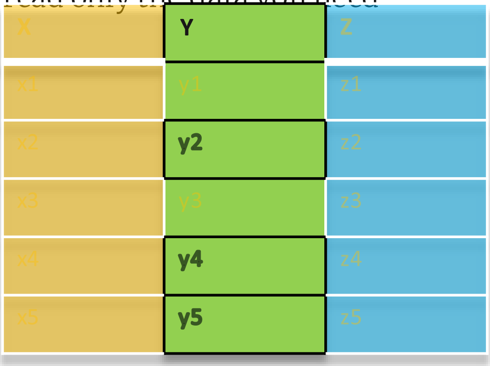
projection push down



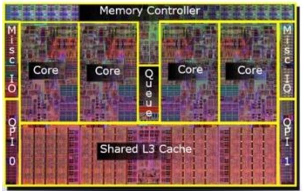
predicate push down



read only the data you need



Minimizes CPU cache misses



SLOW

cache misses costs cpu cycles

Encoding

- -Delta Encoding:
  - -E.g timestamp can be encoded by storing first value and the delta between subsequent values which tend to be small due to temporal validity
- -Prefix Encoding:
  - -delta encoding for strings
- -Dictionary Encoding:
  - -Small set of values, e.g post code, ip addresses etc
- -Run Length Encoding:
  - -repeating data

Read Less Data

Columnar organization

- Encoding: make the data smaller
- Column projection: read only the columns you need

Row group filtering

- Use footer stats to eliminate row groups
- Use dictionary pages to eliminate row groups

Page filtering

- Use page stats to eliminate pages

Encoding

Bit packing:

- Small integers encoded in the minimum bits required
- Useful for repetition level, definition levels and dictionary keys

Run Length Encoding:

- Used in combination with bit packing
- Cheap compression
- Works well for definition level of sparse columns.

Dictionary encoding:

- Useful for columns with few (< 50,000) distinct values
- When applicable, compresses better and faster than heavyweight algorithms (gzip, lzo, snappy)

Extensible: Defining new encodings is supported by the format

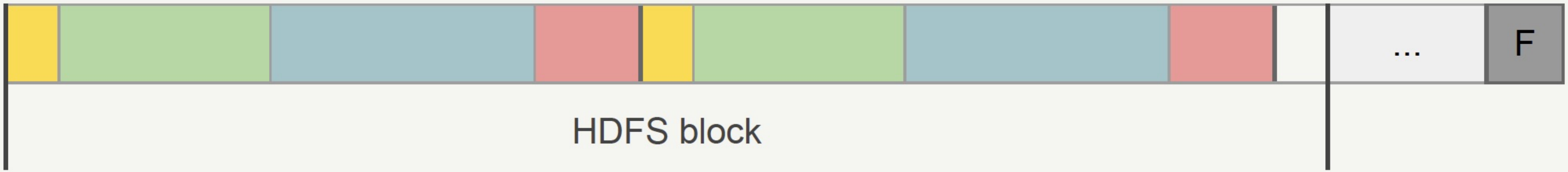
Parquet 2.0

- More encodings: compact storage **without heavyweight compression**
- Delta encodings: for integers, strings and sorted dictionaries.
- Improved encoding for strings and boolean

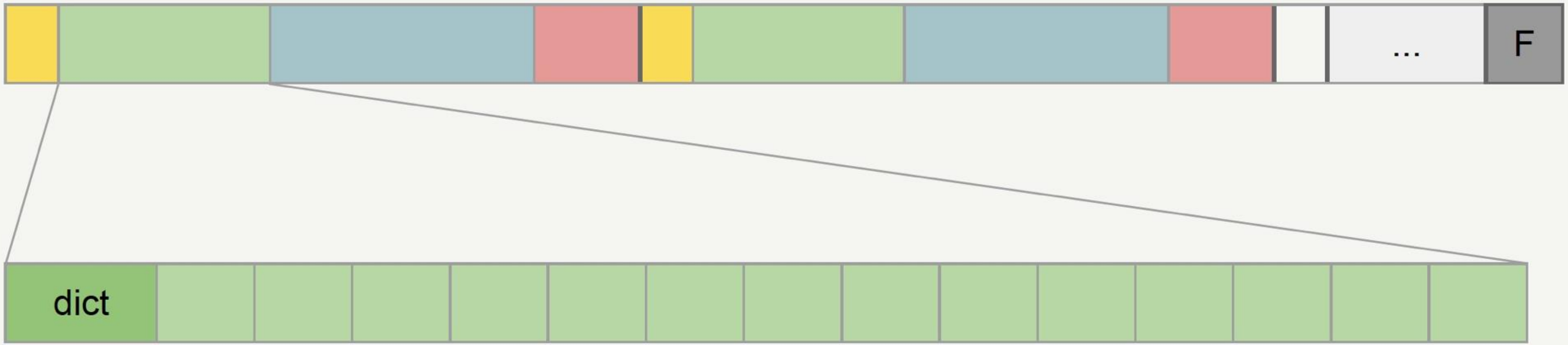
- Statistics: to be used by query planners and predicate pushdown.
- New page format: to facilitate skipping ahead at a more granular level.

.. On HDFS: Row Groups

A	B	C	D
a1	b1	c1	d1
...	...	...	...
aN	bN	cN	dN
...	...	...	...



Column Chunks and Pages



Dictionary is a compact list of all the values.

- Search term missing? Skip the row group
- Like a bloom filter without false positives

Spark SQL

is about more than SQL.

Challenges and Solutions

Challenges

- Perform ETL to and from various (semi- or unstructured) data sources
- Perform advanced analytics (e.g. machine learning, graph processing) that are hard to express in relational systems.

Solutions

- A *DataFrame* API that can perform relational operations on both external data sources and Spark's built-in RDDs.
- A highly extensible optimizer, *Catalyst*, that uses features of Scala to add composable rule, control code gen., and define extensions.

Spark SQL: Declarative BigData Processing

Let Developers Create and Run Spark Programs Faster:

- Write less code
- Read less data
- Let the optimizer do the hard work

Write Less Code: Compute an Average

hadoop

```
private IntWritable one =
  new IntWritable(1)
private IntWritable output =
  new IntWritable()
proctected void map(
    LongWritable key,
    Text value,
    Context context) {
  String[] fields = value.split("\t")
  output.set(Integer.parseInt(fields[1]))
  context.write(one, output)
}

IntWritable one = new IntWritable(1)
DoubleWritable average = new DoubleWritable()

protected void reduce(
    IntWritable key,
    Iterable<IntWritable> values,
    Context context) {
  int sum = 0
  int count = 0
  for(IntWritable value : values) {
     sum += value.get()
     count++
    }
  average.set(sum / (double) count)
  context.Write(key, average)
}
```

Spark

```
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [x.[1], 1])) \
   .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
   .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
   .collect()
```

Write Less Code: Compute an Average

Using RDDs

```
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [int(x[1]), 1])) \
   .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
   .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
   .collect()
```

Using SQL

```
SELECT name, avg(age)
FROM people
GROUP BY name
```

Using Pig

```
P = load '/people' as (name, name);
G = group P by name;
R = foreach G generate … AVG(G.age);
```

Using DataFrames

```
sqlCtx.table("people") \
   .groupBy("name") \
   .agg("name", avg("age")) \
   .collect()
```

Seamlessly Integrated: RDDs

Internally, DataFrame execution is done with Spark RDDs making interoperation with outside sources and custom algorithms easy.

External Input

```
def buildScan(
    requiredColumns: Array[String],
    filters: Array[Filter]):
RDD[Row]
```

Custom Processing

```
queryResult.rdd.mapPartitions { iter =>

  … Your code here …

}
```

Extensible Input & Output

Spark's Data Source API allows optimizations like column pruning and filter pushdown into custom data sources.

Built-In

{ JSON }

JDBC

Parquet

HIVE

MySQL

PostgreSQL

HDFS

amazon web services S3

H2

External

AVRO

CSV

dBase

APACHE HBASE

elasticsearch.

cassandra

amazon web services Amazon Redshift

memsql

and more…

Marco Brambilla.

Seamlessly Integrated

Embedding in a full programming language makes UDFs trivial and allows composition using functions.

```
zipToCity = udf(lambda city: <custom logic here>)

def add_demographics(events):
   u = sqlCtx.table("users")
   events \
     .join(u, events.user_id == u.user_id) \
     .withColumn("city", zipToCity(df.zip))
```

Takes and returns a DataFrame

About Spark SQL

Spark SQL

- Part of the core distribution since Spark 1.0 (April 2014)
- Runs SQL / HiveQL queries, optionally alongside or replacing existing Hive deployments

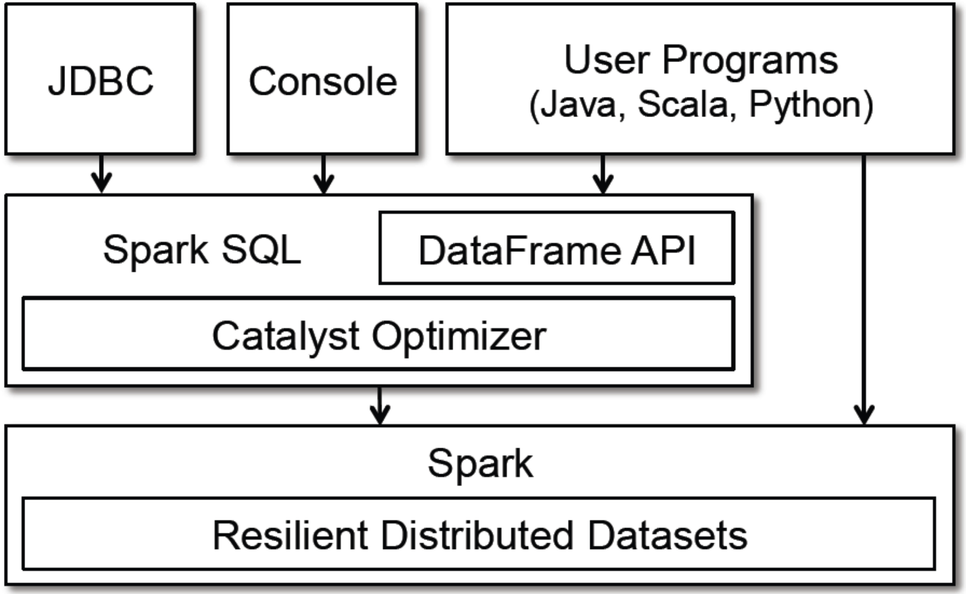
```sql
SELECT COUNT(*)
FROM hiveTable
WHERE hive_udf(data)
```

About Spark SQL

Originally called SHARK

- Can only be used to query external data in Hive catalog → limited data sources
- Can only be invoked via SQL string from Spark→ error prone
- Hive optimizer tailored for MapReduce → difficult to extend

Programming Interface



Advantages over Relational Query Languages

- Holistic optimization across functions composed in different languages.
- Control structures (e.g. *if*, *for*)
- Logical plan analyzed *eagerly* → identify code errors associated with data *schema* issues on the fly.

Querying Native Datasets

- Infer column names and types directly from data objects (via reflection in Java and Scala and data sampling in Python, which is dynamically typed)

```
case class User(name: String, age: Int)
```

- Native objects accessed in-place to avoid expensive data format transformation.
- Benefits:
  - Run relational operations on existing Spark programs.
  - Combine RDDs with external structured data

Columnar storage with *hot* columns cached in memory

User-Defined Functions (UDFs)

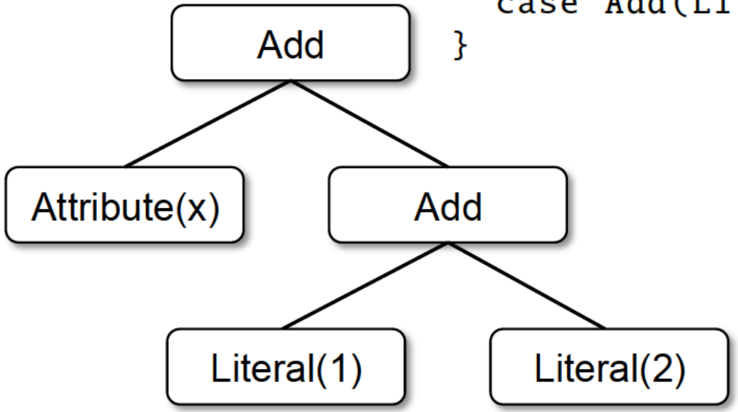
- Easy extension of limited operations supported.
- Allows inline registration of UDFs

  Compare with Pig, which requires the UDF to be written in a Java package that's loaded into the Pig script.
- Can be defined on simple data types or entire tables.
- UDFs available to other interfaces after registration

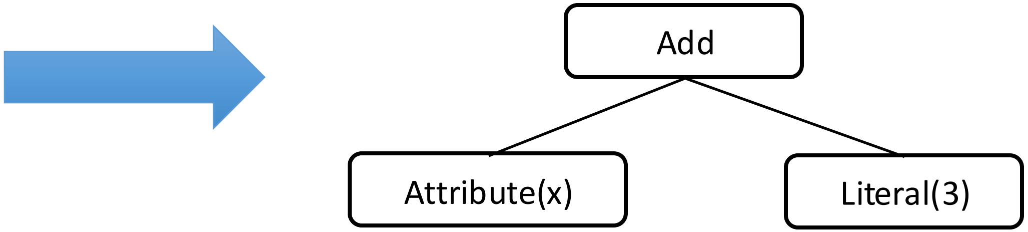
```
val model: LogisticRegressionModel = ...

ctx.udf.register("predict",
  (x: Float, y: Float) => model.predict(Vector(x, y)))

ctx.sql("SELECT predict(age, weight) FROM users")
```

Query Optimization: Catalyst

```
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
}
```



x + (1 + 2)

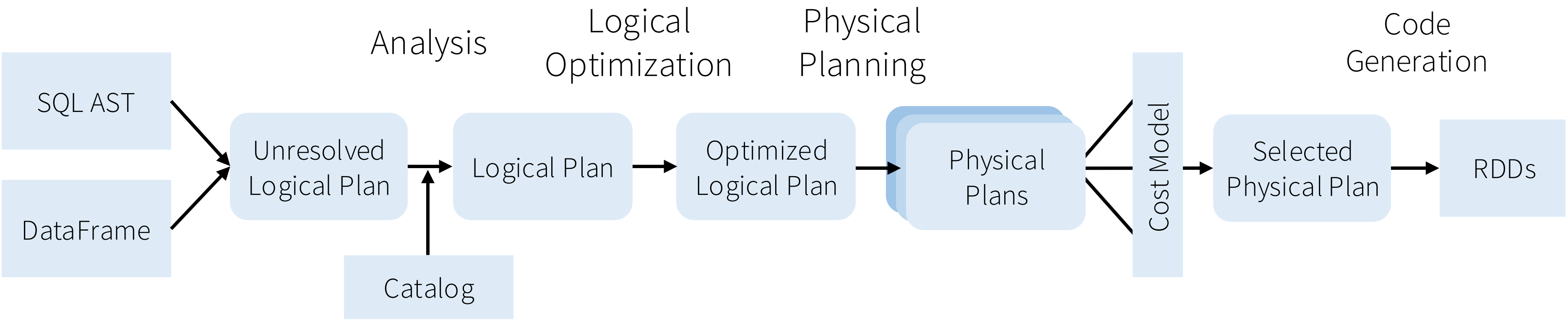


x + 3

Catalyst Rules

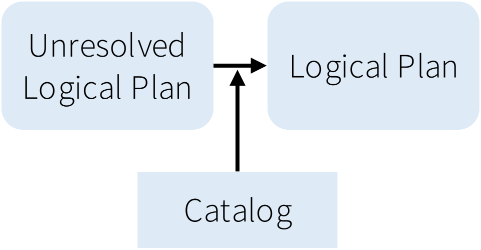
- *Pattern matching* functions that transform subtrees into specific structures.
  - *Partial function*—skip over subtrees that do not match → no need to modify existing rules when adding new types of operators.
- Multiple patterns in the same *transform* call.
- May take multiple *batches* to reach a *fixed point*.
- *transform* can contain arbitrary Scala code.

Plan Optimization & Execution



DataFrames and SQL share the same optimization/execution pipeline

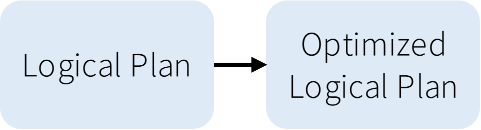
Analysis



SELECT *col* FROM *sales*

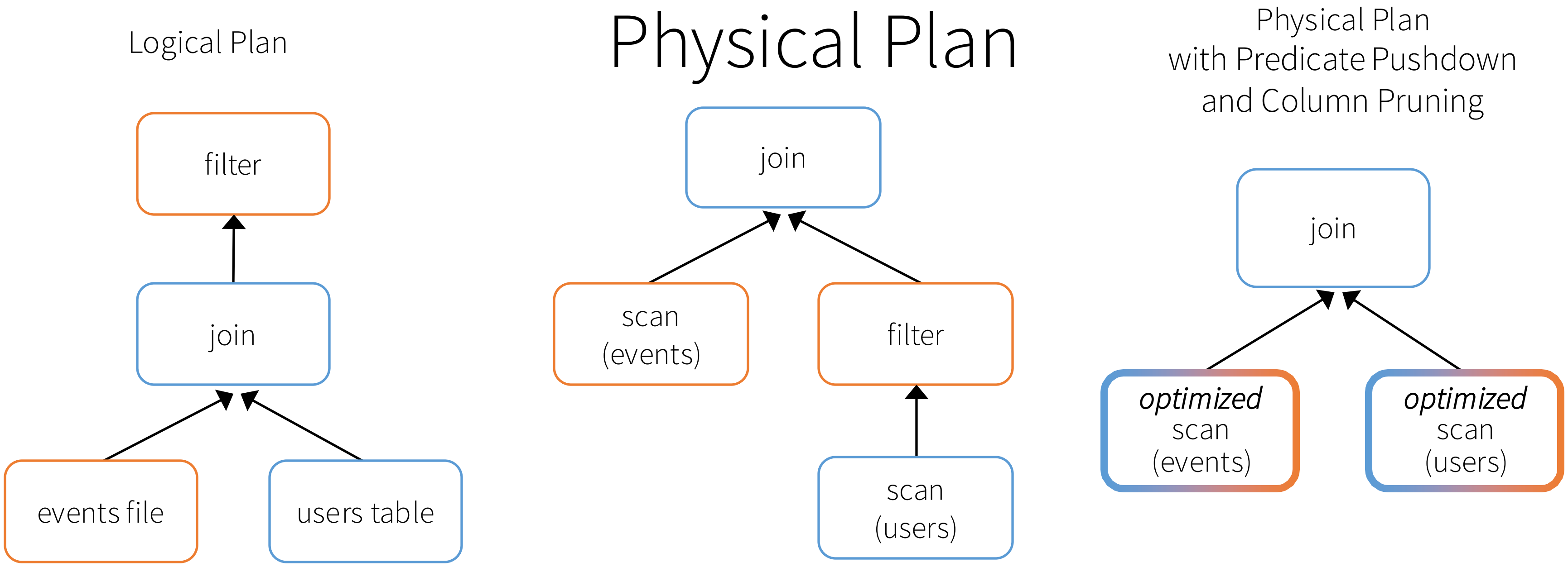
- An attribute is *unresolved* if its type is not known or it's not matched to an input table.
- To resolve attributes:
  - Look up relations by name from the catalog.
  - Map named attributes to the input provided given operator's children.
  - UID for references to the same value
  - Propagate and coerce types through expressions (e.g. 1 + *col*)

Logical Optimization



- Applies standard rule-based optimization (constant folding, predicate-pushdown, projection pruning, null propagation, boolean expression simplification, etc)
- 800LOC

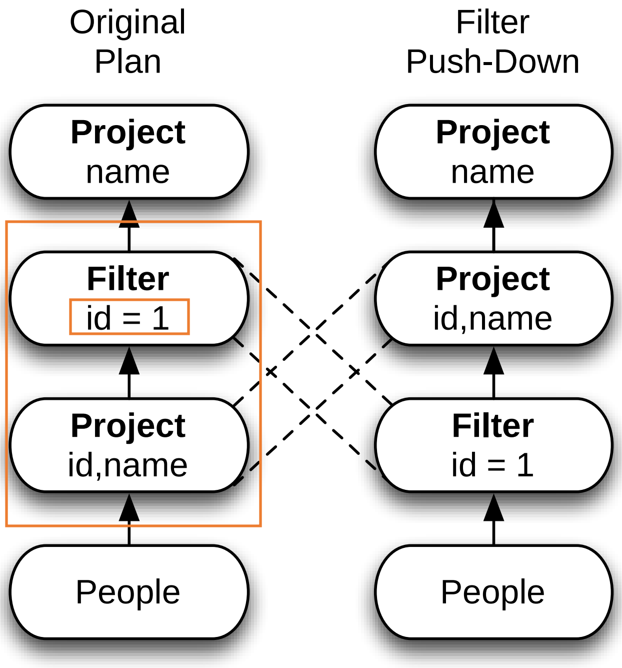
```python
def add_demographics(events):
    u = sqlCtx.table("users")                      # Load partitioned Hive table
    events \
      .join(u, events.user_id == u.user_id) \      # Join on user_id
      .withColumn("city", zipToCity(df.zip))       # Run udf to add city column
events = add_demographics(sqlCtx.load("/data/events", "parquet"))
training_data = events.where(events.city == "Melbourne").select(events.timestamp).collect()
```



Marco Brambilla.

An Example Catalyst Transformation

1. Find filters on top of projections.
2. Check that the filter can be evaluated without the result of the project.
3. If so, switch the operators.



Marco Brambilla.

Code Generation

```
def compile(node: Node): AST = node match {
  case Literal(value) => q"$value"
  case Attribute(name) => q"row.get($name)"
  case Add(left, right) =>
    q"${compile(left)} + ${compile(right)}"
}
```

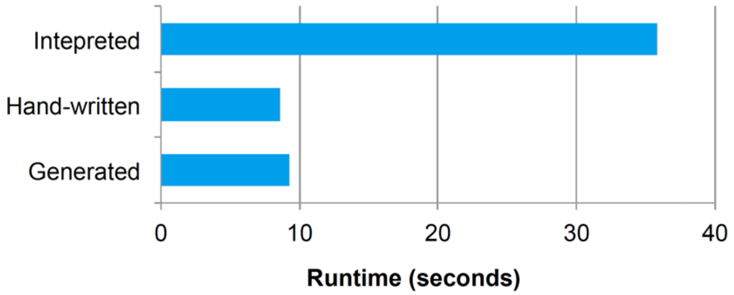


Figure 4: A comparision of the performance evaluating the expresion x+x+x, where x is an integer, 1 billion times.

- Relies on Scala's *quasiquotes* to simplify code gen.
- Catalyst transforms a SQL tree into an abstract syntax tree (AST) for Scala code to eval expr and generate code
- 700LOC

Extensions

Data Sources

- must implement a *createRelation* function that takes a set of key-value params and returns a *BaseRelation* object.
- E.g. CSV, Avro, Parquet, JDBC

User-Defined Types (UDTs)

- *Map* user-defined types to structures composed of Catalyst's built-in types.

```
class PointUDT extends UserDefinedType[Point] {
  def dataType = StructType(Seq( // Our native structure
    StructField("x", DoubleType),
    StructField("y", DoubleType)
  ))
  def serialize(p: Point) = Row(p.x, p.y)
  def deserialize(r: Row) =
    Point(r.getDouble(0), r.getDouble(1))
}
```

Advanced Analytics Features

Schema Inference for Semistructured Data

JSON

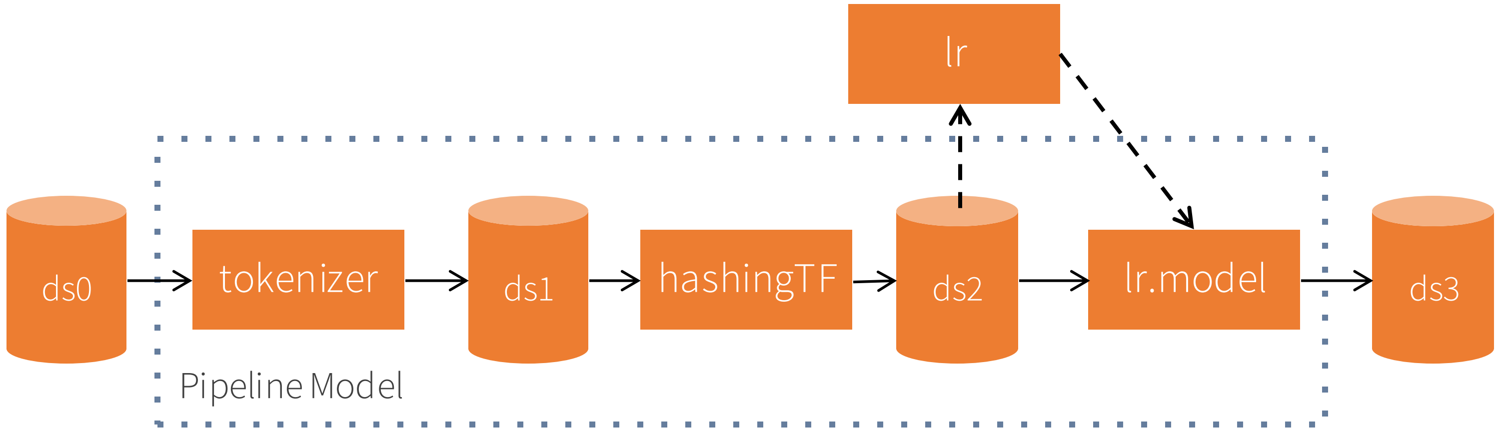
- Automatically infers schema from a set of records, in one pass or sample
- A tree of STRUCT types, each of which may contain atoms, arrays, or other STRUCTs.
- Find the most appropriate type for a field based on all data observed in that column. Determine array element types in the same way.
- Merge schemata of single records in one *reduce* operation.
- Same trick for Python typing

```
{
  "text": "This is a tweet about #Spark",
  "tags": ["#Spark"],
  "loc": {"lat": 45.1, "long": 90}
}

{
  "text": "This is another tweet",
  "tags": [],
  "loc": {"lat": 39, "long": 88.5}
}

{
  "text": "A #tweet without #location",
  "tags": ["#tweet", "#location"]
}
```

```
text STRING NOT NULL,
tags ARRAY<STRING NOT NULL> NOT NULL,
loc STRUCT<lat FLOAT NOT NULL, long FLOAT NOT NULL>
```

Spark MLlib Pipelines

```
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol="words", outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.01)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

df = sqlCtx.load("/path/to/data")
model = pipeline.fit(df)
```



About Performance Again..

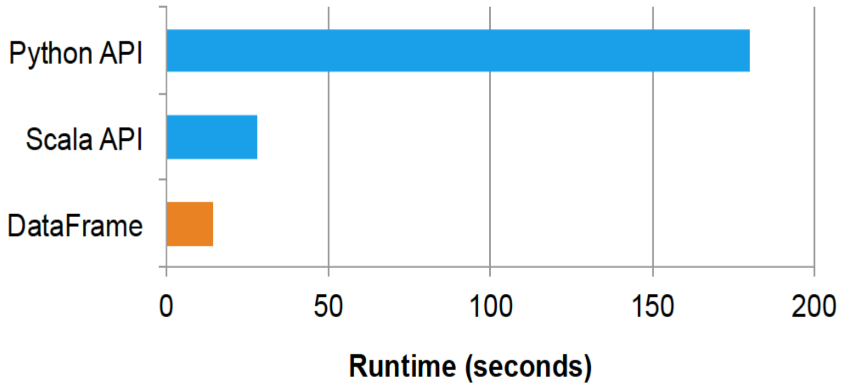


Figure 9: Performance of an aggregation written using the native Spark Python and Scala APIs versus the DataFrame API.

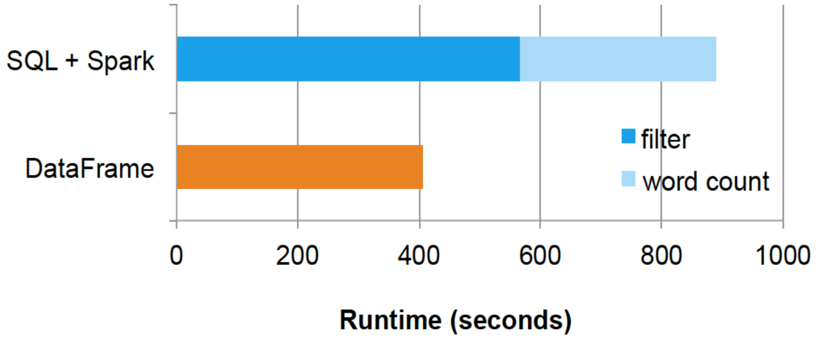
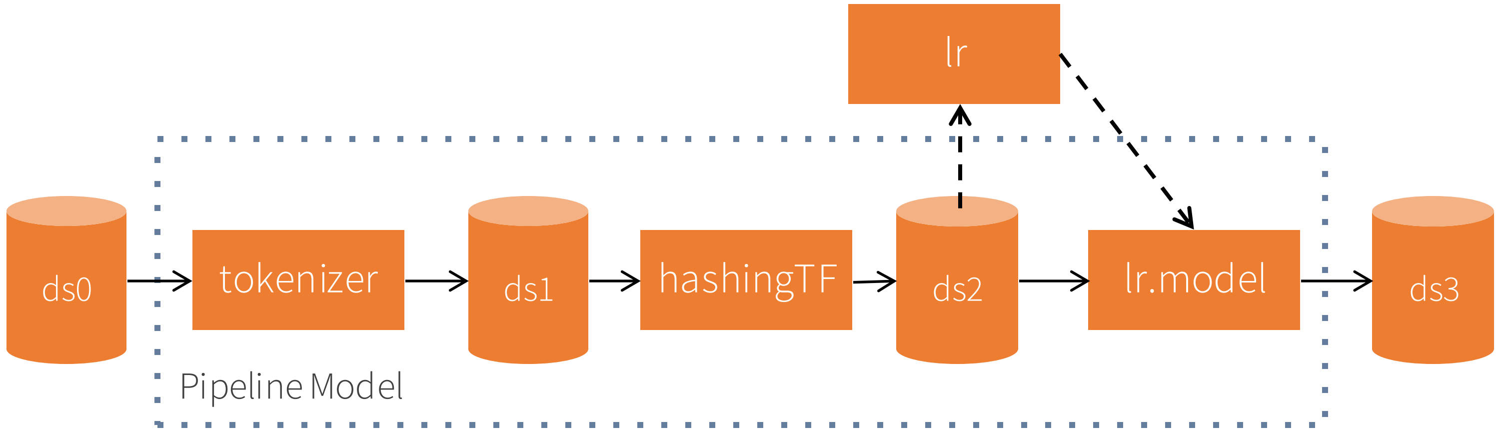


Figure 10: Performance of a two-stage pipeline written as a separate Spark SQL query and Spark job (above) and an integrated DataFrame job (below).

Spark MLlib Pipelines

```
tokenizer = Tokenizer(inputCol="text", outputCol="words”)
hashingTF = HashingTF(inputCol="words", outputCol="features”)
lr = LogisticRegression(maxIter=10, regParam=0.01)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

df = sqlCtx.load("/path/to/data")
model = pipeline.fit(df)
```



http://spark.apache.org/

SYSTEMS AND METHODS FOR BIG AND UNSTRUCTURED DATA

Thanks

Marco Brambilla

marco.brambilla@polimi.it

@marcobrambi